The Noise Protocol Framework

Trevor Perrin (noise@trevp.net)

Revision 33draft, 2017-09-22

Contents

1. Introduction **3**

2. Overview **3**
2.1. Terminology . . . 3
2.2. Overview of handshake state machine . . . 3

3. Message format **5**

4. Crypto functions **6**
4.1. DH functions . . . 7
4.2. Cipher functions . . . 7
4.3. Hash functions . . . 8

5. Processing rules **8**
5.1 The `CipherState` object . . . 9
5.2. The `SymmetricState` object . . . 10
5.3. The `HandshakeState` object . . . 11

6. Prologue **14**

7. Handshake patterns **14**
7.1. Pattern validity . . . 16
7.2. One-way patterns . . . 16
7.3. Interactive patterns . . . 17
7.4. Payload security properties . . . 19
7.5. Identity hiding . . . 23

8. Protocol names and modifiers **26**
8.1. Handshake pattern name section . . . 26
8.2. Cryptographic algorithm name sections . . . 27

9. Pre-shared symmetric keys **27**
9.1. Cryptographic functions . . . 27

9.2. Handshake tokens 27
9.3. Validity rule 28
9.4. Pattern modifiers 28

10. Fallback protocols **32**
10.1. Fallback patterns 32
10.2. Indicating fallback 32
10.3. Noise Pipes 33
10.4. Handshake indistinguishability 34

11. Advanced features **34**
11.1. Dummy keys 34
11.2. Channel binding 35
11.3. Rekey 35
11.4. Out-of-order transport messages 36
11.5. Half-duplex protocols 36

12. DH functions, cipher functions, and hash functions **37**
12.1. The `25519` DH functions 37
12.2. The `448` DH functions 37
12.3. The `ChaChaPoly` cipher functions 37
12.4. The `AESGCM` cipher functions 38
12.5. The `SHA256` hash function 38
12.6. The `SHA512` hash function 38
12.7. The `BLAKE2s` hash function 38
12.8. The `BLAKE2b` hash function 38

13. Application responsibilities **38**

14. Security considerations **40**

15. Rationales **41**
15.1. Ciphers and encryption 41
15.2. Hash functions and hashing 43
15.3. Other 44

16. IPR **45**

17. Acknowledgements **45**

18. References **46**

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**.

2.2. Overview of 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 **handshake hash** 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. The key `k` and nonce `n` are used to encrypt static public keys and handshake payloads. Encryption with `k` uses some **AEAD** cipher mode (in the sense of Rogaway [1]) and uses the current `h` value as **associated data** which is covered by the AEAD authentication. Encryption of static public keys and payloads provides some confidentiality and key confirmation during the handshake phase.

A handshake message consists of some DH public keys followed by a **payload**. The payload may contain certificates or other data chosen by the application. To send a handshake message, the sender specifies the payload and 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`.
- `"ee", "se", "es", "ss"`: A DH is performed between the initiator's key pair (whether static or ephemeral is determined by the first letter) and the responder's key pair (whether static or ephemeral is determined by the second letter). 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 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, ee
```

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 a shared secret key.

Note that a cleartext payload is sent in the first message, after the cleartext ephemeral public key, and an encrypted payload is sent in the response message, after the cleartext ephemeral public key. 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, ee, s, es
```

In this case, the final `ck` and `k` values are a hash of both DH results. Since the `es` 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 authentication data (such as an authentication tag or "synthetic IV"), 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, ee, s, es
-> s, se
```

The following sections flesh out the details, and add some complications. 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 encapsulate payloads that require parsing of some sort, but payloads are handled by the application, not by 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 16 bytes of authentication data. The details depend on the AEAD cipher function, e.g. AES256-GCM, or ChaCha20-Poly1305, but typically the authentication data is either a 16-byte authentication tag appended to the ciphertext, or a 16-byte synthetic IV prepended to 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 are sent in a handshake prior to a DH operation, and will be AEAD ciphertexts if they occur after a DH operation. (If Noise is being used with pre-shared symmetric keys, this rule is different; see Section 9). 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, ee, s, es
-> s, se
```

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 encrypted payload)
3. 88 bytes (one encrypted public key and encrypted payload)

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 12.

The following notation will be used in algorithm pseudocode:

- The `||` operator concatenates byte sequences.

- The `byte()` function constructs a single byte.

4.1. DH functions

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

- `GENERATE_KEYPAIR()`: Generates a new Diffie-Hellman key pair. A DH key pair consists of `public_key` and `private_key` elements. A `public_key` represents an encoding of a DH public key into a byte sequence of length `DHLEN`. The `public_key` encoding details are specific to each set of DH functions.
- `DH(key_pair, public_key)`: Performs a Diffie-Hellman calculation between the private key in `key_pair` and the `public_key` and returns an output sequence of bytes of length `DHLEN`. For security, the Gap-DH problem based on this function must be unsolvable by any practical cryptanalytic adversary [2].

  The `public_key` either encodes some value in a large prime-order group (which may have multiple equivalent encodings), or is an invalid value. Implementations must handle invalid public keys either by returning some output which is purely a function of the public key and does not depend on the private key, or by signaling an error to the caller. The DH function may define more specific rules for handling invalid values.
- `DHLEN` = A constant specifying the size in bytes of public keys and DH outputs. For security reasons, `DHLEN` must be 32 or greater.

4.2. Cipher functions

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` (using the terminology from [1]) and returns a ciphertext that is the same size as the plaintext plus 16 bytes for authentication data. The entire ciphertext must be indistinguishable from random if the key is secret.
- `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.
- `REKEY(k)`: Returns a new 32-byte cipher key as a pseudorandom function of `k`. If this function is not specifically defined for some set of cipher

functions, then it defaults to returning the first 32 bytes from `ENCRYPT(k, maxnonce, zerolen, zeros)`, where `maxnonce` equals 2^{64}-1, `zerolen` is a zero-length byte sequence, and `zeros` is a sequence of 32 bytes filled with zeros.

4.3. Hash functions

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

- `HASH(data)`: Hashes some arbitrary-length data with a collision-resistant cryptographic 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 [3]).

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

- `HMAC-HASH(key, data)`: Applies `HMAC` from [3] using the `HASH()` function. This function is only called as part of `HKDF()`, below.
- `HKDF(chaining_key, input_key_material, num_outputs)`: Takes a `chaining_key` byte sequence of length `HASHLEN`, and an `input_key_material` byte sequence with length either zero bytes, 32 bytes, or `DHLEN` bytes. Returns a pair or triple of byte sequences each of length `HASHLEN`, depending on whether `num_outputs` is two or three:
  - 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))`.
  - If `num_outputs == 2` then returns the pair `(output1, output2)`.
  - Sets `output3 = HMAC-HASH(temp_key, output2 || byte(0x03))`.
  - Returns the triple `(output1, output2, output3)`.

Note that `temp_key`, `output1`, `output2`, and `output3` are all `HASHLEN` bytes in length. Also note that the `HKDF()` function is simply `HKDF` from [4] with the `chaining_key` as HKDF `salt`, and zero-length HKDF `info`.

5. Processing rules

To precisely define the processing rules we adopt an object-oriented terminology, and present three "objects" which encapsulate state variables and contain functions 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 a variable 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 any error is signaled by the `DECRYPT()` or `DH()` functions then the handshake has failed and the `HandshakeState` is deleted.

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` should be deleted except for the hash value `h`, which may be used for post-handshake channel binding (see Section 11.2).

Transport messages are then encrypted and decrypted by calling `EncryptWithAd()` and `DecryptWithAd()` on the relevant `CipherState` with zero-length associated data. If `DecryptWithAd()` signals an error due to `DECRYPT()` failure, then the input message is discarded. The application may choose to delete the `CipherState` and terminate the session on such an error, or may continue to attempt communications. If `EncryptWithAd()` or `DecryptWithAd()` signal an error due to nonce exhaustion, then the application must delete the `CipherState` and terminate the session.

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 functions. The `++` post-increment operator applied to `n` means "use the current `n` value, then increment it". The maximum `n` value (2^{64}-1) is reserved for other use. If incrementing `n` results in 2^{64}-1, then any further `EncryptWithAd()` or `DecryptWithAd()` 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.
- **`SetNonce(nonce)`**: Sets `n = nonce`. This function is used for handling out-of-order transport messages, as described in Section 11.4.
- **`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()` then `n` is not incremented and an error is signaled to the caller.
- **`Rekey()`**: Sets `k = REKEY(k)`.

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 functions:

- **`InitializeSymmetric(protocol_name)`**: Takes an arbitrary-length `protocol_name` byte sequence (see Section 8). 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)`**: Executes the following steps:
  - Sets `ck, temp_k = HKDF(ck, input_key_material, 2)`.
  - If `HASHLEN` is 64, then truncates `temp_k` to 32 bytes.
  - Calls `InitializeKey(temp_k)`.
- **`MixHash(data)`**: Sets `h = HASH(h || data)`.

- `MixKeyAndHash(input_key_material)`: This function is used for handling pre-shared symmetric keys, as described in Section 9. Executes the following steps:
  - Sets `ck, temp_h, temp_k = HKDF(ck, input_key_material, 3)`.
  - Calls `MixHash(temp_h)`.
  - If `HASHLEN` is 64, then truncates `temp_k` to 32 bytes.
  - Calls `InitializeKey(temp_k)`.
- `GetHandshakeHash()`: Returns `h`. This function should only be called at the end of a handshake, i.e. after the `Split()` function has been called. This function is used for channel binding, as described in Section 11.2
- `EncryptAndHash(plaintext)`: Sets `ciphertext = EncryptWithAd(h, plaintext)`, calls `MixHash(ciphertext)`, and returns `ciphertext`. Note that if `k` is `empty`, the `EncryptWithAd()` call will set `ciphertext` equal to `plaintext`.
- `DecryptAndHash(ciphertext)`: Sets `plaintext = DecryptWithAd(h, ciphertext)`, calls `MixHash(ciphertext)`, and returns `plaintext`. Note that if `k` is `empty`, the `DecryptWithAd()` call will set `plaintext` equal to `ciphertext`.
- `Split()`: Returns a pair of `CipherState` objects for encrypting transport messages. Executes the following steps, where `zerolen` is a zero-length byte sequence:
  - Sets `temp_k1, temp_k2 = HKDF(ck, zerolen, 2)`.
  - If `HASHLEN` is 64, then truncates `temp_k1` and `temp_k2` to 32 bytes.
  - 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 variables to track its role, and the remaining portion of the handshake pattern:

- `initiator`: A boolean indicating the initiator or responder role.

- `message_patterns`: A sequence of message patterns. Each message pattern is a sequence of tokens from the set `("e", "s", "ee", "es", "se", "ss")`. (An additional `"psk"` token is introduced in Section 9, but we defer its explanation until then.)

A `HandshakeState` responds to the following functions:

- `Initialize(handshake_pattern, initiator, prologue, s, e, rs, re)`: Takes a valid `handshake_pattern` (see Section 7) 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 key pairs `(s, e)` and public keys `(rs, re)` for initializing local variables, any of which may be empty. Public keys are only passed in if the `handshake_pattern` uses pre-messages (see Section 7). The ephemeral values `(e, re)` are typically left empty, since they are created and exchanged during the handshake; but there are exceptions (see Section 10.1).

  Performs the following steps:

  - Derives a `protocol_name` byte sequence by combining the names for the handshake pattern and crypto functions, as specified in Section 8. Calls `InitializeSymmetric(protocol_name)`.
  - Calls `MixHash(prologue)`.
  - Sets the `initiator`, `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 7 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`.

- `WriteMessage(payload, message_buffer)`: Takes a `payload` byte sequence which may be zero-length, and a `message_buffer` to write the output into. Performs the following steps, aborting if any `EncryptAndHash()` call returns an error:

  - Fetches and deletes the next message pattern from `message_patterns`, then sequentially processes each token from the message pattern:
    * For `"e"`: Sets `e` (which must be empty) to `GENERATE_KEYPAIR()`. Appends `e.public_key` to the buffer. Calls `MixHash(e.public_key)`.
    * For `"s"`: Appends `EncryptAndHash(s.public_key)` to the buffer.
    * For `"ee"`: Calls `MixKey(DH(e, re))`.
    * For `"es"`: Calls `MixKey(DH(e, rs))` if initiator, `MixKey(DH(s, re))` if responder.
    * For `"se"`: Calls `MixKey(DH(s, re))` if initiator, `MixKey(DH(e, rs))` if responder.
    * For `"ss"`: Calls `MixKey(DH(s, rs))`.
  - 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. Performs the following steps, aborting if any `DecryptAndHash()` call returns an error:
  - Fetches and deletes the next message pattern from `message_patterns`, then sequentially processes each token from the message pattern:
    * For `"e"`: Sets `re` (which must be empty) 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` (which must be empty) to `DecryptAndHash(temp)`.
    * For `"ee"`: Calls `MixKey(DH(e, re))`.
    * For `"es"`: Calls `MixKey(DH(e, rs))` if initiator, `MixKey(DH(s, re))` if responder.
    * For `"se"`: Calls `MixKey(DH(s, re))` if initiator, `MixKey(DH(e, rs))` if responder.
    * For `"ss"`: Calls `MixKey(DH(s, rs))`.
  - 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 protocols 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 Section 9).

7. Handshake patterns

A **message pattern** is some sequence of tokens from the set `("e", "s", "ee", "es", "se", "ss", "psk")`. The handling of these tokens within `WriteMessage()` and `ReadMessage()` has been described previously, except for the `"psk"` token, which will be described in Section 9. Future specifications might introduce other tokens.

A **pre-message pattern** is one of the following sequences of tokens:

- `"e"`
- `"s"`
- `"e, s"`
- empty

A **handshake pattern** consists of:

- A pre-message pattern for the initiator, representing information about the initiator's public keys that is known to the responder.
- A pre-message pattern for the responder, representing information about the responder's public keys that is known to the initiator.
- 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 (with one exception - see next paragraph). The next message is sent by the responder, the next from the initiator, and so on in alternating fashion.

(Exceptional case: Noise allows special **fallback patterns** where the responder switches to a different pattern than the initator started with (see Section 10.1). If the initiator's pre-message contains an `"e"` token, then this handshake pattern is a fallback pattern. In the case of a fallback pattern the first handshake message is sent by the *responder*, the next by the *initiator*, and so on.)

The following handshake pattern describes an unauthenticated DH handshake:

```
NN():
  -> e
  <- e, ee
```

The handshake pattern name is `NN`. This naming convention will be explained in Section 7.3. The empty parentheses indicate that neither party is initialized with any key pairs. The tokens `"s"`, `"e"`, or `"e, s"` inside the parentheses would indicate that the initiator is initialized with static and/or ephemeral key pairs. The tokens `"rs"`, `"re"`, or `"re, rs"` would indicate the same thing for the responder.

Right-pointing arrows show messages sent by the initiator. Left-pointing arrows show messages sent by the responder.

Non-empty 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 ("zero-RTT encryption"), although full forward secrecy and replay protection is only achieved with the second message.

```
NK(rs):
  <- s
  ...
  -> e, es
  <- e, ee
```

7.1. Pattern validity

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

1. 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.
2. Parties must not send their static public key or ephemeral public key more than once per handshake (i.e. including the pre-messages, there must be no more than one occurrence of `"e"`, and one occurrence of `"s"`, in the messages sent by any party).
3. After performing a DH between a remote public key and any local private key that is not the ephemeral private key, the local party must not call `ENCRYPT()` unless it has also performed a DH between the ephemeral private key and the remote public key.

Patterns failing the first check are obviously nonsense.

The second check outlaws redundant transmission of values to simplify implementation and testing.

The third check is necessary because Noise uses DH outputs involving ephemeral keys to randomize the shared secret keys, and to provide forward secrecy. Patterns failing this check 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.

7.2. One-way patterns

The following 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 7.1).

One-way patterns are named with a single character, which indicates the status of the sender's static key:

- `N` = `N`o static key for sender
- `K` = Static key for sender `K`nown to recipient
- `X` = Static key for sender `X`mitted ("transmitted") to recipient

---

```
N(rs):
  <- s
  ...
  -> e, es


K(s, rs):
  -> s
  <- s
  ...
  -> e, es, ss


X(s, rs):
  <- s
  ...
  -> e, es, s, ss
```

`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 (`K`) or transmitted under encryption (`X`).

7.3. Interactive patterns

The following handshake patterns represent interactive protocols.

Interactive patterns are named with two characters, which indicate the status of the initator and responder's static keys:

The first character refers to the initiator's static key:

- `N` = `N`o static key for initiator
- `K` = Static key for initiator `K`nown to responder
- `X` = Static key for initiator `X`mitted ("transmitted") to responder
- `I` = Static key for initiator `I`mmediately transmitted to responder, despite reduced or absent identity hiding

The second character refers to the responder's static key:

- `N` = `N`o static key for responder
- `K` = Static key for responder `K`nown to initiator
- `X` = Static key for responder `X`mitted ("transmitted") to initiator

```
NN():
  -> e
  <- e, ee
```

```
KN(s):
  -> s
  ...
  -> e
  <- e, ee, se
```

```
NK(rs):
  <- s
  ...
  -> e, es
  <- e, ee
```

```
KK(s, rs):
  -> s
  <- s
  ...
  -> e, es, ss
  <- e, ee, se
```

```
NX(rs):
  -> e
  <- e, ee, s, es
```

```
KX(s, rs):
  -> s
  ...
  -> e
  <- e, ee, se, s, es
```

```
XN(s):
  -> e
  <- e, ee
  -> s, se
```

```
IN(s):
  -> e, s
  <- e, ee, se
```

```
XK(s, rs):
  <- s
  ...
  -> e, es
  <- e, ee
  -> s, se
```

```
IK(s, rs):
  <- s
  ...
  -> e, es, s, ss
  <- e, ee, se
```

```
XX(s, rs):
  -> e
  <- e, ee, s, es
  -> s, se
```

```
IX(s, rs):
  -> e, s
  <- e, ee, se, s, es
```

The `XX` pattern is the most generically useful, since it supports mutual authentication and transmission of static public keys.

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.

7.4. Payload security properties

The following table lists the security properties for Noise handshake and transport payloads for all the named patterns in Section 7.2 and Section 7.3. 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 (`"ss"`) 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 (`"es"` or `"se"`) 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 (`"ee"`). 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 as 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
N	0	2
K	1	2
X	1	2
NN		
-> e	0	0
<- e, ee	0	1
->	0	1
NK		
<- s		
...		
-> e, es	0	2
<- e, ee	2	1
->	0	5
NX		
-> e	0	0
<- e, ee, s, es	2	1
->	0	5

```
XN
  -> e                    0               0
  <- e, ee                0               1
  -> s, se                2               1
  <-                      0               5


XK
  <- s
  ...
  -> e, es                0               2
  <- e, ee                2               1
  -> s, se                2               5
  <-                      2               5


XX
 -> e                     0               0
 <- e, ee, s, es          2               1
 -> s, se                 2               5
 <-                       2               5


KN
  -> s
  ...
  -> e                    0               0
  <- e, ee, se            0               3
  ->                      2               1
  <-                      0               5


KK
  -> s
  <- s
  ...
  -> e, es, ss            1               2
  <- e, ee, se            2               4
  ->                      2               5
  <-                      2               5
```

```
KX
  -> s
  ...
  -> e                      0               0
  <- e, ee, se, s, es       2               3
  ->                        2               5
  <-                        2               5


IN
  -> e, s                   0               0
  <- e, ee, se              0               3
  ->                        2               1
  <-                        0               5


IK
  <- s
  ...
  -> e, es, s, ss           1               2
  <- e, ee, se              2               4
  ->                        2               5
  <-                        2               5


IX
  -> e, s                   0               0
  <- e, ee, se, s, es       2               3
  ->                        2               5
  <-                        2               5
```

7.5. Identity hiding

The following table lists the identity hiding properties for all the named patterns in Section 7.2 and Section 7.3. 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 pair of (responder's private key, initiator's public key) and learn whether the candidate pair is 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.

	Initiator	Responder
N	-	3
K	5	5
X	4	3
NN	-	-
NK	-	3

NX	-	1
XN	2	-
XK	8	3
XX	8	1
KN	7	-
KK	5	5
KX	7	6
IN	0	-
IK	4	3
IX	0	6

8. Protocol names and modifiers

To produce a **Noise protocol name** for `Initialize()` you concatenate the ASCII string `"Noise_"` with four underscore-separated name sections which sequentially name the handshake pattern, the DH functions, the cipher functions, and then the hash functions. The resulting name must be 255 bytes or less. Examples:

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

Each name section must consist only of alphanumeric characters (i.e. characters in one of the ranges `"A"`...`"Z"`, `"a"`...`"z"`, and `"0"`...`"9"`), and the two special characters `"+"` and `"/"`.

Additional rules apply to each name section, as specified below.

8.1. Handshake pattern name section

A handshake pattern name section contains a handshake pattern name plus a sequence of zero or more **pattern modifiers**.

The handshake pattern name must be an uppercase ASCII string containing only alphabetic characters (e.g. `"XX"` or `"IK"`).

Pattern modifiers specify arbitrary extensions or modifications to the behavior specified by the handshake pattern. For example, a modifier could be applied to a handshake pattern which transforms it into a different pattern according to some rule.

As examples of such a modifier, the `"psk0"` and `"fallback"` modifiers described later in this document modify the base pattern to either incorporate a pre-shared symmetric key, or to be usable as a fallback protocol.

A pattern modifier is named with a lowercase alphanumeric ASCII string which is appended to the base pattern as described below:

The first modifier added onto a base pattern is simply appended. Thus the `"fallback"` modifier, when added to the `"XX"` pattern, produces `"XXfallback"`. Additional modifiers are separated with a plus sign. Thus, adding the `"psk0"` modifier would result in the name section `"XXfallback+psk0"`, or a full protocol name such as `"Noise_XXfallback+psk0_25519_AESGCM_SHA256"`.

In some cases the sequential ordering of modifiers will specify different protocols. However, if the order of some modifiers does not matter, then they are required to be sorted alphabetically (this is an arbitrary convention to ensure interoperability).

8.2. Cryptographic algorithm name sections

The rules for the DH, cipher, and hash name sections are identical. Each name section must contain one or more algorithm names separated by plus signs.

Each algorithm name must consist solely of alphanumeric characters and the forward-slash character (`"/"`). Algorithm names are recommended to be short, and to use the `"/"` character only when necessary to avoid ambiguity (e.g. `"SHA3/256"` is preferable to `"SHA3256"`).

In most cases there will be a single algorithm name in each name section (i.e. no plus signs). Multiple algorithms are only used when called for by the pattern or a modifier.

None of the patterns or modifiers in this document require multiple algorithms in any name section. However, this functionality might be useful in future extensions, e.g. using multiple algorithms in the DH section to provide "hybrid" post-quantum forward secrecy, or using different hash algorithms for different purposes.

9. Pre-shared symmetric keys

Noise provides a **pre-shared symmetric key** or **PSK** mode to support protocols where both parties have a 32-byte shared secret key.

9.1. Cryptographic functions

PSK mode uses the `SymmetricState.MixKeyAndHash()` function to mix the PSK into both the encryption keys and the `h` value.

Note that `MixKeyAndHash()` uses `HKDF(..., 3)`. The third output from `HKDF()` is used as the `k` value so that calculation of `k` may be skipped if `k` is not used.

9.2. Handshake tokens

In a PSK handshake, a `"psk"` token is allowed to appear one or more times in a handshake pattern. This token can only appear in message patterns (not pre-message patterns). This token is processed by calling `MixKeyAndHash(psk)`, where `psk` is a 32-byte secret value provided by the application.

In non-PSK handshakes, the `"e"` token in a pre-message pattern or message pattern always results in a call to `MixHash(e.public_key)`. In a PSK handshake, all of these calls are followed by `MixKey(e.public_key)`. In conjunction with the validity rule in the next section, this ensures that PSK-based encryption uses encryption keys that are randomized using ephemeral public keys as nonces.

9.3. Validity rule

To prevent catastrophic key reuse, handshake patterns using the "psk" token must follow an additional validity rule:

- A party may not send any encrypted data after it processes a "psk" token unless it has previously sent an ephemeral public key (an "e" token), either before or after the "psk" token.

This rule guarantees that a k derived from a PSK will never be used for encryption unless it has also been randomized by MixKey(e.public_key) using a self-chosen ephemeral public key.

9.4. Pattern modifiers

To indicate PSK mode and the placement of the "psk" token, pattern modifiers are used (see Section 8). The modifier psk0 places a "psk" token at the beginning of the first handshake message. The modifiers psk1, psk2, etc., place a "psk" token at the end of the first, second, etc., handshake message.

Any pattern using one of these modifiers must process tokens according to the rules in Section 9.2, and must follow the validity rule in Section 9.3.

The table below lists some unmodified one-way patterns on the left, and the recommended PSK pattern on the right:

N(rs):
  <- s
  ...
  -> e, es
	Npsk0(rs):
	  <- s
	  ...
	  -> psk, e, es

K(s, rs):
  -> s
  <- s
  ...
  -> e, es, ss
	Kpsk0(s, rs):
	  -> s
	  <- s
	  ...
	  -> psk, e, es, ss

X(s, rs):
  <- s
  ...
  -> e, es, s, ss
	Xpsk1(s, rs):
	  <- s
	  ...
	  -> e, es, s, ss, psk

Note that the `psk1` modifier is recommended for `X`. This is because `X` transmits the initiator's static public key. Because PSKs are typically pairwise, the responder likely cannot determine the PSK until it has decrypted the initiator's static public key. Thus, `psk1` is likely to be more useful here than `psk0`.

Following similar logic, we can define the most likely interactive PSK patterns:

```
NN():
  -> e
  <- e, ee
```

```
NNpsk0():
  -> psk, e
  <- e, ee
```

```
NN():
  -> e
  <- e, ee
```

```
NNpsk2():
  -> e
  <- e, ee, psk
```

```
NK(rs):
  <- s
  ...
  -> e, es
  <- e, ee
```

```
NKpsk0(rs):
  <- s
  ...
  -> psk, e, es
  <- e, ee
```

```
NK(rs):
  <- s
  ...
  -> e, es
  <- e, ee
```

```
NKpsk2(rs):
  <- s
  ...
  -> e, es
  <- e, ee, psk
```

```
NX(rs):
  -> e
  <- e, ee, s, es
```

```
NXpsk2(rs):
  -> e
  <- e, ee, s, es, psk
```

```
XN(s):
  -> e
  <- e, ee
  -> s, se
```

```
XNpsk3(s):
  -> e
  <- e, ee
  -> s, se, psk
```

```
XK(s, rs):
  <- s
  ...
  -> e, es
  <- e, ee
  -> s, se
```

```
XXpsk3(s, rs):
  -> e
  <- e, ee, s, es
  -> s, se, psk
```

```
KNpsk0(s):
  -> s
  ...
  -> psk, e
  <- e, ee, se
```

```
KNpsk2(s):
  -> s
  ...
  -> e
  <- e, ee, se, psk
```

```
KKpsk0(s, rs):
  -> s
  <- s
  ...
  -> psk, e, es, ss
  <- e, ee, se
```

```
KKpsk2(s, rs):
  -> s
  <- s
  ...
  -> e, es, ss
  <- e, ee, se, psk
```

```
KX(s, rs):
  -> s
  ...
  -> e
  <- e, ee, se, s, es
```

```
KXpsk2(s, rs):
  -> s
  ...
  -> e
  <- e, ee, se, s, es, psk
```

```
IN(s):
  -> e, s
  <- e, ee, se
```

```
INpsk1(s):
  -> e, s, psk
  <- e, ee, se
```

```
IN(s):
  -> e, s
  <- e, ee, se
```

```
INpsk2(s):
  -> e, s
  <- e, ee, se, psk
```

```
IK(s, rs):
  <- s
  ...
  -> e, es, s, ss
  <- e, ee, se
```

```
IKpsk1(s, rs):
  <- s
  ...
  -> e, es, s, ss, psk
  <- e, ee, se
```

```
IK(s, rs):
  <- s
  ...
  -> e, es, s, ss
  <- e, ee, se
```

```
IKpsk2(s, rs):
  <- s
  ...
  -> e, es, s, ss
  <- e, ee, se, psk
```

```
IX(s, rs):
  -> e, s
  <- e, ee, se, s, es
```

```
IXpsk2(s, rs):
  -> e, s
  <- e, ee, se, s, es, psk
```

The above list does not exhaust all possible patterns that can be formed with these modifiers. In particular, any of these PSK modifiers can be safely applied to any previously named pattern, resulting in patterns like `IKpsk0`, `KKpsk1`, or even `XXpsk0+psk3`, which aren't listed above.

This still doesn't exhaust all the ways that `"psk"` tokens could be used outside of these modifiers (e.g. placement of `"psk"` tokens in the middle of a message pattern). Defining additional PSK modifiers is outside the scope of this document.

10. Fallback protocols

10.1. Fallback patterns

So far we've discussed Noise protocols which execute a single handshake chosen by the initiator. These include "zero-RTT" protocols where the initiator encrypts the initial message based on some stored information about the responder (such as the responder's static public key).

If the initiator's information is out-of-date the responder won't be able to decrypt the message. To handle this, the responder might choose to execute a different Noise handshake, known as a **fallback handshake**.

To support this case Noise allows **fallback patterns**. Fallback patterns differ from other handshake patterns in two ways:

- The initiator and responder roles from the pre-fallback handshake are preserved in the fallback handshake. Thus, the *responder* sends the first message in a fallback handshake. In other words, the first handshake message in a fallback pattern is shown with a left-pointing arrow (from the responder) instead of a right-pointing arrow (from the initiator).
- Any public keys sent in the clear in the initiator's first message are included in the initiator's pre-message in the fallback pattern. The initiator's pre-message must always include an ephemeral public key. An ephemeral public key is not otherwise included in the initiator's pre-message (initiators typically transmit an ephemeral public key in their first message). Thus, the presence of an ephemeral public key in the initiator's pre-message indicates a fallback pattern.

Another caveat for fallback handshakes: If the initial handshake message has a prologue or payload that the responder makes any decisions based on, then the `h` value after processing that handshake message should be included in the prologue for the fallback handshake.

10.2. Indicating fallback

A typical fallback scenario for zero-RTT encryption involves three different Noise handshakes:

- A **full handshake** is used if the initiator doesn't possess stored information about the responder that would enable zero-RTT encryption, or doesn't wish to use the zero-RTT handshake.
- A **zero-RTT handshake** allows encryption of data in the initial message.
- A **fallback handshake** is triggered by the responder if it can't decrypt the initiator's first zero-RTT handshake message.

There must be some way for the responder to distinguish full versus zero-RTT handshakes on receiving the first message. If the initiator makes a zero-RTT attempt, there must be some way for the initiator to distinguish zero-RTT from fallback handshakes on receiving the response.

For example, each handshake message could be preceded by a `type` byte (see Section 13). This byte is not part of the Noise message proper, but signals which handshake is being used:

- If `type == 0` in the initiator's first message then the initiator is performing a full handshake.
- If `type == 1` in the initiator's first message then the initiator is performing a zero-RTT handshake.
- If `type == 0` in the response then the zero-RTT message was accepted.
- If `type == 1` in the response then the responder failed to decrypt the initiator's zero-RTT message and is performing a fallback handshake.

Note that the `type` byte doesn't need to be explicitly authenticated (either as prologue, or as "associated data" in the AEAD encryption), since it's implicitly authenticated if the message is processed succesfully.

10.3. Noise Pipes

This section defines the **Noise Pipe** protocol. This protocol uses three handshake patterns - two defined previously, and a new one. These handshake patterns satisfy the full, zero-RTT, and fallback roles discussed in the previous section, so can be used to provide a full handshake with a simple zero-RTT option:

```
XX(s, rs):
  -> e
  <- e, ee, s, es
  -> s, se

IK(s, rs):
  <- s
  ...
  -> e, es, s, ss
  <- e, ee, se

XXfallback(e, s, rs):
  -> e
  ...
  <- e, ee, s, es
  -> s, se
```

The XX pattern is used for a **full handshake** if the parties haven't communicated before, after which the initiator can cache the responder's static public key.

The IK pattern is used for a **zero-RTT handshake**.

The XXfallback pattern is used if the responder fails to decrypt the first IK message (perhaps due to having changed their static key). In this case the responder will switch to a fallback handshake using XXfallback, which is identical to XX except the ephemeral public key from the first IK message is used as the initiator's pre-message.

10.4. Handshake indistinguishability

Parties might wish to hide from an eavesdropper which type of handshake they are performing. For example, suppose parties are using Noise Pipes, and want to hide whether they are performing a full handshake, zero-RTT handshake, or fallback handshake.

This is fairly easy:

- The first three messages can have their payloads padded with random bytes to a constant size, regardless of which handshake is executed.
- The responder will attempt to decrypt the first message as a NoiseIK message, and will fallback to XXfallback if decryption fails.
- An initiator who sends a IK initial message can use trial decryption to differentiate between a response using IK or XXfallback.
- An initiator attempting a full handshake will send an ephemeral public key, then random padding, and will use XXfallback to handle the response. Note that XX isn't used, because the server can't distinguish a XX message from a failed IK attempt by using trial decryption.

This leaves the Noise ephemeral public keys in the clear. Ephemeral public keys are randomly chosen DH public values, but they will typically have enough structure that an eavesdropper might suspect the parties are using Noise, even if the eavesdropper can't distinguish the different handshakes. To make the ephemerals indistinguishable from random byte sequences, techniques like Elligator [5] could be used.

11. Advanced features

11.1. Dummy keys

Consider a protocol where an initiator will authenticate herself if the responder requests it. This could be viewed as the initiator choosing between patterns like

NX and 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 XX. The initiator can simulate the NX case by sending a **dummy static public key** if authentication is not requested. The value of the dummy public key doesn't matter.

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 or computation time. It could be extended to allow a XX pattern to support any permutation of authentications (initiator only, responder only, both, or none).

Similarly, **dummy PSKs** (e.g. a PSK of all zeros) would allow a protocol to optionally support PSKs.

11.2. Channel binding

Parties might wish to execute a Noise protocol, then perform authentication at the application layer using signatures, passwords, or something else.

To support this, Noise libraries may call GetHandshakeHash() after the handshake is complete and expose the returned value to the application as a **handshake hash** which uniquely identifies the Noise session.

Parties can then sign the handshake hash, or hash it along with their password, to get an authentication token which has a "channel binding" property: the token can't be used by the receiving party with a different sesssion.

11.3. Rekey

Parties might wish to periodically update their cipherstate keys using a one-way function, so that a compromise of cipherstate keys will not decrypt older messages. Periodic rekey might also be used to reduce the volume of data encrypted under a single cipher key (this is usually not important with good ciphers, though note the discussion on AESGCM data volumes in Section 14).

To enable this, Noise supports a Rekey() function which may be called on a CipherState.

It is up to to the application if and when to perform rekey. For example:

- Applications might perform continuous rekey, where they rekey the relevant cipherstate after every transport message sent or received. This is simple and gives good protection to older ciphertexts, but might be difficult for implementations where changing keys is expensive.
- Applications might rekey a cipherstate automatically after it has has been used to send or receive some number of messages.

- Applications might choose to rekey based on arbitrary criteria, in which case they signal this to the other party by sending a message.

Applications must make these decisions on their own; there are no modifiers which specify rekey behavior.

Note that rekey only updates the cipherstate's `k` value, it doesn't reset the cipherstate's `n` value, so applications performing rekey must still perform a new handshake if sending 2^{64} or more transport messages.

11.4. Out-of-order transport messages

In some use cases, Noise transport messages might be lost or arrive out-of-order (e.g. when messages are sent over UDP). To handle this, an application protocol can send the `n` value used for encrypting each transport message alongside that message. On receiving such a message the recipient would call the `SetNonce()` function on the receiving `CipherState` using the received `n` value.

Recipients doing this must track the received `n` values for which decryption was successful and reject any message which repeats such a value, to prevent replay attacks.

Note that lossy and out-of-order message delivery introduces many other concerns (including out-of-order handshake messages and denial of service risks) which are outside the scope of this document.

11.5. Half-duplex protocols

In some application protocols the parties strictly alternate sending messages. In this case Noise can be used in a "half-duplex" mode [6] where the first `CipherState` returned by `Split()` is used for encrypting messages in both directions, and the second `CipherState` returned by `Split()` is unused. This allows some small optimizations, since `Split()` only has to calculate a single output `CipherState`, and both parties only need to store a single `CipherState` during the transport phase.

This feature must be used with extreme caution. In particular, it would be a catastrophic security failure if the protocol is not strictly alternating and both parties encrypt different messages using the same `CipherState` and nonce value.

12. DH functions, cipher functions, and hash functions

12.1. The 25519 DH functions

- **GENERATE_KEYPAIR()**: Returns a new Curve25519 key pair.
- **DH(keypair, public_key)**: Executes the Curve25519 DH function (aka "X25519" in [7]). Invalid public key values will produce an output of all zeros.

  Alternatively, implementations are allowed to detect inputs that produce an all-zeros output and signal an error instead. This behavior is discouraged because it adds complexity and implementation variance, and does not improve security. This behavior is allowed because it might match the behavior of some software.
- **DHLEN** = 32

12.2. The 448 DH functions

- **GENERATE_KEYPAIR()**: Returns a new Curve448 key pair.
- **DH(keypair, public_key)**: Executes the Curve448 DH function (aka "X448" in [7]). Invalid public key values will produce an output of all zeros.

  Alternatively, implementations are allowed to detect inputs that produce an all-zeros output and signal an error instead. This behavior is discouraged because it adds complexity and implementation variance, and does not improve security. This behavior is allowed because it might match the behavior of some software.
- **DHLEN** = 56

12.3. The ChaChaPoly cipher functions

- **ENCRYPT(k, n, ad, plaintext) / DECRYPT(k, n, ad, ciphertext)**: AEAD_CHACHA20_POLY1305 from [8]. 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; with these implementations it's compatible to encode **n** directly into the ChaCha20 nonce without the 32-bit zero prefix).

12.4. The AESGCM cipher functions

- **ENCRYPT(k, n, ad, plaintext) / DECRYPT(k, n, ad, ciphertext)**: AES256 with GCM from [9] with a 128-bit tag appended to the ciphertext. The 96-bit nonce is formed by encoding 32 bits of zeros followed by big-endian encoding of **n**.

12.5. The SHA256 hash function

- **HASH(input)**: SHA-256 from [10].
- **HASHLEN** = 32
- **BLOCKLEN** = 64

12.6. The SHA512 hash function

- **HASH(input)**: SHA-512 from [10].
- **HASHLEN** = 64
- **BLOCKLEN** = 128

12.7. The BLAKE2s hash function

- **HASH(input)**: BLAKE2s from [11] with digest length 32.
- **HASHLEN** = 32
- **BLOCKLEN** = 64

12.8. The BLAKE2b hash function

- **HASH(input)**: BLAKE2b from [11] with digest length 64.
- **HASHLEN** = 64
- **BLOCKLEN** = 128

13. Application responsibilities

An application built on Noise must consider several issues:

- **Choosing crypto functions**: The 25519 DH functions are recommended for typical uses, though the 448 DH functions might offer extra security in case a cryptanalytic attack is developed against elliptic curve cryptography. The 448 DH functions should be used with a 512-bit hash like SHA512 or BLAKE2b. The 25519 DH functions may be used with a 256-bit hash like SHA256 or BLAKE2s, though a 512-bit hash might offer extra security in

case a cryptanalytic attack is developed against the smaller hash functions. `AESGCM` is hard to implement with high speed and constant time in software.

- **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.
- **Session 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 an interactive session, or the end of a one-way 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 might wish to include a single-byte type field prior to each Noise handshake message (and prior to the length field, if one is included). A recommended idiom is for zero to indicate no change from the current protocol, and for applications to reject messages with an unknown value. This allows future protocol versions to specify fallback handshakes, different versions, or other different types of messages during a handshake.

14. Security considerations

This section collects various security considerations:

- **Authentication**: A Noise protocol with static public keys verifies that the corresponding private keys are possessed by the participant(s), but it's up to the application to determine whether the remote party's static public key is acceptable. Methods for doing so include certificates which sign the public key (and which may be passed in handshake payloads), preconfigured lists of public keys, or "pinning" / "key-continuity" approaches where parties remember public keys they encounter and check whether the same party presents the same public key in the future.
- **Session termination**: Preventing attackers from truncating a stream of transport messages is an application responsibility. See previous section.
- **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 fallback handshakes, and requires careful attention if a Noise handshake is preceded by communication between the parties.
- **Misusing public keys as secrets**: It might be tempting to use a pattern with a pre-message public key and assume that a successful handshake implies the other party's knowledge of the public key. Unfortunately, this is not the case, since setting public keys to invalid values might cause predictable DH output. For example, a `Noise_NK_25519` initiator might send an invalid ephemeral public key to cause a known DH output of all zeros, despite not knowing the responder's static public key. If the parties want to authenticate with a shared secret, it should be used as a PSK.
- **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 by setting public keys to invalid values that cause predictable DH output (as in the previous bullet). It might also be possible to set public keys to equivalent values that cause the same DH output for different inputs. This is why a higher-level protocol should use the handshake hash (`h`) for a unique channel binding, instead of `ck`, as explained in Section 11.2.
- **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, and the maximum nonce value is reserved. This means parties are not allowed to send more than 2^{64}-1 transport messages.
- **Fresh ephemerals**: Every party in a Noise protocol must send a fresh ephemeral public key prior to sending any encrypted data. Ephemeral keys must never be reused. Violating these rules is likely to cause catastrophic

key reuse. This is one rationale behind the patterns in Section 7, and the validity rules in Section 7.1. 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 must be secret values with 256 bits of entropy.
- **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 then different cipher functions should be chosen.
- **Hash collisions**: If an attacker can find hash collisions on prologue data or the handshake hash, they may be able to perform "transcript collision" attacks that trick the parties into having different views of handshake data. It is important to use Noise with collision-resistant hash functions, and replace the hash function at any sign of weakness.
- **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.

15. Rationales

This section collects various design rationales.

15.1. Ciphers and encryption

Cipher keys and PSKs are 256 bits because:

- 256 bits is a conservative length for cipher keys when considering cryptanalytic safety margins, time/memory tradeoffs, multi-key attacks, rekeying, and quantum attacks.
- Pre-shared key length is fixed to simplify testing and implementation, and to deter users from mistakenly using low-entropy passwords as pre-shared keys.

Nonces are 64 bits because:

- Some ciphers only have 64 bit nonces (e.g. Salsa20).
- 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 authentication data in a ciphertext (i.e. the authentication tag or synthetic IV) 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 guesses for authentication data are correct.
- A single fixed length is simpler than supporting variable-length tags.

Rekey defaults to using encryption with the nonce 2^{64}-1 because:

- With `AESGCM` and `ChaChaPoly` rekey can be computed efficiently (the "encryption" just needs to apply the cipher, and can skip calculation of the authentication tag).

Rekey doesn't reset `n` to zero because:

- Leaving `n` unchanged is simple.
- If the cipher has a weakness such that repeated rekeying gives rise to a cycle of keys, then letting `n` advance will avoid catastrophic reuse of the same `k` and `n` values.
- Letting `n` advance puts a bound on the total number of encryptions that can be performed with a set of derived keys.

The `AESGCM` data volume 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}).

Cipher nonces are big-endian for `AESGCM`, and little-endian for `ChaCha20`, because:

- ChaCha20 uses a little-endian block counter internally.
- AES-GCM uses a big-endian block counter internally.
- It makes sense to use consistent endianness in the cipher code.

15.2. Hash functions and hashing

The recommended hash function families are SHA2 and BLAKE2 because:

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

Hash output lengths of both 256 bits and 512 bits are supported because:

- 256-bit hashes provide sufficient collision resistance at the 128-bit security level.
- The 256-bit hashes (SHA-256 and BLAKE2s) require less RAM, and less computation when processing smaller inputs (due to smaller block size), than SHA-512 and BLAKE2b.
- SHA-256 and BLAKE2s are faster on 32-bit processors than the larger hashes, which use 64-bit operations internally.

The `MixKey()` design uses HKDF because:

- HKDF is well-known and HKDF "chains" are used in similar ways in other protocols (e.g. Signal, IPsec, TLS 1.3).
- HKDF has a published analysis [12].
- HKDF applies multiple layers of hashing between each `MixKey()` input. This "extra" hashing might mitigate the impact of hash function weakness.

HMAC is used with all hash functions instead of allowing hashes to use a more specialized function (e.g. keyed BLAKE2), because:

- HKDF requires the use of HMAC, and some of the HKDF analysis in [12] depends on the nested structure of HMAC.
- HMAC is widely used with Merkle-Damgard hashes such as SHA2. SHA3 candidates such as Keccak and BLAKE were required to be suitable with HMAC. Thus, HMAC should be applicable to all widely-used hash functions.
- HMAC applies nested hashing to process each input. This "extra" hashing might mitigate the impact of hash function weakness.
- HMAC (and HKDF) are widely-used constructions. If some weakness is found in a hash function, cryptanalysts will likely analyze that weakness in the context of HMAC and HKDF.
- Applying HMAC consistently is simple, and avoids having custom designs with different cryptanalytic properties when using different hash functions.
- HMAC is easy to build on top of a hash function interface. If a more specialized function (e.g. keyed BLAKE2) can't be implemented using only

the underlying hash, then it is not guaranteed to be available everywhere the hash function is available.

`MixHash()` is used instead of sending all inputs directly through `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.

The `h` value hashes handshake ciphertext instead of plaintext because:

- This ensures `h` is a non-secret value that can be used for channel-binding or other purposes without leaking secret information.
- This provides stronger guarantees against ciphertext malleability.

15.3. Other

Big-endian length fields are recommended because:

- 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.

Session termination is left to the application because:

- Providing a termination signal in Noise doesn't help the application much, since the application still has to use the signal correctly.
- For an application with its own termination signal, having a second termination signal in Noise is likely to be confusing rather than helpful.

Explicit random nonces (like TLS "Random" fields) are not used because:

- One-time ephemeral public keys make explicit nonces unnecessary.
- Explicit nonces allow reuse of ephemeral public keys. However reusing ephemerals (with periodic replacement) is more complicated, requires a secure time source, is less secure in case of ephemeral compromise, and only provides a small optimization, since key generation can be done for a fraction of the cost of a DH operation.
- Explicit nonces increase message size.

- Explicit nonces make it easier to "backdoor" crypto implementations, e.g. by modifying the RNG so that key recovery data is leaked through the nonce fields.

16. IPR

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

17. Acknowledgements

Noise is inspired by:

- The NaCl and CurveCP protocols from Dan Bernstein et al [13], [14].
- The SIGMA and HOMQV protocols from Hugo Krawczyk [15], [16].
- The Ntor protocol from Ian Goldberg et al [17].
- The analysis of OTR by Mario Di Raimondo et al [18].
- The analysis by Caroline Kudla and Kenny Paterson of "Protocol 4" by Simon Blake-Wilson et al [19], [20].
- Mike Hamburg's proposals for a sponge-based protocol framework, which led to STROBE [21], [22].
- The KDF chains used in the Double Ratchet Algorithm [23].

General feedback on the spec and design came from: Moxie Marlinspike, Jason Donenfeld, Rhys Weatherley, Mike Hamburg, David Wong, Jake McGinty, Tiffany Bennett, Jonathan Rudenberg, Stephen Touset, Tony Arcieri, Alex Wied, Alexey Ermishkin, and Olaoluwa Osuntokun.

Thanks to Tom Ritter, Karthikeyan Bhargavan, David Wong, Klaus Hartke, Dan Burkert, Jake McGinty, Yin Guanhao, and Nazar Mokrynskyi for editorial feedback.

Moxie Marlinspike, Hugo Krawczyk, Samuel Neves, Christian Winnerlein, J.P. Aumasson, and Jason Donenfeld provided helpful input and feedback on the key derivation design.

The PSK approach was largely motivated and designed by Jason Donenfeld, based on his experience with PSKs in WireGuard.

The rekey design benefited from discussions with Rhys Weatherley, Alexey Ermishkin, and Olaoluwa Osuntokun.

The BLAKE2 team (in particular J.P. Aumasson, Samuel Neves, and Zooko) provided helpful discussion on using BLAKE2 with Noise.

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

18. References

[1] P. Rogaway, "Authenticated-encryption with Associated-data," in Proceedings of the 9th ACM Conference on Computer and Communications Security, 2002. http://web.cs.ucdavis.edu/~rogaway/papers/ad.pdf

[2] Okamoto, Tatsuaki and Pointcheval, David, "The Gap-Problems: A New Class of Problems for the Security of Cryptographic Schemes," in Proceedings of the 4th International Workshop on Practice and Theory in Public Key Cryptography: Public Key Cryptography, 2001. https://www.di.ens.fr/~pointche/Documents/Papers/2001_pkc.pdf

[3] H. Krawczyk, M. Bellare, and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication." Internet Engineering Task Force; RFC 2104 (Informational); IETF, Feb-1997. http://www.ietf.org/rfc/rfc2104.txt

[4] H. Krawczyk and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)." Internet Engineering Task Force; RFC 5869 (Informational); IETF, May-2010. http://www.ietf.org/rfc/rfc5869.txt

[5] D. J. Bernstein, M. Hamburg, A. Krasnova, and T. Lange, "Elligator: Elliptic-curve points indistinguishable from uniform random strings." Cryptology ePrint Archive, Report 2013/325, 2013. http://eprint.iacr.org/2013/325

[6] Markku-Juhani O. Saarinen, "Beyond Modes: Building a Secure Record Protocol from a Cryptographic Sponge Permutation." Cryptology ePrint Archive, Report 2013/772, 2013. http://eprint.iacr.org/2013/772

[7] A. Langley, M. Hamburg, and S. Turner, "Elliptic Curves for Security." Internet Engineering Task Force; RFC 7748 (Informational); IETF, Jan-2016. http://www.ietf.org/rfc/rfc7748.txt

[8] Y. Nir and A. Langley, "ChaCha20 and Poly1305 for IETF Protocols." Internet Engineering Task Force; RFC 7539 (Informational); IETF, May-2015. http://www.ietf.org/rfc/rfc7539.txt

[9] M. J. Dworkin, "SP 800-38D. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC," National Institute of Standards & Technology, Gaithersburg, MD, United States, 2007. http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38d.pdf

[10] NIST, "FIPS 180-4. Secure Hash Standard (SHS)," National Institute of Standards & Technology, Gaithersburg, MD, United States, 2012. http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf

[11] M.-J. Saarinen and J.-P. Aumasson, "The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC)." Internet Engineering Task Force; RFC 7693 (Informational); IETF, Nov-2015. http://www.ietf.org/rfc/rfc7693.txt

[12] H. Krawczyk, "'Cryptographic extraction and key derivation: The hkdf scheme'." Cryptology ePrint Archive, Report 2010/264, 2010. http://eprint.iacr.

org/2010/264

[13] D. J. Bernstein, T. Lange, and P. Schwabe, “NaCl: Networking and Cryptography Library.”. https://nacl.cr.yp.to/

[14] D. J. Bernstein, “CurveCP: Usable security for the Internet.”. https://curvecp.org

[15] H. Krawczyk, “SIGMA: The ‘SIGn-and-MAc’ Approach to Authenticated Diffie-Hellman and Its Use in the IKE Protocols,” in Advances in Cryptology - CRYPTO 2003, 2003. http://webee.technion.ac.il/~hugo/sigma.html

[16] S. Halevi and H. Krawczyk, “One-Pass HMQV and Asymmetric Key-Wrapping.” Cryptology ePrint Archive, Report 2010/638, 2010. http://eprint.iacr.org/2010/638

[17] I. Goldberg, D. Stebila, and B. Ustaoglu, “Anonymity and One-way Authentication in Key Exchange Protocols,” Design, Codes, and Cryptography, vol. 67, no. 2, May 2013. http://cacr.uwaterloo.ca/techreports/2011/cacr2011-11.pdf

[18] M. Di Raimondo, R. Gennaro, and H. Krawczyk, “Secure Off-the-record Messaging,” in Proceedings of the 2005 ACM Workshop on Privacy in the Electronic Society, 2005. http://www.dmi.unict.it/diraimondo/web/wp-content/uploads/papers/otr.pdf

[19] C. Kudla and K. G. Paterson, “Modular Security Proofs for Key Agreement Protocols,” in Advances in Cryptology - ASIACRYPT 2005: 11th International Conference on the Theory and Application of Cryptology and Information Security, 2005. http://www.isg.rhul.ac.uk/~kp/ModularProofs.pdf

[20] S. Blake-Wilson, D. Johnson, and A. Menezes, “Key agreement protocols and their security analysis,” in Crytography and Coding: 6th IMA International Conference Cirencester, UK, December 17–19, 1997 Proceedings, 1997. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.25.387

[21] M. Hamburg, “Key Exchange and DuplexWrap-like protocols.” Noise@moderncrypto.org Mailing List, 2015. https://moderncrypto.org/mail-archive/noise/2015/000098.html

[22] Mike Hamburg, “The STROBE protocol framework.” Cryptology ePrint Archive, Report 2017/003, 2017. http://eprint.iacr.org/2017/003

[23] T. Perrin and M. Marlinspike, “The Double Ratchet Algorithm,” 2016. https://whispersystems.org/docs/specifications/doubleratchet/