The NoiseSocket Protocol

Alexey Ermishkin (scratch@virgilsecurity.com)

Revision 0draft, 2017-05-01

Contents

Abstract

NoiseSocket is an extension of the Noise Protocol Framework (developed by the authors of Signal and currently used by WhatsApp) that enables quick and seamless Transport Layer Security (TLS) between multiple parties with minimal code space overhead, small keys, and extremely fast speed. NoiseSocket is designed to overcome the shortcomings of existing TLS implementations and

targets IoT devices, microservices, back-end applications such as datacenter-to-datacenter communications, and use cases where third-party certificate of authority infrastructure is not optimal.

Noise protocol framework

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 includes 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. 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 9.

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 DH 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 DH calculation between the private key in `key_pair` and `public_key` and returns an output sequence of bytes of length `DHLEN`. 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 8.1 it is recommended for the function to have a

- `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.

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 [2]).

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

- `HMAC-HASH(key, data)`: Applies `HMAC` from [2] using the `HASH()` function. This function is only called as part of `HKDF()`, below.
- `HKDF(chaining_key, input_key_material)`: 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 two byte sequences of length `HASHLEN`, as follows:
  - 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. Also note that the `HKDF()` function is simply `HKDF` from [3] 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 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 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 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". The maximum `n` value (2^{64}-1) is reserved for future use and must not be used. If incrementing `n` results in 2^{64}-1 (the maximum value), 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.
- **`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 10). 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 truncates `temp_k` to 32 bytes. Calls `InitializeKey(temp_k)`.
- **`MixHash(data)`**: Sets `h = HASH(h || data)`.
- **`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)`.
  - 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 `("s", "e", "ee", "es", "se", "ss")`.

A `HandshakeState` responds to the following methods:

- **`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.

  - Derives a `protocol_name` byte sequence by combining the names for the handshake pattern and crypto functions, as specified in Section 10. 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.
  - Fetches and deletes the next message pattern from `message_patterns`, then 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 `"xy"`: Calls `MixKey(DH(x, ry))` if `initiator`, otherwise `MixKey(DH(y, rx))`.
  - 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 and deletes the next message pattern from `message_patterns`, then sequentially processes each token from the message pattern:
    - For `"e"`: Sets `re` to the next `DHLEN` bytes from the message, overwriting any previous value for `re`. 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 "xy": Calls MixKey(DH(x, ry)) if initiator, otherwise 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 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 next section).

7. Handshake patterns

A **message pattern** is some sequence of tokens from the set ("e", "s", "ee", "es", "se", "ss").

A **handshake pattern** consists of:

- A pattern for the initiator's **pre-message** that is either:
    - "e"
    - "s"
    - "e, s"
    - 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 message 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, ee
```

The handshake pattern name is `Noise_NN`. This naming convention will be explained in Section 7.2. 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.

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, 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 an ephemeral public key, more than once per handshake (i.e. ignoring 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. Parties must send an 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"`).
4. After performing a DH between a remote public key and any local private key that is not an ephemeral private key, the local party must not send any encrypted data unless they have also performed a DH between an 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 and fourth 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.

7.2. Interactive patterns

The following example 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 responder
- `X` = Static key for responder `X`mitted ("transmitted") to initiator

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

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

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

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

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

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

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

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

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

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

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

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

The `Noise_XX` pattern is the most generically useful, since it is efficient and 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.

7.3. 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. 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
Noise_NN		
-> e	0	0
<- e, ee	0	1
->	0	1
Noise_NK		
<- s		
...		
-> e, es	0	2
<- e, ee	2	1
->	0	5
Noise_NX		
-> e	0	0
<- e, ee, s, es	2	1
->	0	5
Noise_XN		
-> e	0	0
<- e, ee	0	1
-> s, se	2	1
<-	0	5

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


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


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


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


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

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


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


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

---

8. Advanced uses

8.1. Dummy static public 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 `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.

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).

8.2. Channel binding

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

To support this, Noise libraries should expose the final value of `h` 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.

9. DH functions, cipher functions, and hash functions

9.1. The `25519` DH functions

- `GENERATE_KEYPAIR()`: Returns a new Curve25519 key pair.
- `DH(keypair, public_key)`: Executes the Curve25519 DH function (aka "X25519" in [4].
- `DHLEN` = 32

9.2. The `448` DH functions

- `GENERATE_KEYPAIR()`: Returns a new Curve448 key pair.
- `DH(keypair, public_key)`: Executes the Curve448 DH function (aka "X448" in [4].
- `DHLEN` = 56

9.3. The `ChaChaPoly` cipher functions

- `ENCRYPT(k, n, ad, plaintext)` / `DECRYPT(k, n, ad, ciphertext)`: `AEAD_CHACHA20_POLY1305` from [5]. 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).

9.4. The AESGCM cipher functions

- **ENCRYPT(k, n, ad, plaintext) / DECRYPT(k, n, ad, ciphertext)**: AES256 with GCM from [6] 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**.

9.5. The SHA256 hash function

- **HASH(input)**: SHA-256 from [7].
- **HASHLEN** = 32
- **BLOCKLEN** = 64

9.6. The SHA512 hash function

- **HASH(input)**: SHA-512 from [7].
- **HASHLEN** = 64
- **BLOCKLEN** = 128

9.7. The BLAKE2s hash function

- **HASH(input)**: BLAKE2s from [8] with digest length 32.
- **HASHLEN** = 32
- **BLOCKLEN** = 64

9.8. The BLAKE2b hash function

- **HASH(input)**: BLAKE2b from [8] with digest length 64.
- **HASHLEN** = 64
- **BLOCKLEN** = 128

10. Protocol names

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

- Noise_XX_25519_AESGCM_SHA256
- Noise_N_25519_ChaChaPoly_BLAKE2s
- Noise_XXfallback_448_AESGCM_SHA512
- Noise_IK_448_ChaChaPoly_BLAKE2b

11. 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 some 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 some extra security in case a cryptanalytic attack is developed against the smaller hash functions.
- **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 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). A recommended idiom is for the value zero to indicate no change from the current Noise protocol, and for applications to reject messages with an unknown value. This allows future protocol versions to specify handshake re-initialization or any other compatibility-breaking change (protocol extensions that don't break compatibility can be handled within Noise payloads).

12. Security considerations

This section collects various security considerations:

- **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 handshake re-initialization, 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 must be passed in 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 previous bullet). 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 8.2.
- **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 for future use. This means parties are not allowed to send more than 2^{64}-1 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 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.

- **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.
- **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.

13. Rationale

This section collects various design rationale:

Nonces are 64 bits in length 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 recommended hash function families are SHA2 and BLAKE2 because:

- SHA2 is widely available.
- SHA2 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:

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

- SHA-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 data in a ciphertext 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 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.

Cipher nonces are big-endian for AES-GCM, 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.

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

- HKDF applies multiple layers of hashing between each `MixKey()` input. This "extra" hashing might mitigate the impact of hash function weakness.
- HKDF is well-known and is used in similar ways in other protocols (e.g. Signal, IPsec).
- HKDF and HMAC are widely-used constructions. If some weakness is found in a hash function, cryptanalysts will likely analyze that weakness in the context of HKDF and HMAC.

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

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.

NoiseSocket protocol

NoiseSocket describes how to compose and parse handshake and transport messages, do versioning and negotiation. There is only one mandatory pattern that must be present in any first handshake message: Noise_XX. Noise_XX allows any combination of authentications (client, server, mutual, none).

Other patterns may be supported by concrete implementations, but at least one Noise_XX message must be included first in any first message

Traffic in NoiseSocket is split into packets each less than or equal to 65535 bytes (2^16 - 1) which allows for easy parsing and memory management.

All numbers are in big endian form.

1. Packet structure

Both handshake and transport packets have the following structure:

- 2 bytes packet size
- Data

2. Handshake packets

The handshake process consists of set of messages which client and server send to each other. First two of them have a specific data structure

2.1. First handshake message

In the **First handshake message** client offers server a set of sub-messages, identified by a string name. For example, Noise protocol.

Each handshake sub-message contains following fields:

- 1 byte length of the following string, indicating the ciphersuite/protocol used, i.e. message type
- String indicating message type
- 2 bytes big-endian length of following sub-message
- **Sub message**

When using Noise, **Sub message** is received by calling **WriteMessage** on the corresponding HandshakeState

2.2. Second handshake message

In the **Second handshake message** server responds to client with the following structure:

- 1 byte sub-message index server responds to
- Handshake message

3. Prologue

Noise prologue is calculated as follows:

- 1 byte number of message types (N)
- N times:
  - 1 byte message type length
  - Message type (ex. Noise protocol string)

An example of such prologue could be found in Appendix

4. Data packets

After handshake is complete and both Cipher states are created, all following packets must be encrypted using those cipherstates.

5. API

We present a set of methods which will help to implement NoiseSocket flow

- `ReadString(buffer)` reads 1 byte `len` of the following string from `buffer` and then `len` bytes string itself. Advances read position to `len + 1`
- `WriteString(string, buffer)` writes 1 byte `len` of the following string to `buffer` and then string itself.
- `ReadData(buffer)` reads 2 byte `len` of the following data from `buffer` and then `len` bytes of data. Advances position to `len + 2`
- `WriteData(data, buffer)` writes 2 bytes `len` of the following data to `buffer` and then the data itself.
- `CalculatePrologue(protocols)` takes a list of protocol names in the order they will be used in the handshake.

  Variables:

  - `prologue_buffer` - a byte buffer to write to

Algorithm:

- Writes 1 byte number of protocols `N` to `prologue_buffer`
- Does `N` times:
  - Takes the next `protocol_name` from `protocols`
  - Calls `WriteString(protocol_name, prologue_buffer)`

Returns:

- `prologue_buffer`

- **`ComposeInitiatorHandshakeMessages(s, data, protocols)`** takes client's static key `s`, optional payload `data` and a list of protocols, same that was used for caling `CalculatePrologue` Variables:
  - **`result_buffer`** - buffer, containing the resulting byte sequence
  - **`message_buffer`** - temporary buffer to hold the result of calling `WriteMessage` on the current `handshake_state`
  - **`handshake_states`** - an array of all instances of `HandshakeState` objects, created during this method

  Algorithm:

  - Calls `CalculatePrologue(protocols)` to receive `prologue`
  - Writes the 1 byte number of protocols `N` to `result_buffer`
  - Does `N` times
    - Takes the next `protocol` from `protocols`
    - Calls `WriteString(protocol_name, result_buffer)`
    - Calls `GENERATE_KEYPAIR()` to generate new `e`
    - Initializes new `HandshakeState` instance with `DH functions`, `Cipher functions` and `Hash functions`, described in `protocol` and also `s`, `e` and `prologue` to receive `handshake_state`
    - initializes new `message_buffer`
    - Calls `WriteMessage(data, message_buffer)` on `handshake_state`
    - Calls `WriteData(message_buffer, result_buffer)`
    - Adds `handshake_state` to `handshake_states`

  Returns:

  - `result_buffer`
  - `handshake_states`

- **`ParseFirstMessage(message, s)`** receives `message`, created by calling `ComposeInitiatorHandshakeMessages` and static keypair `s` Variables:
  - **`protocols`** - a list of protocol names, parsed from `message`
  - **`sub_messages`** - a list of byte sequences in order they were written to `message` by calling `WriteMessage`
  - **`handshake_state`** - a state that was created when server chose one of the incoming messages
  - **`message_index`** - an index of the message that server chose

- **payload** - an optional payload, provided in the first message

Algorithm:

- Reads 1 byte number of sub-messages `N`
- Does `N` times:
  - Calls `ReadString(message)` to receive `protocol_name`
  - Calls `ReadData(message)` to receive `sub_message`
  - Appends `protocol_name` to `protocols`
  - Appends `sub_message` to `sub_messages`
- Calls `CalculatePrologue(protocols)` to receive `prologue`
- Chooses a protocol, according to server protocol priority and the corresponding `sub_message` from `sub_messages`.
- Writes index of the chosen `protocol` to `message_index`
- Calls `GENERATE_KEYPAIR()` to generate new `e`
- Initializes new `HandshakeState` instance with `DH functions`, `Cipher functions` and `Hash functions`, described in `protocol` and also `s`, `e` and `prologue` to receive `handshake_state`
- Calls `ReadMessage(sub_messages)` on `handshake_state` to receive an optional `payload`

Returns:

- `message_index`
- `handshake_state`
- `payload`

- **`ComposeServerResponseMessage(handshake_state, index, data)`** receives `handshake_state`, `index` and optional `data` to send to client

  Variables:

  - **`result_buffer`** - buffer, containing the resulting byte sequence

  Algorithm:

  - Writes `index` to `result_buffer`
  - Calls WriteMessage(result_buffer, data) on handshake_state

  Returns:

  - `result_buffer`

- **`SetMaxPacketLen(len)`** Sets the global valiable **`max_packet_len`** to `len`. `len` must satisfy the following condition: $127 <$ `len` < 65536 . The default value of **`max_packet_len`** is 65535

- **`GetMaxPacketLen`** Returns the number of bytes of the plaintext that can be placed into the current packet

  Algorithm:

  - If handshake is not complete return **`max_packet_len`**

  - Else return `max_packet_len` - `CIPHER_OVERHEAD` . `CIPHER_OVERHEAD` is the number of bytes added by the chosen symmetric cipher

- `Write(data)` writes data to outgoing socket

  Variables:

  - `buffer` - temporary buffer

  Algorithm:

  - While length(data) > 0
    *

  Returns:

  - `result_buffer`

6. Test vectors

Test vectors consist of one initial message, prologue and a set of private keys. Two for initiator (static, ephemeral) and two for responder.

Initial message contains 16 sub-messages each correspond to a specific Noise protocol. The order of protocols can be seen in `Protocols` array.

"Server" chooses which sub-message to answer and this forms a `session`. Each session contains an array of transport messages which consist of raw wire data ("Packet" field), and payload

7. IPR

8. 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] 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

[3] 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

[4] 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

[5] 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

[6] 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

[7] 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

[8] 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