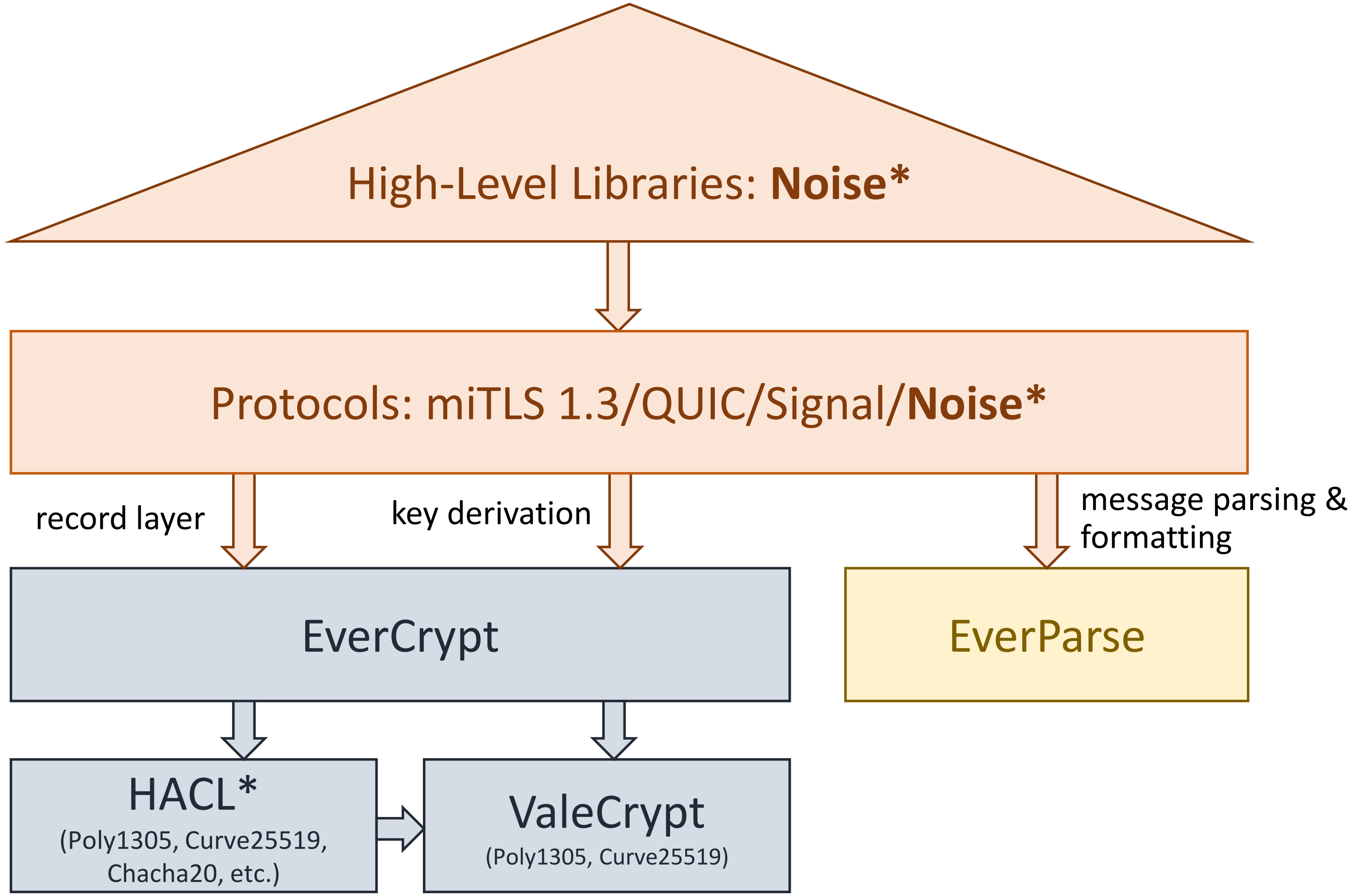
Noise*,
A Library of Verified High-Performance Secure Channel Protocol Implementations

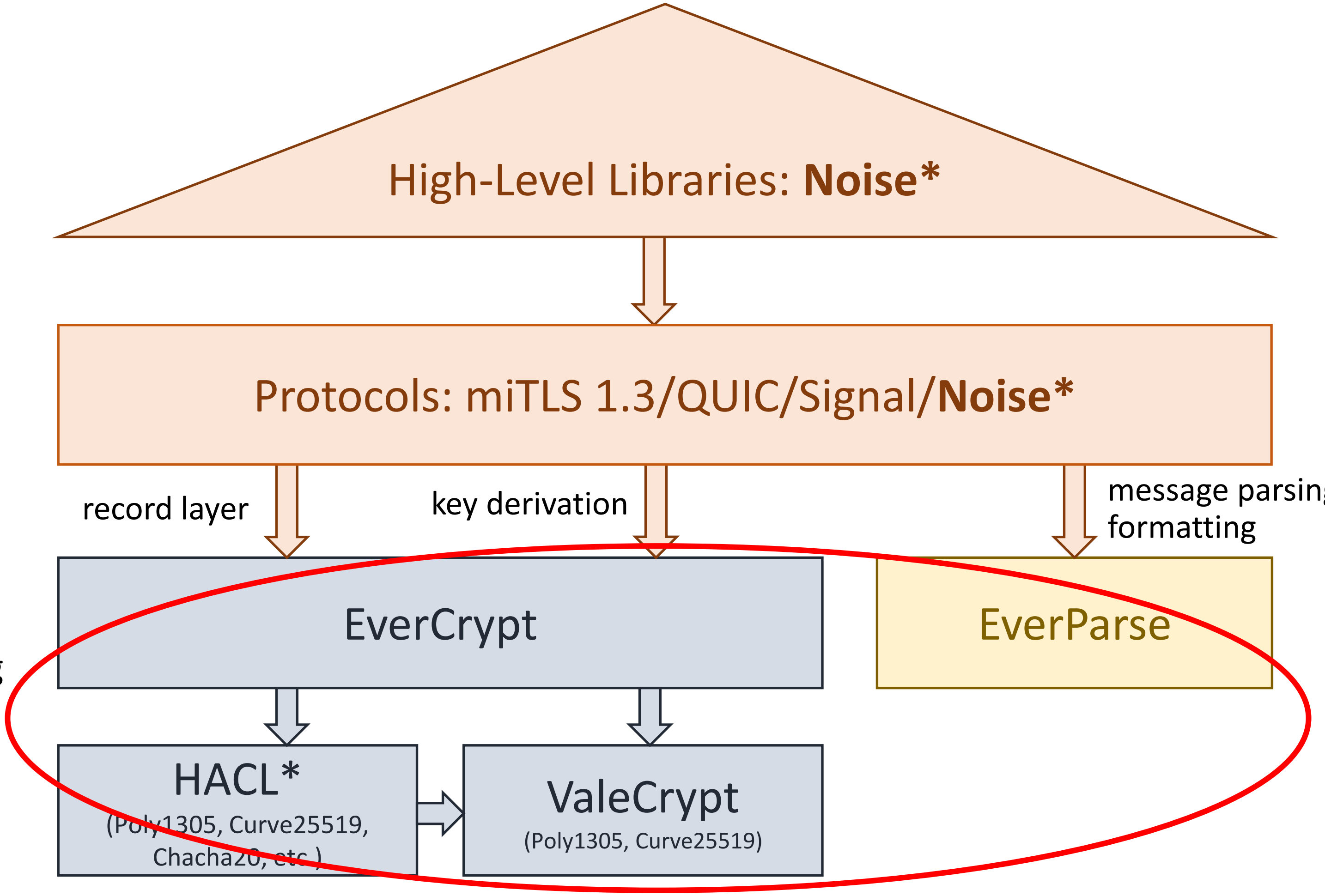
S. Ho, J. Protzenko, A. Bichhawat, K. Bhargavan

Microsoft Research - Inria
JOINT CENTRE
Inria
INVENTEURS DU MONDE NUMÉRIQUE
Microsoft
INDIAN INSTITUTE OF TECHNOLOGY GANDHINAGAR
भारतीय प्रौद्योगिकी संस्थान गांधीनगर

Everest: Verified Components for the HTTPS Ecosystem

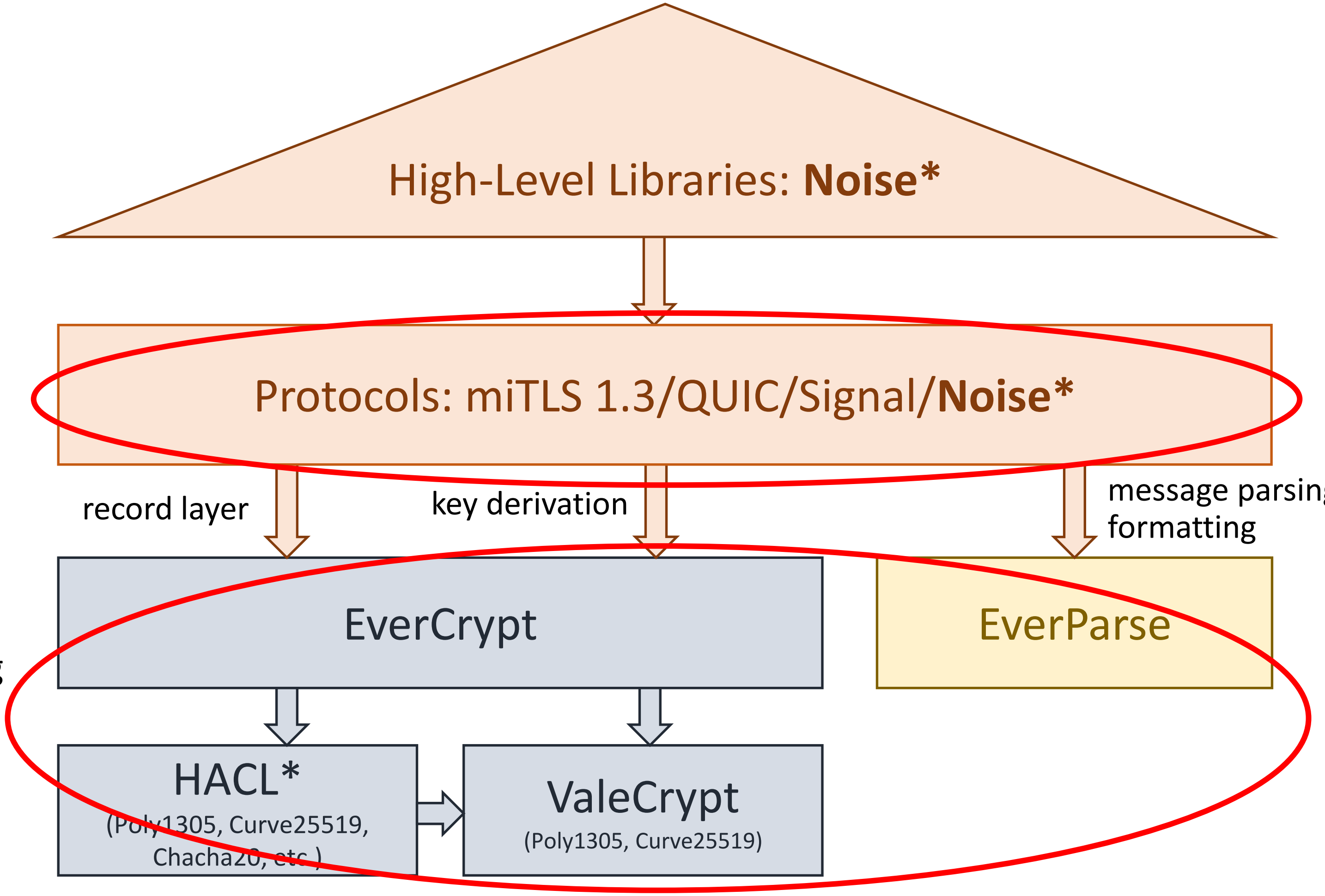


Everest: Verified Components for the HTTPS Ecosystem



Memory and type safety (buffer overflows, dangling pointers...), functional correctness (optimized implementations are "correct"), side channels...

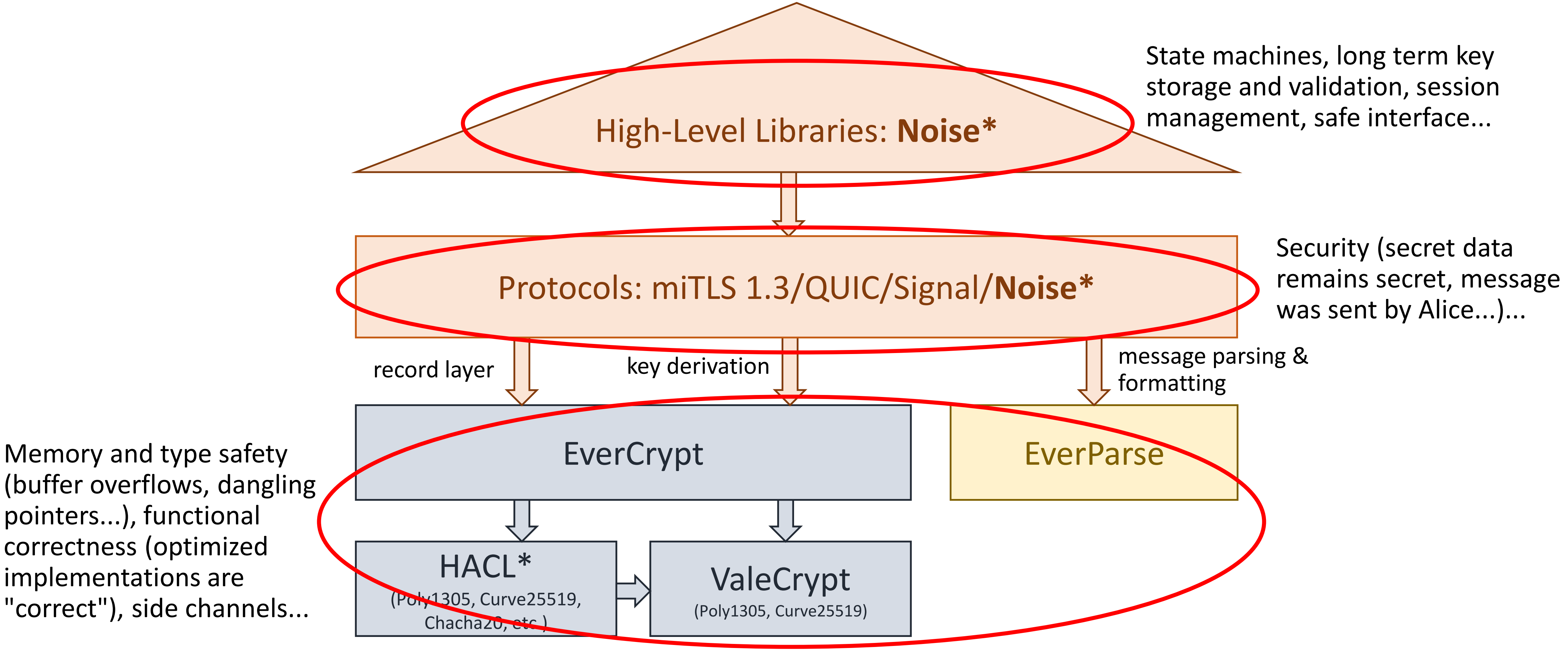
Everest: Verified Components for the HTTPS Ecosystem



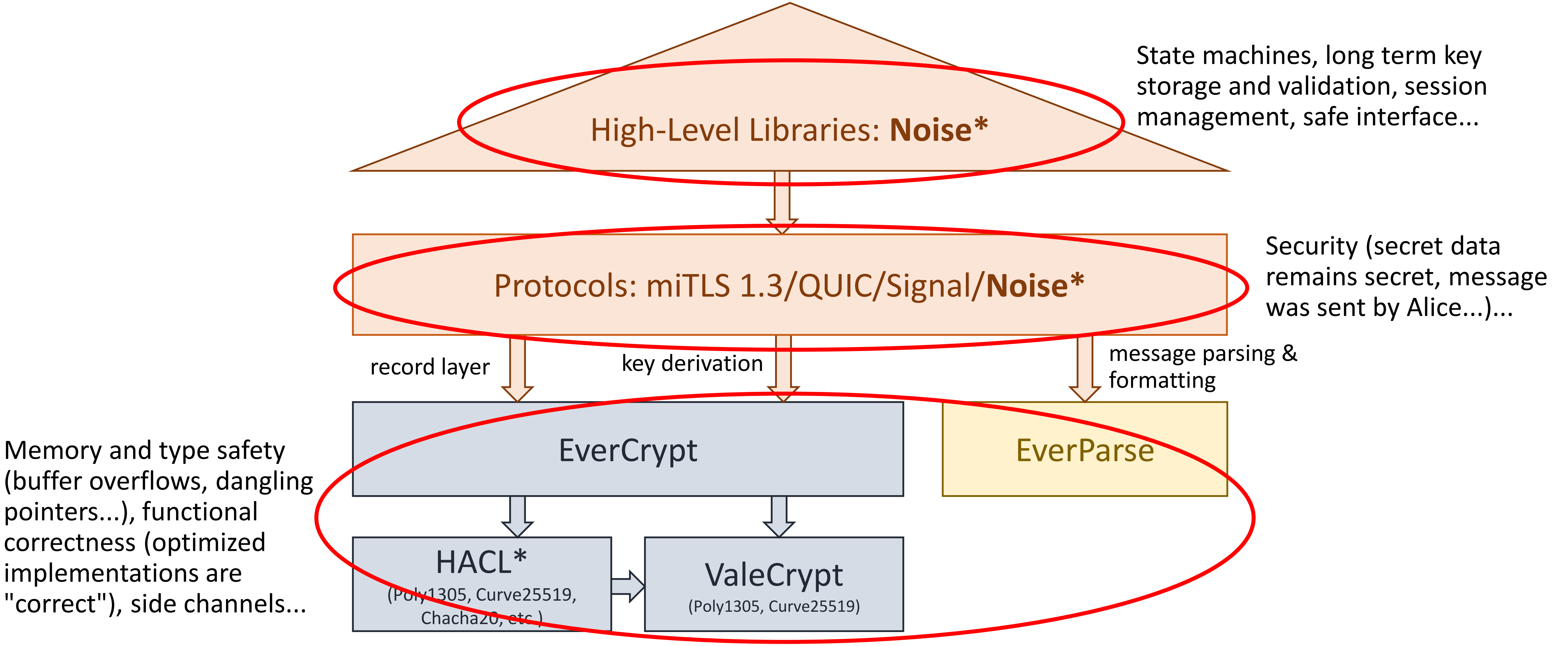
Security (secret data remains secret, message was sent by Alice...)...

Memory and type safety (buffer overflows, dangling pointers...), functional correctness (optimized implementations are "correct"), side channels...

Everest: Verified Components for the HTTPS Ecosystem

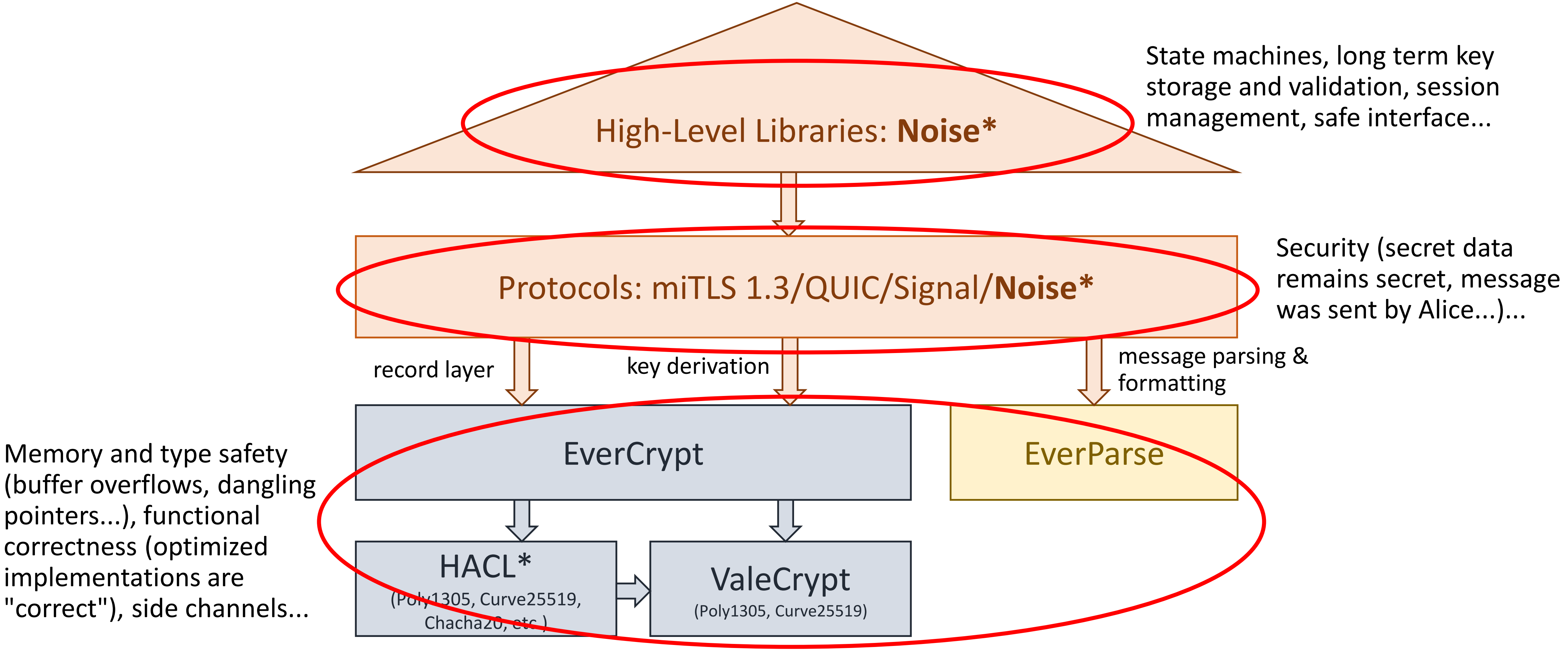


Everest: Verified Components for the HTTPS Ecosystem



Formal methods: formally specify how components should behave and *prove* they satisfy those properties

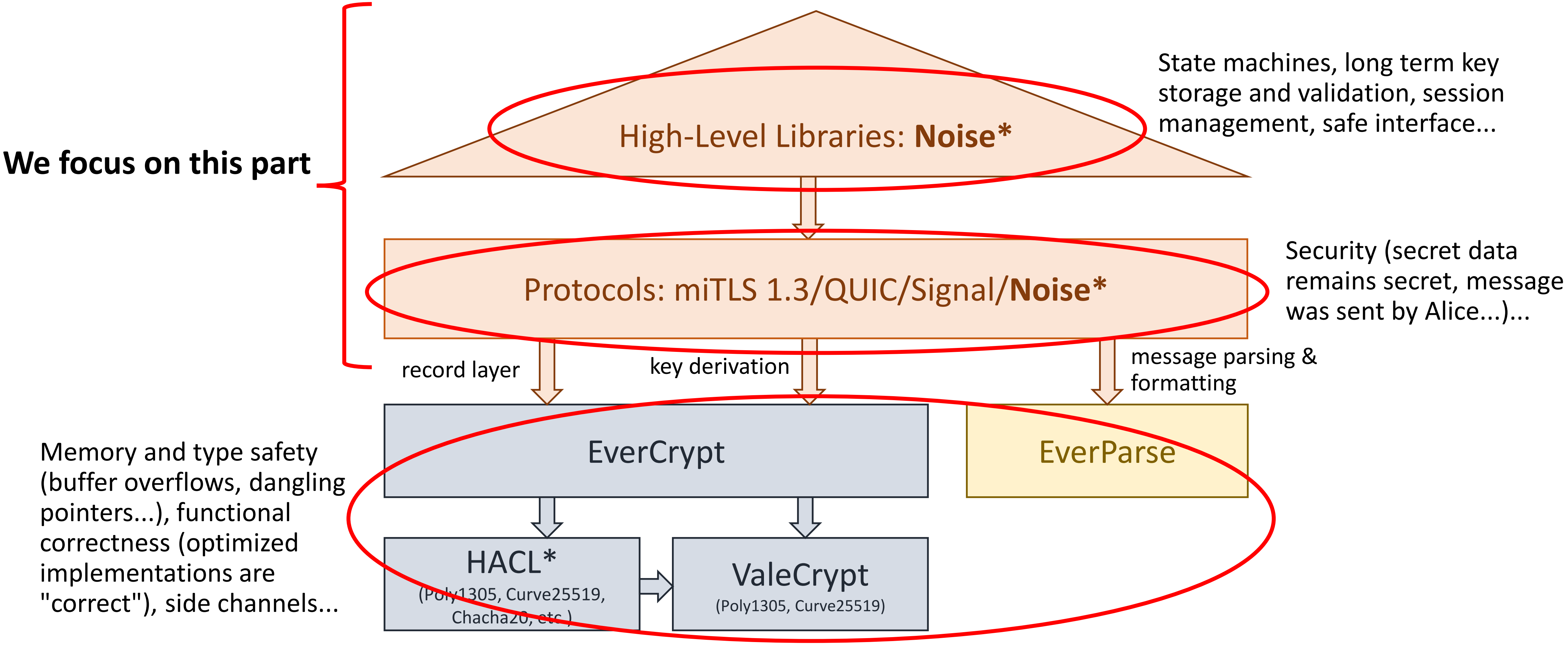
Everest: Verified Components for the HTTPS Ecosystem



Formal methods: formally specify how components should behave and *prove* they satisfy those properties

A lot of protocols! Systematic results? Noise covers 59+ protocols used in WhatsApp, Wireguard, Signal, Facebook Messenger

Everest: Verified Components for the HTTPS Ecosystem



Formal methods: formally specify how components should behave and *prove* they satisfy those properties

A lot of protocols! Systematic results? Noise covers 59+ protocols used in WhatsApp, Wireguard, Signal, Facebook Messenger

What is Noise?

- **What does a handshake protocol do?**
  - Exchange data to have a **shared secret** to communicate
  - Various use cases (one-way encryption, authenticated servers, mutual authentication, etc.)
  - Varying security

What is Noise?

- **What does a handshake protocol do?**
  - Exchange data to have a **shared secret** to communicate
  - Various use cases (one-way encryption, authenticated servers, mutual authentication, etc.)
  - Varying security
- Various protocols, some of them **very advanced and complex** (ex.: TLS):
  - Backward compatibility
  - Cipher suites negotiation
  - Session resumption
  - …

What is Noise?

- **What does a handshake protocol do?**
  - Exchange data to have a **shared secret** to communicate
  - Various use cases (one-way encryption, authenticated servers, mutual authentication, etc.)
  - Varying security
- Various protocols, some of them **very advanced and complex** (ex.: TLS):
  - Backward compatibility
  - Cipher suites negotiation
  - Session resumption
  - …
- When advanced features not needed: **Noise** family of protocols

Noise Protocol Framework : Examples

X:

← s

. . .

→ e, es, s, ss

IK:

← s

. . .

→ e, es, s, ss

← e, ee, se

IKpsk2:

← s

. . .

→ e, es, s, ss

← e, ee, se, psk

NX:

→ e

← e, ee, s, es

XX:

→ e

← e, ee, s, es

→ s, se

XK:

← s

. . .

→ e, es

← e, ee

→ s, se

Today: **59+ protocols** (but might increase)

Noise Protocol Framework : Examples

X:

← s

. . .

→ e, es, s, ss

(one-way encryption: NaCl Box, HPKE...)

IK:

← s

. . .

→ e, es, s, ss

← e, ee, se

IKpsk2:

← s

. . .

→ e, es, s, ss

← e, ee, se, psk

NX:

→ e

← e, ee, s, es

XX:

→ e

← e, ee, s, es

→ s, se

XK:

← s

. . .

→ e, es

← e, ee

→ s, se

Today: **59+ protocols** (but might increase)

Noise Protocol Framework : Examples

X:

← s

. . .

→ e, es, s, ss

(one-way encryption: NaCl Box, HPKE...)

IK:

← s

. . .

→ e, es, s, ss

← e, ee, se

IKpsk2:

← s

. . .

→ e, es, s, ss

← e, ee, se, psk

NX:

→ e

← e, ee, s, es

(authenticated server)

XX:

→ e

← e, ee, s, es

→ s, se

XK:

← s

. . .

→ e, es

← e, ee

→ s, se

Today: **59+ protocols** (but might increase)

Noise Protocol Framework : Examples

X:

← s

. . .

→ e, es, s, ss

(one-way encryption: NaCl Box, HPKE...)

IK:

← s

. . .

→ e, es, s, ss

← e, ee, se

IKpsk2:

← s

. . .

→ e, es, s, ss

← e, ee, se, psk

(mutual authentication and 0-RTT)

NX:

→ e

← e, ee, s, es

(authenticated server)

XX:

→ e

← e, ee, s, es

→ s, se

XK:

← s

. . .

→ e, es

← e, ee

→ s, se

Today: **59+ protocols** (but might increase)

Noise Protocol Framework : Examples

X:

← s

. . .

→ e, es, s, ss

(one-way encryption: NaCl Box, HPKE...)

IK: **WhatsApp**

← s

. . .

→ e, es, s, ss

← e, ee, se

IKpsk2: **Wireguard VPN**

← s

. . .

→ e, es, s, ss

← e, ee, se, psk

(mutual authentication and 0-RTT)

NX:

→ e

← e, ee, s, es

(authenticated server)

XX:

→ e

← e, ee, s, es

→ s, se

XK: **Lightning, I2P**

← s

. . .

→ e, es

← e, ee

→ s, se

Today: **59+ protocols** (but might increase)

Noise Protocol Example: IKpsk2

IKpsk2:

← s

. . .

→ e, es, s, ss

← e, ee, se, psk

Noise Protocol Example: IKpsk2

IKpsk2:

← s

. . .

→ e, es, s, ss, [d0]

← e, ee, se, psk, [d1]

↔ [d2, d3, …]

Noise Protocol Example: IKpsk2

Initiator **Responder**

IKpsk2:

← s

. . .

→ e, es, s, ss, [d0]

← e, ee, se, psk, [d1]

↔ [d2, d3, …]

The handshake describes how to:

- Exchange key material
- Use those to derive shared secrets (Diffie-Hellman operations…)
- Send/receive encrypted data

Noise Protocol Example: IKpsk2

Initiator **Responder**

IKpsk2:

← s

. . . **Exchange key material**

→ e, es, s, ss, [d0]

← e, ee, se, psk, [d1]

↔ [d2, d3, …]

The handshake describes how to:

- **Exchange key material**
- Use those to derive shared secrets (Diffie-Hellman operations…)
- Send/receive encrypted data

Noise Protocol Example: IKpsk2

Initiator **Responder**

IKpsk2:

← s

. . .

→ e, es, s, ss, [d0]

← e, ee, se, psk, [d1]

↔ [d2, d3, …]

s: static
e: ephemeral

The handshake describes how to:

- **Exchange key material**
- Use those to derive shared secrets (Diffie-Hellman operations…)
- Send/receive encrypted data

Noise Protocol Example: IKpsk2

Initiator **Responder**

IKpsk2:

← s

. . . **Derive shared secrets (Diffie-Hellman operations…)**

→ e, es, s, ss, [d0]

← e, ee, se, psk, [d1]

↔ [d2, d3, …]

The handshake describes how to:

- Exchange key material
- **Use those to derive shared secrets (Diffie-Hellman operations…)**
- Send/receive encrypted data

Noise Protocol Example: IKpsk2

Initiator **Responder**

IKpsk2:

← s

. . .

→ e, es, s, ss, [d0]

← e, ee, se, psk, [d1] **Send/receive encrypted data**

↔ [d2, d3, …]

The handshake describes how to:

- Exchange key material
- Use those to derive shared secrets (Diffie-Hellman operations…)
- **Send/receive encrypted data**

Noise Protocol Example: IKpsk2

Initiator **Responder**

IKpsk2:

Premessages:

← s

. . .

→ e, es, s, ss, [d0]

← e, ee, se, psk, [d1]

↔ [d2, d3, …]

The handshake describes how to:

- Exchange key material
- Use those to derive shared secrets (Diffie-Hellman operations…)
- Send/receive encrypted data

Noise Protocol Example: IKpsk2

Initiator **Responder**

IKpsk2:

Premessages
← s
. . .

Handshake Messages
→ e, es, s, ss, [d0]
← e, ee, se, psk, [d1]

↔ [d2, d3, …]

The handshake describes how to:

- Exchange key material
- Use those to derive shared secrets (Diffie-Hellman operations…)
- Send/receive encrypted data

Noise Protocol Example: IKpsk2

Initiator **Responder**

IKpsk2:

Premessages
- ← s
- . . .

Handshake Messages
- → e, es, s, ss, [d0]
- ← e, ee, se, psk, [d1]

Transport
- ↔ [d2, d3, …]

The handshake describes how to:

- Exchange key material
- Use those to derive shared secrets (Diffie-Hellman operations…)
- Send/receive encrypted data

Noise Protocol Example: IKpsk2

Initiator **Responder**

IKpsk2:

Premessages
- ← s
- . . .

Handshake Messages
- → e, es, s, ss, [d0]
- ← e, ee, se, psk, [d1]

Transport
- ↔ [d2, d3, …]

The handshake describes how to:
- Exchange key material
- Use those to derive shared secrets (Diffie-Hellman operations…)
- Send/receive encrypted data

Secrets are **chained**:
- d0 encrypted with a key derived from es, ss
- d1 encrypted with a key derived from es, ss, ee, se, psk

⇒ **The more the handshake progresses, the more secure the shared secrets are**

Noise Protocol Example: IKpsk2

Initiator **Responder**

IKpsk2:

Premessages
- ← s
- . . .

Handshake Messages
- → e, es, s, ss, [d0]
- ← e, ee, se, psk, [d1]

Transport
- ↔ [d2, d3, …]

The handshake describes how to:
- Exchange key material
- Use those to derive shared secrets (Diffie-Hellman operations…)
- Send/receive encrypted data

Secrets are **chained**:
- d0 encrypted with a key derived from es, ss
- d1 encrypted with a key derived from es, ss, ee, se, psk

⇒ The more the handshake progresses, the more secure the shared secrets are

Noise Protocol Example: IKpsk2

Initiator **Responder**

IKpsk2:

Premessages
← s
. . .

Handshake Messages
→ e, es, s, ss, [d0]
← e, ee, se, psk, [d1]

Transport
↔ [d2, d3, …]

The handshake describes how to:

- Exchange key material
- Use those to derive shared secrets (Diffie-Hellman operations…)
- Send/receive encrypted data

Secrets are **chained**:

- d0 encrypted with a key derived from es, ss
- d1 encrypted with a key derived from es, ss, ee, se, psk

⇒ **The more the handshake progresses, the more secure the shared secrets are**

Noise Protocol Example: IKpsk2

Initiator **Responder**

IKpsk2:

Premessages
← s
. . .

Handshake Messages
→ e, es, s, ss, [d0]
← e, ee, se, psk, [d1]

Transport
↔ [d2, d3, …]

Early data (TLS 0-RTT, 0.5-RTT)

The handshake describes how to:

- Exchange key material
- Use those to derive shared secrets (Diffie-Hellman operations…)
- Send/receive encrypted data

Secrets are **chained**:

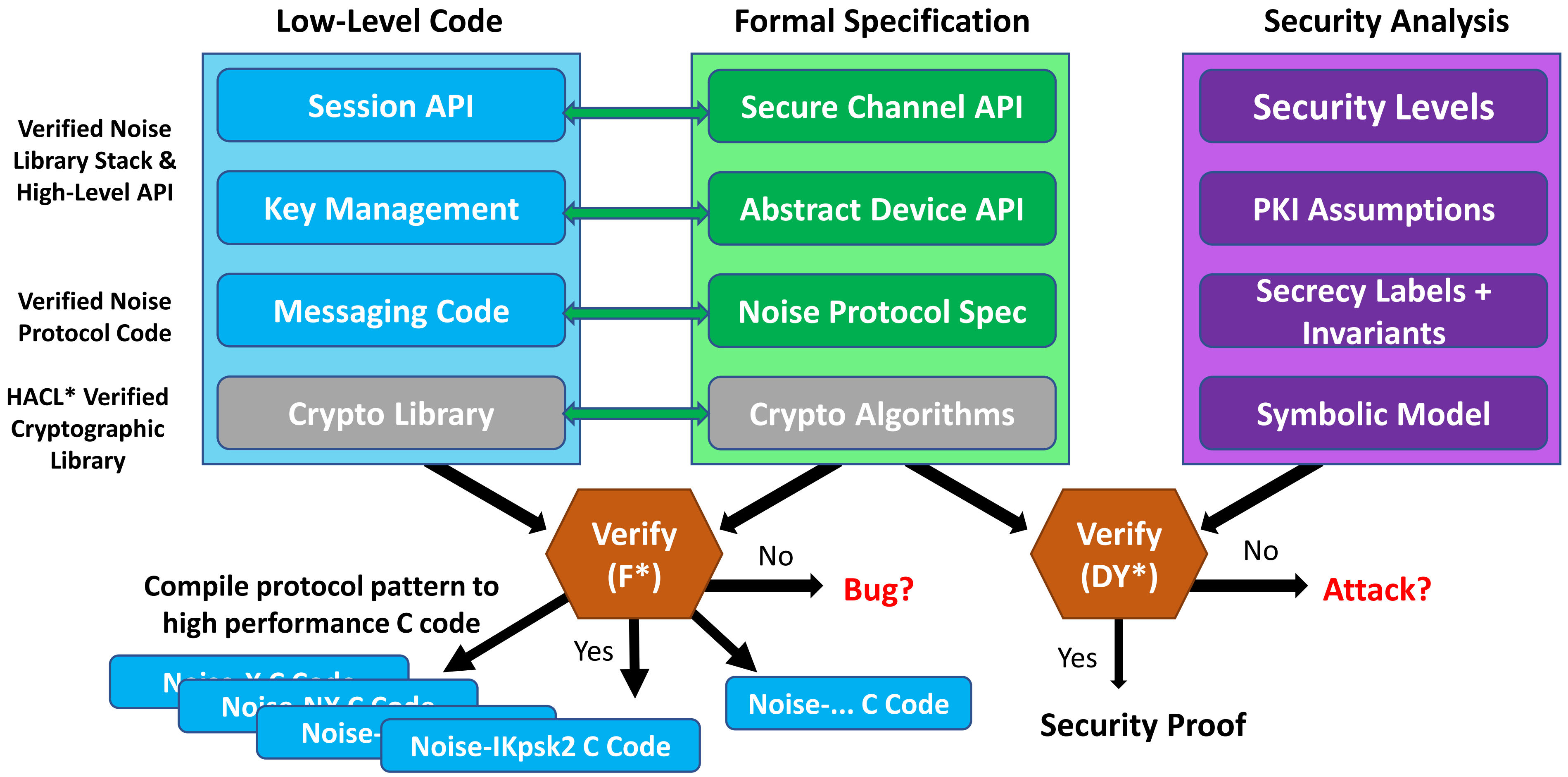
- d0 encrypted with a key derived from es, ss
- d1 encrypted with a key derived from es, ss, ee, se, psk

⇒ **The more the handshake progresses, the more secure the shared secrets are**

What is **Noise***?

Correctly implemented protocols?

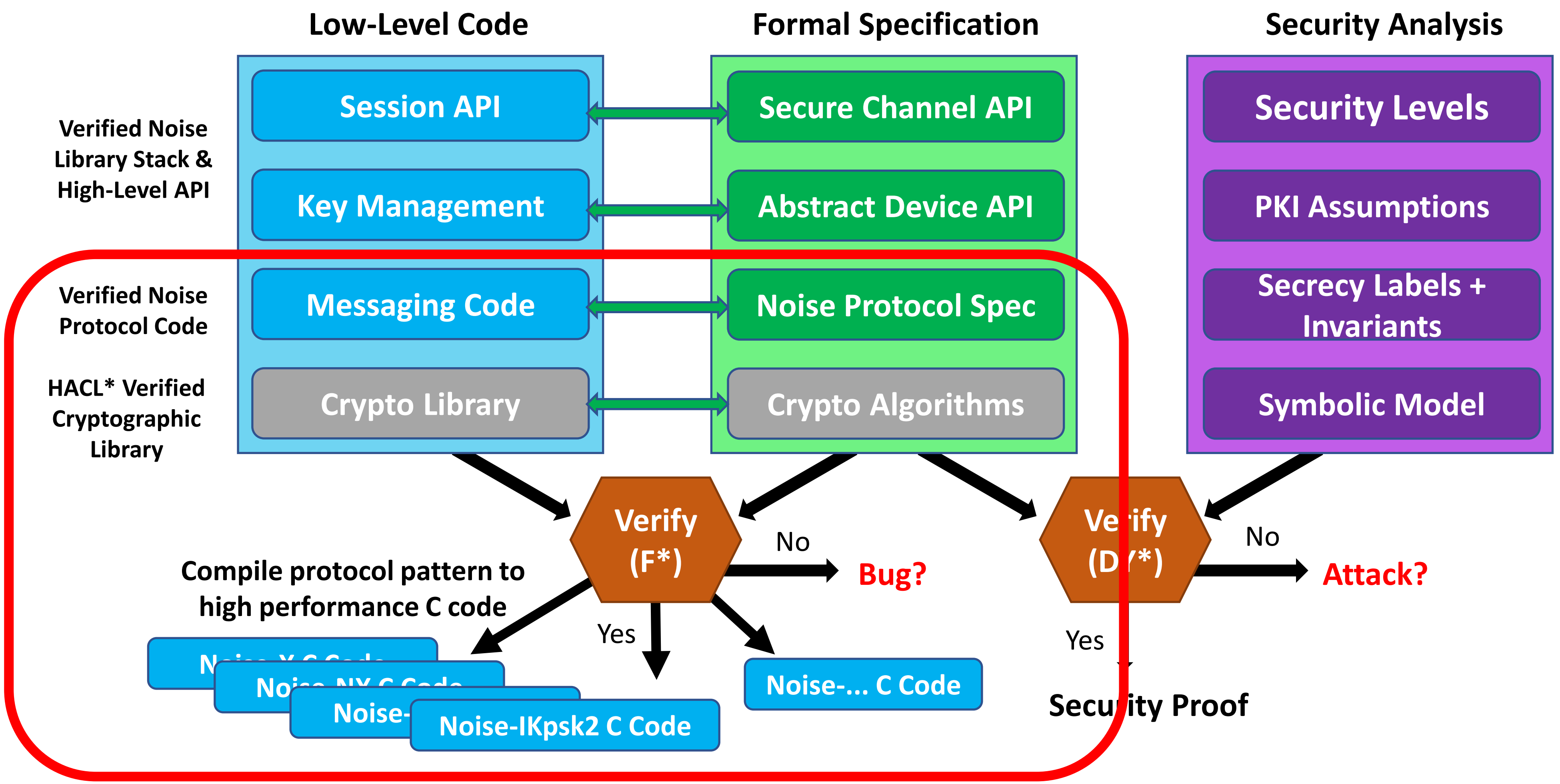
- **Noise* compiler**: Noise protocol "pattern" → verified, specialized C implementation
- On top: complete, verified **library stack** exposed through a **high-level, defensive API**
- Complemented with a formal **symbolic security analysis**



What is **Noise***?

Correctly implemented protocols?

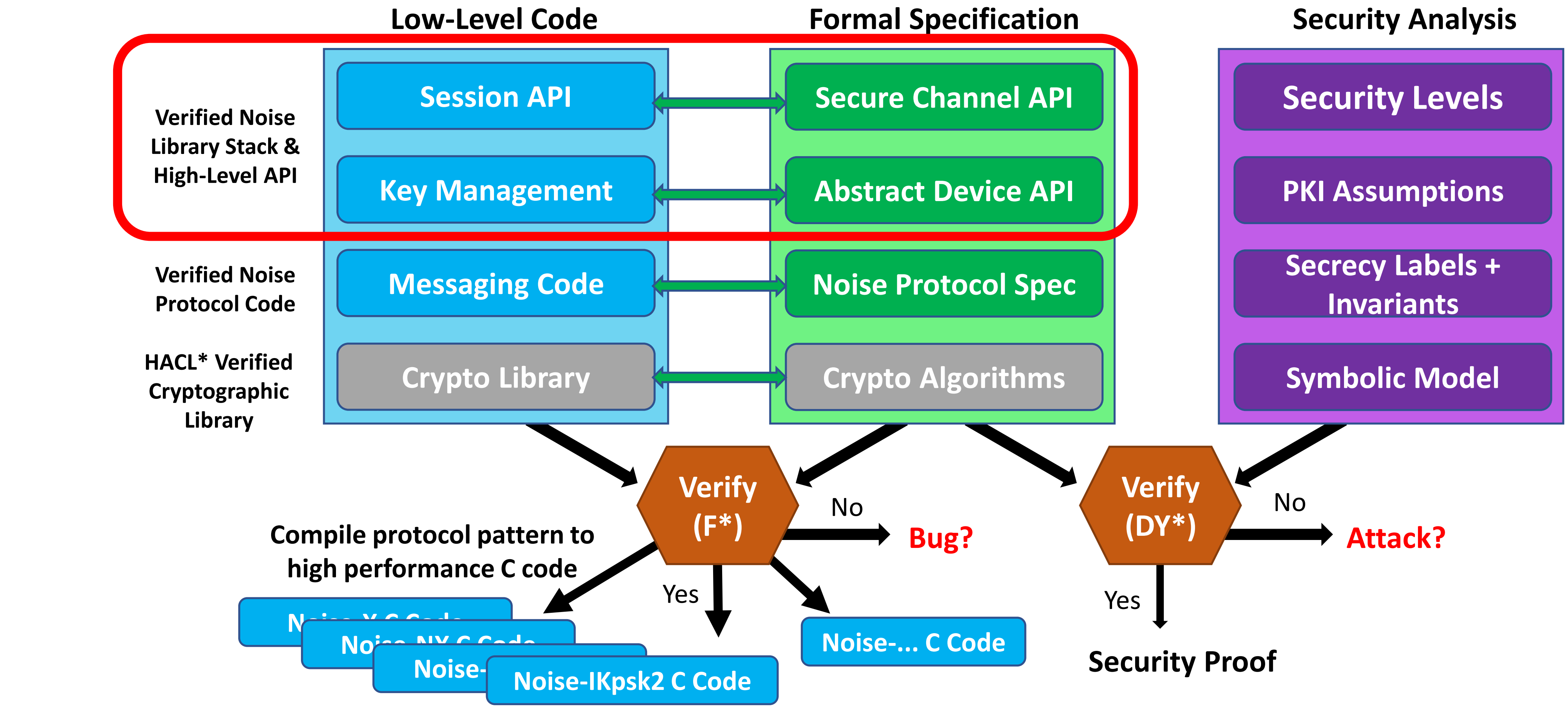
- **Noise* compiler**: Noise protocol “pattern” → verified, specialized C implementation
- On top: complete, verified **library stack** exposed through a **high-level, defensive API**
- Complemented with a formal **symbolic security analysis**



What is **Noise***?

Correctly implemented protocols?

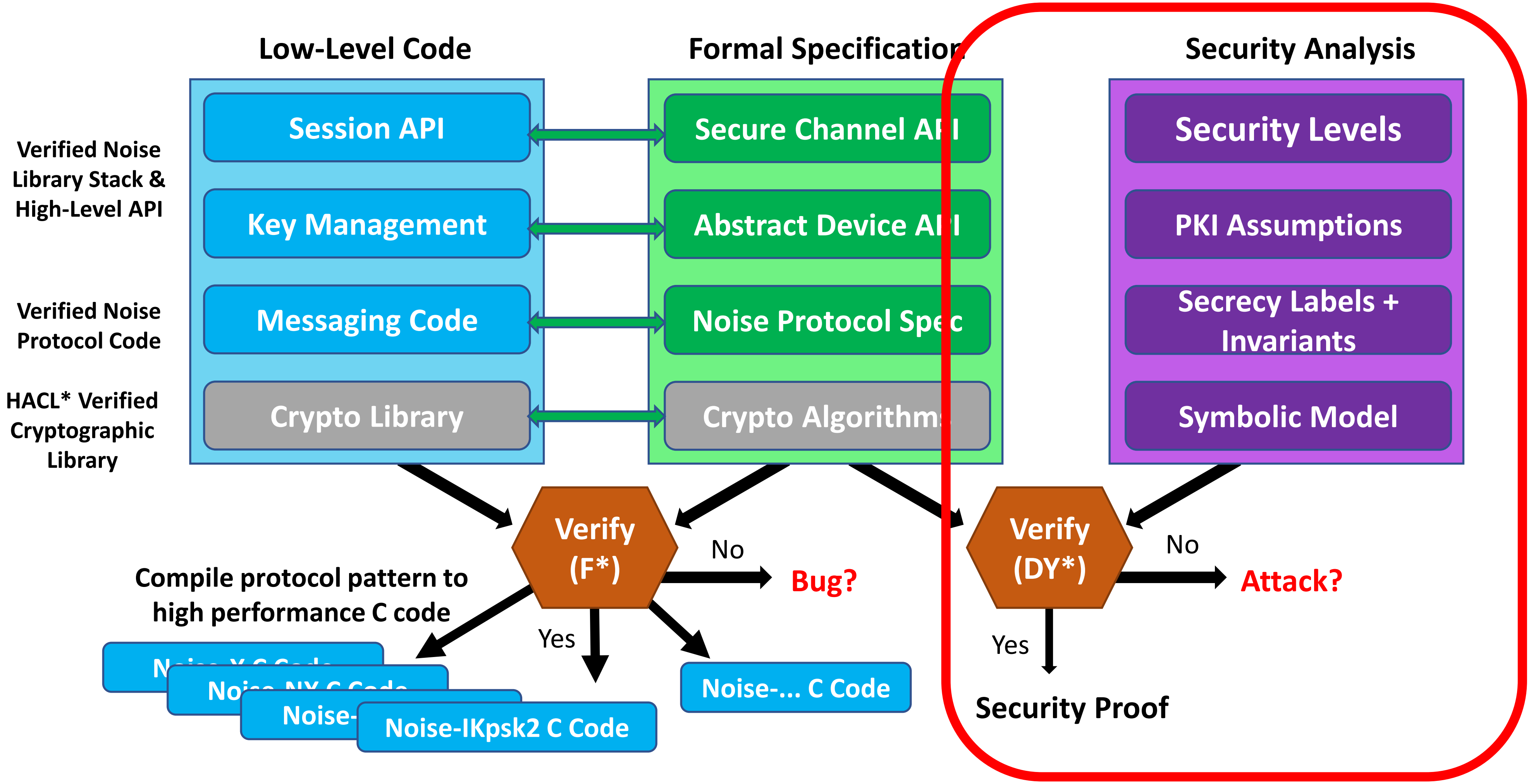
- **Noise* compiler**: Noise protocol “pattern” → verified, specialized C implementation
- On top: complete, verified **library stack** exposed through a **high-level, defensive API**
- Complemented with a formal **symbolic security analysis**



What is **Noise***?

Correctly implemented protocols?

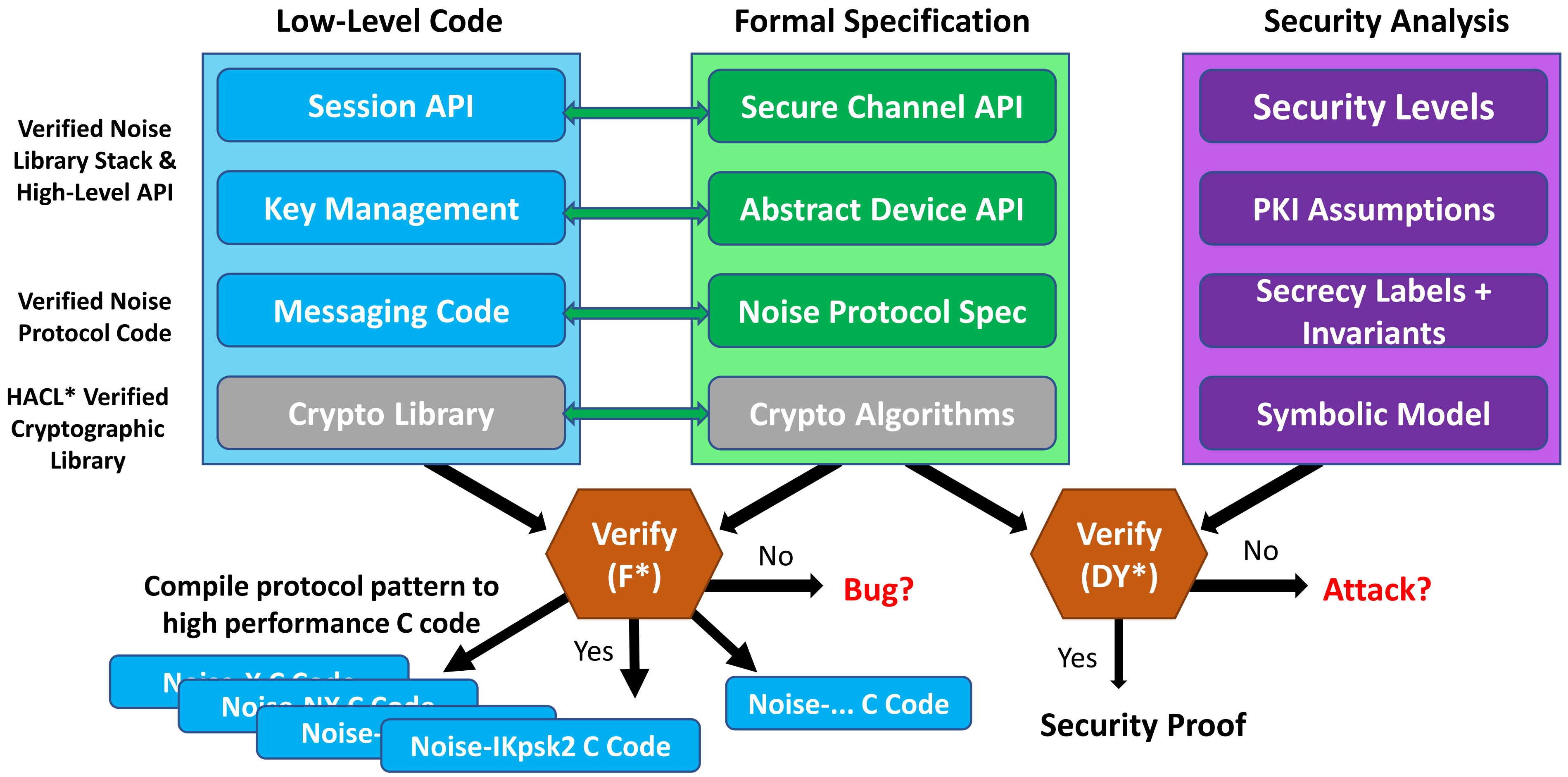
- **Noise* compiler**: Noise protocol “pattern” → verified, specialized C implementation
- On top: complete, verified **library stack** exposed through a **high-level, defensive API**
- Complemented with a formal **symbolic security analysis**



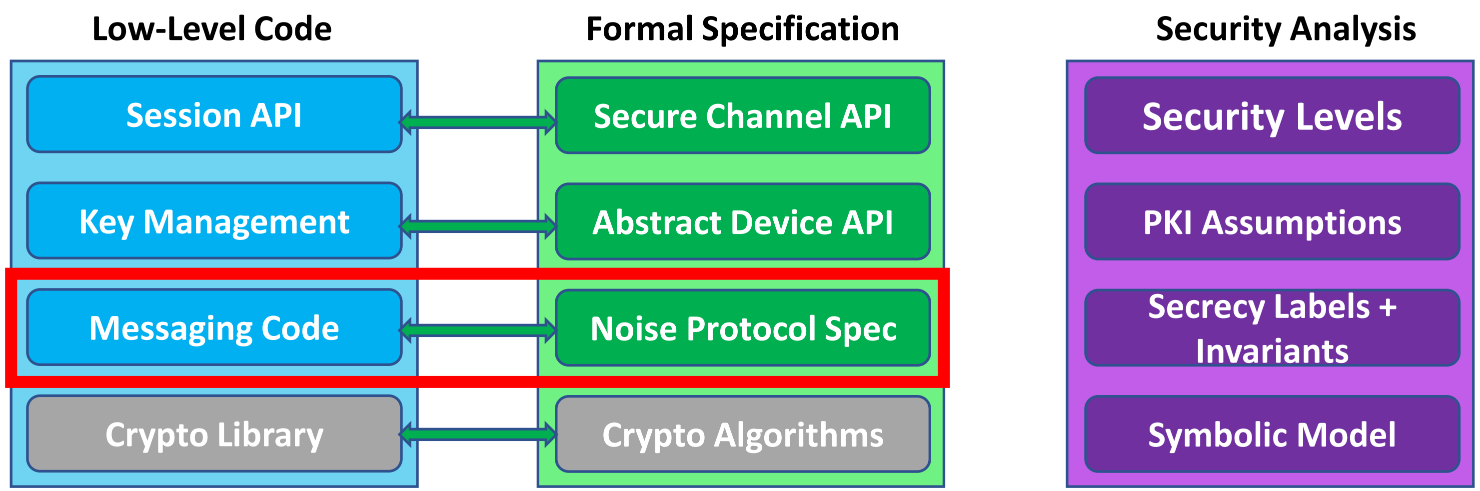
What is **Noise***?

Correctly implemented protocols?

- **Noise* compiler**: Noise protocol “pattern” → verified, specialized C implementation
- On top: complete, verified **library stack** exposed through a **high-level, defensive API**
- Complemented with a formal **symbolic security analysis**



How did we implement the Noise* protocol compiler?



Formal Functional Specification of Noise

noiseprotocol.org:

- `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).

  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. If multiple public keys are listed in either party's pre-message, the public keys are hashed in the order that they are listed.
  - 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:

Formal Functional Specification of Noise

noiseprotocol.org:

- `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).

  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. If multiple public keys are listed in either party's pre-message, the public keys are hashed in the order that they are listed.
  - 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:

F* theorem prover

F* **specification** written as an **interpreter**:

```
// Process a message (without its payload)
let rec send_message_tokens #nc initiator is_psk tokens
  (st : handshake_state) : result (bytes & handshake_state) =
  match tokens with
  | [] -> Res (lbytes_empty, st)
  | tk::tokens1 ->
    // First token
    match send_message_token initiator is_psk tk st with
    | Fail e -> Fail e
    | Res (msg1, st1) ->
      // Remaining tokens
      match send_message_tokens initiator is_psk tokens st1 with
      | Fail e -> Fail e
      | Res (msg2, st2) ->
        Res (msg1 @ msg2, st2)
```

Shallow embedding in Low*

Low*: effectful subset of F* modelling C

```
// Low* signature
val aead_encrypt :
     key:lbuffer uint8 32ul
  -> nonce:lbuffer uint8 12ul
  -> alen:size_t
  -> aad:lbuffer uint8 alen
  -> len:size_t
  -> input:lbuffer uint8 len
  -> output:lbuffer uint8 len
  -> tag:lbuffer uint8 16ul ->

  Stack unit

  (requires (fun h0 ->
    live h0 key /\ live h0 nonce /\ … /\
    disjoint key output /\ disjoint nonce output /\ … ))

  (ensures (fun h0 _ h1 ->
    modifies2 output tag h0 h1 /\
    Seq.append (as_seq h1 output) (as_seq h1 tag) ==
    Spec.aead_encrypt (as_seq h0 key) (as_seq h0 nonce)
                      (as_seq h0 input) (as_seq h0 aad)))


// Low* implementation
let aead_encrypt k n aadlen aad mlen m cipher mac =
  chacha20_encrypt mlen cipher m k n 1ul;
  derive_key_poly1305_do k n aadlen aad mlen cipher mac
```

Shallow embedding in Low*

Low*: effectful subset of F* modelling C

```
// Low* signature
val aead_encrypt :
     key:lbuffer uint8 32ul
  -> nonce:lbuffer uint8 12ul
  -> alen:size_t
  -> aad:lbuffer uint8 alen
  -> len:size_t
  -> input:lbuffer uint8 len
  -> output:lbuffer uint8 len
  -> tag:lbuffer uint8 16ul ->

  Stack unit
```

Formal pre/postconditions

```
  (requires (fun h0 ->
    live h0 key /\ live h0 nonce /\ … /\
    disjoint key output /\ disjoint nonce output /\ … ))

  (ensures (fun h0 _ h1 ->
    modifies2 output tag h0 h1 /\
    Seq.append (as_seq h1 output) (as_seq h1 tag) ==
    Spec.aead_encrypt (as_seq h0 key) (as_seq h0 nonce)
                      (as_seq h0 input) (as_seq h0 aad)))


// Low* implementation
let aead_encrypt k n aadlen aad mlen m cipher mac =
  chacha20_encrypt mlen cipher m k n 1ul;
  derive_key_poly1305_do k n aadlen aad mlen cipher mac
```

Shallow embedding in Low*

Low*: effectful subset of F* modelling C

```
// Low* signature
val aead_encrypt :
     key:lbuffer uint8 32ul
  -> nonce:lbuffer uint8 12ul
  -> alen:size_t
  -> aad:lbuffer uint8 alen
  -> len:size_t
  -> input:lbuffer uint8 len
  -> output:lbuffer uint8 len
  -> tag:lbuffer uint8 16ul ->

  Stack unit
```

Formal pre/postconditions

```
  (requires (fun h0 ->
    live h0 key /\ live h0 nonce /\ … /\
    disjoint key output /\ disjoint nonce output /\ … ))

  (ensures (fun h0 _ h1 ->
    modifies2 output tag h0 h1 /\
    Seq.append (as_seq h1 output) (as_seq h1 tag) ==
    Spec.aead_encrypt (as_seq h0 key) (as_seq h0 nonce)
                      (as_seq h0 input) (as_seq h0 aad)))
```

```
// Low* implementation
let aead_encrypt k n aadlen aad mlen m cipher mac =
  chacha20_encrypt mlen cipher m k n 1ul;
  derive_key_poly1305_do k n aadlen aad mlen cipher mac
```

Proof obligations sent to Z3 SMT solver

Shallow embedding in Low*

Low*: effectful subset of F* modelling C

```
// Low* signature
val aead_encrypt :
     key:lbuffer uint8 32ul
  -> nonce:lbuffer uint8 12ul
  -> alen:size_t
  -> aad:lbuffer uint8 alen
  -> len:size_t
  -> input:lbuffer uint8 len
  -> output:lbuffer uint8 len
  -> tag:lbuffer uint8 16ul ->

  Stack unit
```

Formal pre/postconditions

```
  (requires (fun h0 ->
    live h0 key /\ live h0 nonce /\ … /\
    disjoint key output /\ disjoint nonce output /\ … ))

  (ensures (fun h0 _ h1 ->
    modifies2 output tag h0 h1 /\
    Seq.append (as_seq h1 output) (as_seq h1 tag) ==
    Spec.aead_encrypt (as_seq h0 key) (as_seq h0 nonce)
                      (as_seq h0 input) (as_seq h0 aad)))
```

```
// Low* implementation
let aead_encrypt k n aadlen aad mlen m cipher mac =
  chacha20_encrypt mlen cipher m k n 1ul;
  derive_key_poly1305_do k n aadlen aad mlen cipher mac
```

Proof obligations sent to Z3 SMT solver

KreMLin: Low* → C (erasure, monomorphization…)

```
// Generated C code
void
Hacl_Chacha20Poly1305_32_aead_encrypt(
  uint8_t *k,
  uint8_t *n,
  uint32_t aadlen,
  uint8_t *aad,
  uint32_t mlen,
  uint8_t *m,
  uint8_t *cipher,
  uint8_t *mac
)
{
  Hacl_Chacha20_chacha20_encrypt(mlen, cipher, m, k, n, (uint32_t)1U);
  uint8_t tmp[64U] = { 0U };
  Hacl_Chacha20_chacha20_encrypt((uint32_t)64U, tmp, tmp, k, n, (uint32_t)0U);
  uint8_t *key = tmp;
  poly1305_do_32(key, aadlen, aad, mlen, cipher, mac);
}
```

Shallow embedding in Low*

Low*: effectful subset of F* modelling C

```
// Low* signature
val aead_encrypt :
     key:lbuffer uint8 32ul
  -> nonce:lbuffer uint8 12ul
  -> alen:size_t
  -> aad:lbuffer uint8 alen
  -> len:size_t
  -> input:lbuffer uint8 len
  -> output:lbuffer uint8 len
  -> tag:lbuffer uint8 16ul ->

  Stack unit
```

Formal pre/postconditions

```
  (requires (fun h0 ->
    live h0 key /\ live h0 nonce /\ … /\
    disjoint key output /\ disjoint nonce output /\ … ))

  (ensures (fun h0 _ h1 ->
    modifies2 output tag h0 h1 /\
    Seq.append (as_seq h1 output) (as_seq h1 tag) ==
    Spec.aead_encrypt (as_seq h0 key) (as_seq h0 nonce)
                      (as_seq h0 input) (as_seq h0 aad)))
```

```
// Low* implementation
let aead_encrypt k n aadlen aad mlen m cipher mac =
  chacha20_encrypt mlen cipher m k n 1ul;
  derive_key_poly1305_do k n aadlen aad mlen cipher mac
```

Proof obligations sent to Z3 SMT solver

KreMLin: Low* → C (erasure, monomorphization…)

```
// Generated C code
void
Hacl_Chacha20Poly1305_32_aead_encrypt(
  uint8_t *k,
  uint8_t *n,
  uint32_t aadlen,
  uint8_t *aad,
  uint32_t mlen,
  uint8_t *m,
  uint8_t *cipher,
  uint8_t *mac
)
{
  Hacl_Chacha20_chacha20_encrypt(mlen, cipher, m, k, n, (uint32_t)1U);
  uint8_t tmp[64U] = { 0U };
  Hacl_Chacha20_chacha20_encrypt((uint32_t)64U, tmp, tmp, k, n, (uint32_t)0U);
  uint8_t *key = tmp;
  poly1305_do_32(key, aadlen, aad, mlen, cipher, mac);
}
```

Low* has been successfully used for: cryptographic primitives, protocols (TLS, Signal…), …

⇒ **We can implement an interpreter for Noise in Low***

Target code

Wireguard VPN (IKpsk2):

```
/* First message: e, es, s, ss */
handshake_init(handshake->chaining_key, handshake->hash,
               handshake->remote_static);

/* e */
curve25519_generate_secret(handshake->ephemeral_private);
if (!curve25519_generate_public(dst->unencrypted_ephemeral,
                                handshake->ephemeral_private))
        goto out;
message_ephemeral(dst->unencrypted_ephemeral,
                  dst->unencrypted_ephemeral, handshake->chaining_key,
                  handshake->hash);

/* es */
if (!mix_dh(handshake->chaining_key, key, handshake->ephemeral_private,
            handshake->remote_static))
        goto out;

/* s */
message_encrypt(dst->encrypted_static,
                handshake->static_identity->static_public,
                NOISE_PUBLIC_KEY_LEN, key, handshake->hash);

/* ss */
if (!mix_precomputed_dh(handshake->chaining_key, key,
                        handshake->precomputed_static_static))
        goto out;
```

Our Low* code follows the structure of the below spec.:

```
let rec send_message_tokens #nc initiator is_psk tokens st =
  match tokens with
  | [] -> Res (lbytes_empty, st)
  | tk::tokens1 ->
    // First token
    match send_message_token initiator is_psk tk st with
    | Fail e -> Fail e
    | Res (msg1, st1) ->
      // Remaining tokens
      match send_message_tokens initiator is_psk tokens st1 with
      | Fail e -> Fail e
      | Success (msg2, st2) ->
        Res (msg1 @ msg2, st2)
```

Target code

Wireguard VPN (IKpsk2):

```
/* First message: e, es, s, ss */
handshake_init(handshake->chaining_key, handshake->hash,
               handshake->remote_static);

/* e */
curve25519_generate_secret(handshake->ephemeral_private);
if (!curve25519_generate_public(dst->unencrypted_ephemeral,
                                 handshake->ephemeral_private))
        goto out;
message_ephemeral(dst->unencrypted_ephemeral,
                  dst->unencrypted_ephemeral, handshake->chaining_key,
                  handshake->hash);

/* es */
if (!mix_dh(handshake->chaining_key, key, handshake->ephemeral_private,
            handshake->remote_static))
        goto out;

/* s */
message_encrypt(dst->encrypted_static,
                handshake->static_identity->static_public,
                NOISE_PUBLIC_KEY_LEN, key, handshake->hash);

/* ss */
if (!mix_precomputed_dh(handshake->chaining_key, key,
                        handshake->precomputed_static_static))
        goto out;
```

Our Low* code follows the structure of the below spec.:

```
let rec send_message_tokens #nc initiator is_psk tokens st =
  match tokens with
  | [] -> Res (lbytes_empty, st)
  | tk::tokens1 ->
    // First token
    match send_message_token initiator is_psk tk st with
    | Fail e -> Fail e
    | Res (msg1, st1) ->
      // Remaining tokens
      match send_message_tokens initiator is_psk tokens st1 with
      | Fail e -> Fail e
      | Success (msg2, st2) ->
        Res (msg1 @ msg2, st2)
```

Specialized, idiomatic C code: no recursion, no token lists, etc.

Target code

Wireguard VPN (IKpsk2):

```
/* First message: e, es, s, ss */
handshake_init(handshake->chaining_key, handshake->hash,
               handshake->remote_static);

/* e */
curve25519_generate_secret(handshake->ephemeral_private);
if (!curve25519_generate_public(dst->unencrypted_ephemeral,
                                handshake->ephemeral_private))
        goto out;
message_ephemeral(dst->unencrypted_ephemeral,
                  dst->unencrypted_ephemeral, handshake->chaining_key,
                  handshake->hash);

/* es */
if (!mix_dh(handshake->chaining_key, key, handshake->ephemeral_private,
            handshake->remote_static))
        goto out;

/* s */
message_encrypt(dst->encrypted_static,
                handshake->static_identity->static_public,
                NOISE_PUBLIC_KEY_LEN, key, handshake->hash);

/* ss */
if (!mix_precomputed_dh(handshake->chaining_key, key,
                        handshake->precomputed_static_static))
        goto out;
```

Our Low* code follows the structure of the below spec.:

```
let rec send_message_tokens #nc initiator is_psk tokens st =
  match tokens with
  | [] -> Res (lbytes_empty, st)
  | tk::tokens1 ->
    // First token
    match send_message_token initiator is_psk tk st with
    | Fail e -> Fail e
    | Res (msg1, st1) ->
      // Remaining tokens
      match send_message_tokens initiator is_psk tokens st1 with
      | Fail e -> Fail e
      | Success (msg2, st2) ->
        Res (msg1 @ msg2, st2)
```

Specialized, idiomatic C code: no recursion, no token lists, etc.

How to specialize an interpreter for a given input?
How to turn an interpreter into a compiler?

Hybrid Embeddings

Idea: use F* to meta-program as much as possible:

- Similar to super advanced **C++ templates**
- Write a meta-program once, specialize N times (⇒ 59 patterns)

Hybrid Embeddings

Idea: use F* to meta-program as much as possible:

- Similar to super advanced **C++ templates**
- Write a meta-program once, specialize N times (⇒ 59 patterns)

Historically, long arc of meta-programming in F*:

- EverCrypt agility (2020)
- HACLxN vector types (2021)
- Streaming Functor (2021)
- EverParse (Ongoing)

Hybrid Embeddings

Idea: use F* to meta-program as much as possible:

- Similar to super advanced **C++ templates**
- Write a meta-program once, specialize N times (⇒ 59 patterns)

Historically, long arc of meta-programming in F*:

- EverCrypt agility (2020)
- HACLxN vector types (2021)
- Streaming Functor (2021)
- EverParse (Ongoing)

➡ With **Noise***: complete, meta-programmed protocol stack

Hybrid Embeddings

```
let send_IKpsk2_message0 (st : handshake_state) =
  send_message_tokens true true [E; ES; S; SS] st
```

Hybrid Embeddings

```
let send_IKpsk2_message0 (st : handshake_state) =
  send_message_tokens true true [E; ES; S; SS] st
```

Hybrid Embeddings

```
let send_IKpsk2_message0 (st : handshake_state) =
  match [E; ES; S; SS] with
  | [] -> Res (empty, st)
  | tk :: tokens1 ->
    match send_message_token true true tk st with
    | Fail e -> Fail e
    | Res (msg1, st1) ->
      match send_message_tokens true true tokens1 st1 with
      | Fail e -> Fail e
      | Res (msg2, st2) ->
        Res (msg1 @ msg2, st2)
```

Hybrid Embeddings

```
let send_IKpsk2_message0 (st : handshake_state) =
  match send_message_token true true E st with
  | Fail e -> Fail e
  | Res (msg1, st1) ->
    match send_message_tokens true true [ES; S; SS] st1 with
    | Fail e -> Fail e
    | Res (msg2, st2) ->
      Res (msg1 @ msg2, st2)
```

Hybrid Embeddings

```
let send_IKpsk2_message0 (st : handshake_state) =
  match send_message_token true true E st with
  | Fail e -> Fail e
  | Res (msg1, st1) ->
    match send_message_token true true ES st1 with
    | Fail e -> Fail e
    | Res (msg2, st2) ->
      match send_message_token true true S st2 with
      | Fail e -> Fail e
      | Res (msg3, st3) ->
        match send_message_token true true SS st3 with
        | Fail e -> Fail e
        | Res (msg4, st4) ->
          Res (msg1 @ msg2 @ msg3 @ msg4, st4)
```

Hybrid Embeddings

```
let send_IKpsk2_message0 (st : handshake_state) =
  match // E
    begin match st.ephemeral with
    | None -> Fail No_key
    | Some e ->
      let sym_st1 = mix_hash k.pub st.sym_state in
      let sym_st2 =
      if true // This is `is_psk`
      then mix_key e.pub sym_st1
      else sym_st1
      in
      let st1 = { st with sym_state = sym_st2; } in
      let msg1 = e.pub in
      Res (msg1, st1)
    end
  with
  | Fail e -> Fail e
  | Res (msg1, st1) -> // Other tokens:
    match send_message_token true true ES st1 with
    | Fail e -> Fail e
    | Res (msg2, st2) ->
      match send_message_token true true S st2 with
      | Fail e -> Fail e
      | Res (msg3, st3) ->
        …
```

Hybrid Embeddings

```
let send_IKpsk2_message0 (st : handshake_state) =
  match // E
    begin match st.ephemeral with
    | None -> Fail No_key // Unreachable if proper precondition
    | Some e ->
      let sym_st1 = mix_hash k.pub st.sym_state in
      let sym_st2 = mix_key e.pub sym_st1 in
      let st1 = { st with sym_state = sym_st2; } in
      let msg1 = e.pub in
      Res (msg1, st1)
    end
  with
  | Fail e -> Fail e // Unreachable if proper precondition
  | Res (msg1, st1) -> // Other tokens:
    match send_message_token true true ES st1 with
    | Fail e -> Fail e
    | Res (msg2, st2) ->
      match send_message_token true true S st2 with
      | Fail e -> Fail e
      | Res (msg3, st3) ->
        match send_message_token true true S st2 with
        | Fail e -> Fail e
        | Res (msg4, st4) ->
          Res (msg1 @ msg2 @ msg3 @ msg4, st3)
```

Hybrid Embeddings

```
let send_IKpsk2_message0 (st : handshake_state) =
  match // E
    begin match st.ephemeral with
    | None -> Fail No_key // Unreachable if proper precondition
    | Some e ->
      let sym_st1 = mix_hash k.pub st.sym_state in
      let sym_st2 = mix_key e.pub sym_st1 in
      let st1 = { st with sym_state = sym_st2; } in
      let msg1 = e.pub in
      Res (msg1, st1)
    end
  with
  | Fail e -> Fail e // Unreachable if proper precondition
  | Res (msg1, st1) -> // Other tokens:
    match send_message_token true true ES st1 with
    | Fail e -> Fail e
    | Res (msg2, st2) ->
      match send_message_token true true S st2 with
      | Fail e -> Fail e
      | Res (msg3, st3) ->
        match send_message_token true true S st2 with
        | Fail e -> Fail e
        | Res (msg4, st4) ->
          Res (msg1 @ msg2 @ msg3 @ msg4, st3)
```

Embeddings in Low* are **shallow**: partial reduction applies!

```
// Simplified
let rec send_message_tokens_m =
  fun smi initiator is_psk tokens st outlen out ->
  match tokens with
  | Nil -> success _
  | tk :: tokens' ->
    let tk_outlen = token_message_vs nc smi tk in
    let tk_out = sub out 0ul tk_outlen in
    let r1 = send_message_token_m smi initiator ... In
    ...
```

⇒ Compilation through **staging**: first step with F* normalizer

Hybrid Embeddings

```
let send_IKpsk2_message0 (st : handshake_state) =
  match // E
    begin match st.ephemeral with
    | None -> Fail No_key // Unreachable if proper precondition
    | Some e ->
      let sym_st1 = mix_hash k.pub st.sym_state in
      let sym_st2 = mix_key e.pub sym_st1 in
      let st1 = { st with sym_state = sym_st2; } in
      let msg1 = e.pub in
      Res (msg1, st1)
    end
  with
  | Fail e -> Fail e // Unreachable if proper precondit
  | Res (msg1, st1) -> // Other tokens:
    match send_message_token true true ES st1 with
    | Fail e -> Fail e
    | Res (msg2, st2) ->
      match send_message_token true true S st2 with
      | Fail e -> Fail e
      | Res (msg3, st3) ->
        match send_message_token true true S st2 with
        | Fail e -> Fail e
        | Res (msg4, st4) ->
          Res (msg1 @ msg2 @ msg3 @ msg4, st3)
```

Embeddings in Low* are **shallow**: partial reduction applies!

```
// Simplified
let rec send_message_tokens_m =
  fun smi initiator is_psk tokens st outlen out ->
  match tokens with
  | Nil -> success _
  | tk :: tokens' ->
    let tk_outlen = token_message_vs nc smi tk in
    let tk_out = sub out 0ul tk_outlen in
    let r1 = send_message_token_m smi initiator ... In
    ...
```

⇒ Compilation through **staging**: first step with F* normalizer

E disappeared!
⇒ "**meta**" parameters (and computations) vs "runtime" parameters (and computations)

Tweaking Control-Flow and Types

```
// Spec
match send_message_token true true S st with
| Fail e -> Fail e
| Res (msg, st') -> ...
```

```
// Low*
let r : error_code = send_message_token ... S ... in
if r = Success then
  ... // "if" branch
else
  ... // "else" branch
```

Tweaking Control-Flow and Types

```
// Spec
match send_message_token true true S st with
| Fail e -> Fail e
| Res (msg, st') -> ...
```

Unreachable!

```
// Low*
let r : error_code = send_message_token ... S ... in
if r = Success then
  ... // "if" branch
else
  ... // "else" branch
```

Always succeeds!

Tweaking Control-Flow and Types

```
// Spec
match send_message_token true true S st with
| Fail e -> Fail e
| Res (msg, st’) -> ...
```

Unreachable!

```
// Low*
let r : error_code = send_message_token ... S ... in
if r = Success then
  ... // “if” branch
else
  ... // “else” branch
```

Always succeeds!

F* has dependent types!

```
type error_code_or_unit (b : bool) =
  if b then error_code else unit

let is_success (b : bool) (r : error_code_or_unit b) :
  bool =
  if b then r = Success else true
```

Tweaking Control-Flow and Types

```
// Spec
match send_message_token true true S st with
| Fail e -> Fail e
| Res (msg, st') -> ...
```

Unreachable!

```
// Low*
let r : error_code = send_message_token ... S ... in
if r = Success then
  ... // "if" branch
else
  ... // "else" branch
```

Always succeeds!

F* has dependent types!

```
type error_code_or_unit (b : bool) =
  if b then error_code else unit

let is_success (b : bool) (r : error_code_or_unit b) :
  bool =
  if b then r = Success else true
```

```
let can_fail (tk : token) : bool =
  match tk with
  | S -> false
  | ...
```

Tweaking Control-Flow and Types

```
// Spec
match send_message_token true true S st with
| Fail e -> Fail e
| Res (msg, st’) -> ...
```

Unreachable!

```
// Low*
let r : error_code_or_unit (can_fail S) = send_message_token ... S ... in
if is_success (can_fail S) r then
  ... // “if” branch
else
  ... // “else” branch
```

F* has dependent types!

```
type error_code_or_unit (b : bool) =
  if b then error_code else unit

let is_success (b : bool) (r : error_code_or_unit b) :
  bool =
  if b then r = Success else true
```

```
let can_fail (tk : token) : bool =
  match tk with
  | S -> false
  | ...
```

Tweaking Control-Flow and Types

```
// Spec
match send_message_token true true S st with
| Fail e -> Fail e
| Res (msg, st') -> ...
```

Unreachable!

```
// Low*
let r : error_code_or_unit (can_fail S) = send_message_token ... S ... in
if is_success (can_fail S) r then
  ... // "if" branch
else
  ... // "else" branch
```

After partial reduction

```
// Low*
let r : error_code_or_unit false = send_message_token ... S ... in
if is_success #false r then
  ... // "if" branch
else
  ... // "else" branch
```

F* has dependent types!

```
type error_code_or_unit (b : bool) =
  if b then error_code else unit

let is_success (b : bool) (r : error_code_or_unit b) :
  bool =
  if b then r = Success else true
```

```
let can_fail (tk : token) : bool =
  match tk with
  | S -> false
  | ...
```

Tweaking Control-Flow and Types

```
// Spec
match send_message_token true true S st with
| Fail e -> Fail e
| Res (msg, st') -> ...
```

Unreachable!

```
// Low*
let r : error_code_or_unit (can_fail S) = send_message_token ... S ... in
if is_success (can_fail S) r then
  ... // "if" branch
else
  ... // "else" branch
```

After partial reduction

```
// Low*
let r : unit = send_message_token ... S ... in
if is_success #false r then
  ... // "if" branch
else
  ... // "else" branch
```

F* has dependent types!

```
type error_code_or_unit (b : bool) =
  if b then error_code else unit

let is_success (b : bool) (r : error_code_or_unit b) :
  bool =
  if b then r = Success else true
```

```
let can_fail (tk : token) : bool =
  match tk with
  | S -> false
  | ...
```

Tweaking Control-Flow and Types

```
// Spec
match send_message_token true true S st with
| Fail e -> Fail e
| Res (msg, st') -> ...
```

Unreachable!

```
// Low*
let r : error_code_or_unit (can_fail S) = send_message_token ... S ... in
if is_success (can_fail S) r then
  ... // "if" branch
else
  ... // "else" branch
```

After partial reduction

```
// Low*
let r : unit = send_message_token ... S ... in
if true then
  ... // "if" branch
else
  ... // "else" branch
```

F* has dependent types!

```
type error_code_or_unit (b : bool) =
  if b then error_code else unit

let is_success (b : bool) (r : error_code_or_unit b) :
  bool =
  if b then r = Success else true
```

```
let can_fail (tk : token) : bool =
  match tk with
  | S -> false
  | ...
```

Tweaking Control-Flow and Types

```
// Spec
match send_message_token true true S st with
| Fail e -> Fail e
| Res (msg, st') -> ...
```

Unreachable!

```
// Low*
let r : error_code_or_unit (can_fail S) = send_message_token ... S ... in
if is_success (can_fail S) r then
  ... // "if" branch
else
  ... // "else" branch
```

After partial reduction

```
// Low*
let r : unit = send_message_token ... S ... in
... // "if" branch
```

F* has dependent types!

```
type error_code_or_unit (b : bool) =
  if b then error_code else unit

let is_success (b : bool) (r : error_code_or_unit b) :
  bool =
  if b then r = Success else true
```

```
let can_fail (tk : token) : bool =
  match tk with
  | S -> false
  | ...
```

Tweaking Control-Flow and Types

```
// Spec
match send_message_token true true S st with
| Fail e -> Fail e
| Res (msg, st') -> ...
```

Unreachable!

```
// Low*
let r : error_code_or_unit (can_fail S) = send_message_token ... S ... in
if is_success (can_fail S) r then
  ... // "if" branch
else
  ... // "else" branch
```

After partial reduction

```
// Low*
let r : unit = send_message_token ... S ... in
... // "if" branch
```

F* has dependent types!

```
type error_code_or_unit (b : bool) =
  if b then error_code else unit

let is_success (b : bool) (r : error_code_or_unit b) :
  bool =
  if b then r = Success else true
```

```
let can_fail (tk : token) : bool =
  match tk with
  | S -> false
  | ...
```

Write **general dependent types** which reduce to **precise non-dependent types**:

- Drastically improve code quality (make it smaller, more readable, more idiomatic)

Tweaking Control-Flow and Types

```
// Spec
match send_message_token true true S st with
| Fail e -> Fail e
| Res (msg, st') -> ...
```

Unreachable!

```
// Low*
let r : error_code_or_unit (can_fail S) = send_message_token ... S ... in
if is_success (can_fail S) r then
  ... // "if" branch
else
  ... // "else" branch
```

After partial reduction

```
// Low*
let r : unit = send_message_token ... S ... in
... // "if" branch
```

F* has dependent types!

```
type error_code_or_unit (b : bool) =
  if b then error_code else unit

let is_success (b : bool) (r : error_code_or_unit b) :
  bool =
  if b then r = Success else true
```

```
let can_fail (tk : token) : bool =
  match tk with
  | S -> false
  | ...
```

Write **general dependent types** which reduce to **precise non-dependent types**:

- Drastically improve code quality (make it smaller, more readable, more idiomatic)
- Make extracted types (structures, etc.) more precise

Tweaking Control-Flow and Types

```
// Spec
match send_message_token true true S st with
| Fail e -> Fail e
| Res (msg, st') -> ...
```

Unreachable!

```
// Low*
let r : error_code_or_unit (can_fail S) = send_message_token ... S ... in
if is_success (can_fail S) r then
  ... // "if" branch
else
  ... // "else" branch
```

After partial reduction

```
// Low*
let r : unit = send_message_token ... S ... in
... // "if" branch
```

F* has dependent types!

```
type error_code_or_unit (b : bool) =
  if b then error_code else unit

let is_success (b : bool) (r : error_code_or_unit b) :
  bool =
  if b then r = Success else true
```

```
let can_fail (tk : token) : bool =
  match tk with
  | S -> false
  | ...
```

Write **general dependent types** which reduce to **precise non-dependent types**:

- Drastically improve code quality (make it smaller, more readable, more idiomatic)
- Make extracted types (structures, etc.) more precise
- Make **function signatures** more informative (**unit elimination**)

```
val f (x : uint32_t) (y : unit) : unit // Low*
void f (x : uint32_t); // Generated C
```

Tweaking Control-Flow and Types

```
// Spec
match send_message_token true true S st with
| Fail e -> Fail e
| Res (msg, st') -> ...
```

Unreachable!

```
// Low*
let r : error_code_or_unit (can_fail S) = send_message_token ... S ... in
if is_success (can_fail S) r then
  ... // "if" branch
else
  ... // "else" branch
```

After partial reduction

```
// Low*
let r : unit = send_message_token ... S ... in
... // "if" branch
```

F* has dependent types!

```
type error_code_or_unit (b : bool) =
  if b then error_code else unit

let is_success (b : bool) (r : error_code_or_unit b) :
  bool =
  if b then r = Success else true
```

```
let can_fail (tk : token) : bool =
  match tk with
  | S -> false
  | ...
```

Write **general dependent types** which reduce to **precise non-dependent types**:

- Drastically improve code quality (make it smaller, more readable, more idiomatic)
- Make extracted types (structures, etc.) more precise
- Make **function signatures** more informative (**unit elimination**)

```
val f (x : uint32_t) (y : unit) : unit // Low*
void f (x : uint32_t); // Generated C
```

- **We don't have to choose between genericity and efficiency**

Generated Code (IKpsk2)

Noise*

```
/* e */
Impl_Noise_Instances_mix_hash(ms_h, (uint32_t)32U, mepub);
Impl_Noise_Instances_kdf(ms_ck, (uint32_t)32U, mepub, ms_ck, mc_state, NULL);
memcpy(tk_out, mepub, (uint32_t)32U * sizeof (uint8_t));
/* es */
uint8_t *out_ = pat_out + (uint32_t)32U;
Impl_Noise_Types_error_code
r11 = Impl_Noise_Instances_mix_dh(mepriv, mremote_static, mc_state, ms_ck, ms_h);
Impl_Noise_Types_error_code r2;
if (r11 == Impl_Noise_Types_CSuccess)
{
  /* s */
  uint8_t *out_1 = out_;
  uint8_t *tk_out2 = out_1;
  Impl_Noise_Instances_encrypt_and_hash((uint32_t)32U,
    mspub,
    tk_out2,
    mc_state,
    ms_h,
    (uint64_t)0U);
  /* ss */
  Impl_Noise_Types_error_code
  r = Impl_Noise_Instances_mix_dh(mspriv, mremote_static, mc_state, ms_ck, ms_h);
  Impl_Noise_Types_error_code r20 = r;
  Impl_Noise_Types_error_code r21 = r20;
  r2 = r21;
}
else
  r2 = r11;
```

Wireguard VPN (for reference):

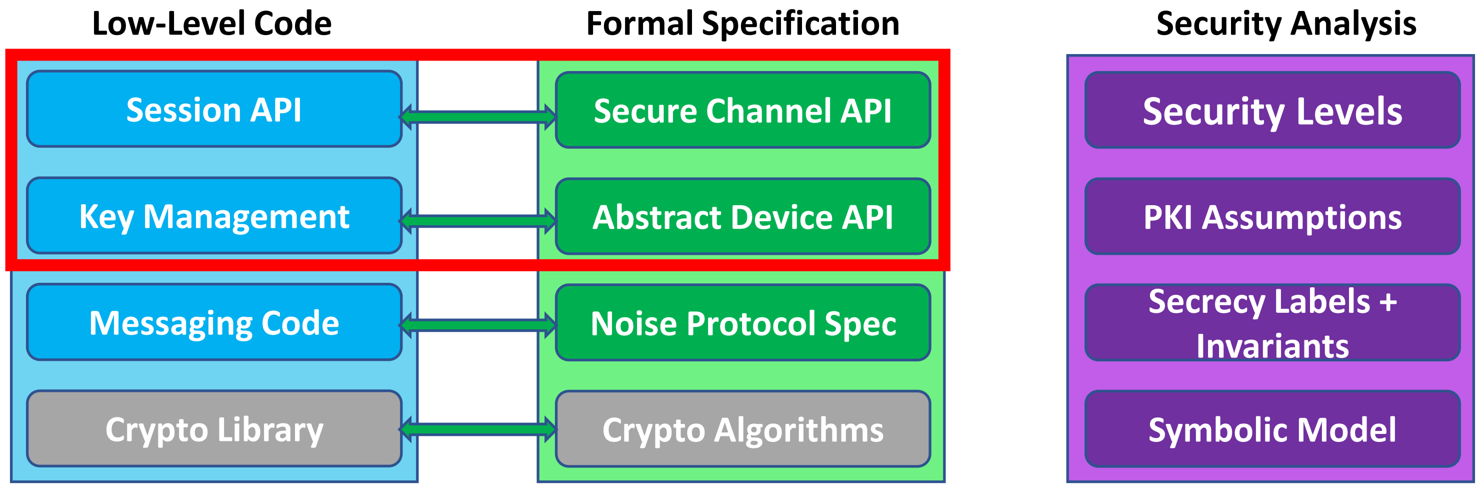
```
/* e */
curve25519_generate_secret(handshake->ephemeral_private);
if (!curve25519_generate_public(dst->unencrypted_ephemeral,
                                handshake->ephemeral_private))
        goto out;
message_ephemeral(dst->unencrypted_ephemeral,
                  dst->unencrypted_ephemeral, handshake->chaining_key,
                  handshake->hash);

/* es */
if (!mix_dh(handshake->chaining_key, key, handshake->ephemeral_private,
            handshake->remote_static))
        goto out;

/* s */
message_encrypt(dst->encrypted_static,
                handshake->static_identity->static_public,
                NOISE_PUBLIC_KEY_LEN, key, handshake->hash);

/* ss */
if (!mix_precomputed_dh(handshake->chaining_key, key,
                        handshake->precomputed_static_static))
        goto out;
```

What does the high-level API give us?



- State Machines
- Peer Management
- Key Storage & Validation
- Message Encapsulation

High-Level API

IKpsk2:

← s

. . .

→ e, es, s, ss, [d0]

← e, ee, se, psk, [d1]

↔ [d2, d3, …]

- Initiator and responder must remember which key belongs to whom
- Responder receives a static key during the handshake
  - Peer lookup (if key already registered)
  - Unknown key validation
- Long-term key storage
- Transitions are low-level
  - State Machine
  - Message lengths
  - Invalid states (if failure)
- Early data
  - when is it safe to send secret data?
  - when can we trust the data we received?

High-Level API

IKpsk2:

← s

. . .

→ e, es, s, ss, [d0]

← e, ee, se, psk, [d1]

↔ [d2, d3, …]

- Initiator and responder must remember which key belongs to whom
- Responder receives a static key during the handshake
  - Peer lookup (if key already registered)
  - Unknown key validation
- Long-term key storage
- Transitions are low-level
  - State Machine
  - Message lengths
  - Invalid states (if failure)
- Early data
  - when is it safe to send secret data?
  - when can we trust the data we received?

Peer Management

High-Level API

IKpsk2:

← s

. . .

→ e, es, s, ss, [d0]

← e, ee, se, psk, [d1]

↔ [d2, d3, …]

- Initiator and responder must remember which key belongs to whom
- Responder receives a static key during the handshake
  - Peer lookup (if key already registered)
  - Unknown key validation
- Long-term key storage
- Transitions are low-level
  - State Machine
  - Message lengths
  - Invalid states (if failure)
- Early data
  - when is it safe to send secret data?
  - when can we trust the data we received?

Peer Management

Key Validation

High-Level API

IKpsk2:

← s

. . .

→ e, es, s, ss, [d0]

← e, ee, se, psk, [d1]

↔ [d2, d3, …]

- Initiator and responder must remember which key belongs to whom
- Responder receives a static key during the handshake
  - Peer lookup (if key already registered)
  - Unknown key validation
- ➡ Long-term key storage
- Transitions are low-level
  - State Machine
  - Message lengths
  - Invalid states (if failure)
- Early data
  - when is it safe to send secret data?
  - when can we trust the data we received?

Peer Management

Key Validation

Key Storage

High-Level API

IKpsk2:

← s

. . .

→ e, es, s, ss, [d0]

← e, ee, se, psk, [d1]

↔ [d2, d3, …]

- Initiator and responder must remember which key belongs to whom
- Responder receives a static key during the handshake
  - Peer lookup (if key already registered)
  - Unknown key validation
- Long-term key storage
- Transitions are low-level
  - State Machine
  - Message lengths
  - Invalid states (if failure)
- Early data
  - when is it safe to send secret data?
  - when can we trust the data we received?

Peer Management

Key Validation

Key Storage

State Machine

High-Level API

IKpsk2:

← s

. . .

→ e, es, s, ss, [d0]

← e, ee, se, psk, [d1]

↔ [d2, d3, …]

- Initiator and responder must remember which key belongs to whom
- Responder receives a static key during the handshake
  - Peer lookup (if key already registered)
  - Unknown key validation
- Long-term key storage
- Transitions are low-level
  - State Machine
  - Message lengths
  - Invalid states (if failure)
- Early data
  - when is it safe to send secret data?
  - when can we trust the data we received?

Peer Management

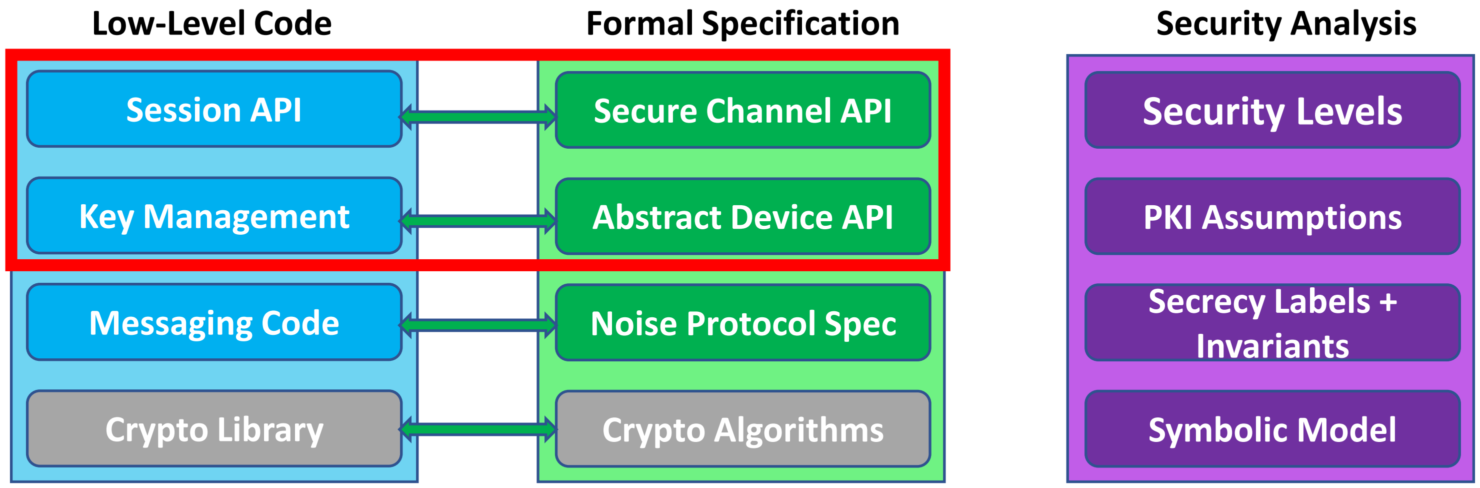
Key Validation

Key Storage

State Machine

Message Encapsulation

What does the high-level API give us?



- State Machines
- Peer Management
- Key Storage & Validation
- Message Encapsulation

Meta-Programmed State Machine

With 3 messages (ex.: XX):

```
//
error_code handshake_send(..., uint step, ...) {
  if (step == 0)
    return send_message0(...);
  else if (step == 1)
    return send_message1(...);
  else if (step == 2)
    return send_message2(...);
  ...
}
```

Meta-Programmed State Machine

With 3 messages (ex.: XX):

```
//
error_code handshake_send(..., uint step, ...) {
  if (step == 0)
    return send_message0(...);
  else if (step == 1)
    return send_message1(...);
  else if (step == 2)
    return send_message2(...);
  else
    ... // Unreachable!!
}
```

Meta-Programmed State Machine

With 3 messages (ex.: XX):

```
// With precondition: step <= 2
error_code handshake_send(..., uint step, ...) {
  if (step == 0)
    return send_message0(...);
  else if (step == 1)
    return send_message1(...);
  else // No check - step == 2
    return send_message2(...);
}
```

Meta-Programmed State Machine

With 3 messages (ex.: XX):

```
// With precondition: step <= 2
error_code handshake_send(..., uint step, ...) {
  if (step == 0)
    return send_message0(...); // initiator state
  else if (step == 1)
    return send_message1(...); // responder state!
  else // No check - step == 2
    return send_message2(...); // initiator state
}
```

`state` is a **dependent type**, reduced and monomorphized at extraction time!

Meta-Programmed State Machine

With 3 messages (ex.: XX):

```
// With precondition: step <= 2 /\ (step % 2) == 0
error_code initiator_handshake_send(..., uint step, ..., initiator_state st) {
  if (step == 0) {
    return send_message0(...);
  else // No check - step == 2
    return send_message2(...);
}
```

```
// With precondition: step <= 2 /\ (step % 2) == 1
error_code responder_handshake_send(..., uint step, ..., responder_state st) {
  return send_message1(...);
}
```

`state` is a **dependent type**, reduced and monomorphized at extraction time!

Meta-Programmed State Machine

With 3 messages (ex.: XX):

```
// With precondition: step <= 2 /\ (step % 2) == 0
error_code initiator_handshake_send(..., uint step, ..., initiator_state st) {
  if (step == 0) {
    return send_message0(...);
  else // No check - step == 2
    return send_message2(...);
}
```

```
// With precondition: step <= 2 /\ (step % 2) == 1
error_code responder_handshake_send(..., uint step, ..., responder_state st) {
  return send_message1(...);
}
```

`state` is a **dependent type**, reduced and monomorphized at extraction time!

```
// Generated from an F* inductive
struct state {
  int tag;
  union {
    struct initiator_state;
    struct responder_state;
  } val;
}
```

```
// Top-level `handshake_send` function
error_code handshake_send(..., uint step, ..., state* st) =
   // Match and call the proper function
   ...
}
```

Meta-Programmed State Machine

With 3 messages (ex.: XX):

```
// With precondition: step <= 2 /\ (step % 2) == 0
error_code initiator_handshake_send(..., uint step, ..., initiator_state st) {
  if (step == 0) {
    return send_message0(...);
  else // No check - step == 2
    return send_message2(...);
}
```

```
// With precondition: step <= 2 /\ (step % 2) == 1
error_code responder_handshake_send(..., uint step, ..., responder_state st) {
  return send_message1(...);
}
```

`state` is a **dependent type**, reduced and monomorphized at extraction time!

Actually inlined: structs passed by value

```
// Generated from an F* inductive
struct state {
  int tag;
  union {
    struct initiator_state;
    struct responder_state;
  } val;
}
```

```
// Top-level `handshake_send` function
error_code handshake_send(..., uint step, ..., state* st) =
   // Match and call the proper function
   ...
}
```

Meta Programmed State Machine(s)

We program the 2 state machines (initiator/responder) at once:

Target C code:

```
error_code initiator_handshake_send(...) {
  if (step == 0) {
    return send_message0(...);
  else
    return send_message2(...);
}

error_code responder_handshake_send(...) {
  return send_message1(...);
}
```

F* code:

```
// Pre: initiator==((i%2)==0) /\ i < num_handshake_messages
let rec handshake_send_i (initiator:bool) ... (i:nat) (step:UInt32.t) =
  if i+2 >= num_handshake_messages then
    ... // last possible send_message function
  else if step = size i then
    ... // instantiated send_message function
  else
    handshake_send ... (i+2) step // Increment i by 2
```

Meta Programmed State Machine(s)

We program the 2 state machines (initiator/responder) at once:

Target C code:

```
error_code initiator_handshake_send(...) {
  if (step == 0) {
    return send_message0(...);
  else
    return send_message2(...);
}

error_code responder_handshake_send(...) {
  return send_message1(...);
}
```

F* code:

Meta parameter
(i ∈ {0, 1, …})

```
// Pre: initiator==((i%2)==0) /\ i < num_handshake_messages
let rec handshake_send_i (initiator:bool) ... (i:nat) (step:UInt32.t) =
  if i+2 >= num_handshake_messages then
    ... // last possible send_message function
  else if step = size i then
    ... // instantiated send_message function
  else
    handshake_send ... (i+2) step // Increment i by 2
```

Meta Programmed State Machine(s)

We program the 2 state machines (initiator/responder) at once:

Target C code:

```
error_code initiator_handshake_send(...) {
  if (step == 0) {
    return send_message0(...);
  else
    return send_message2(...);
}

error_code responder_handshake_send(...) {
  return send_message1(...);
}
```

F* code:

Meta parameter ($i \in \{0, 1, \ldots\}$)

Runtime parameter

```
// Pre: initiator==((i%2)==0) /\ i < num_handshake_messages
let rec handshake_send_i (initiator:bool) ... (i:nat) (step:UInt32.t) =
  if i+2 >= num_handshake_messages then
    ... // last possible send_message function
  else if step = size i then
    ... // instantiated send_message function
  else
    handshake_send ... (i+2) step // Increment i by 2
```

Meta Programmed State Machine(s)

We program the 2 state machines (initiator/responder) at once:

Target C code:

```
error_code initiator_handshake_send(...) {
  if (step == 0) {
    return send_message0(...);
  else
    return send_message2(...);
}

error_code responder_handshake_send(...) {
  return send_message1(...);
}
```

F* code:

Meta parameter (i ∈ {0, 1, …})

Runtime parameter

```
// Pre: initiator==((i%2)==0) /\ i < num_handshake_messages
let rec handshake_send_i (initiator:bool) ... (i:nat) (step:UInt32.t) =
  if i+2 >= num_handshake_messages then
    ... // last possible send_message function
  else if step = size i then
    ... // instantiated send_message function
  else
    handshake_send ... (i+2) step // Increment i by 2
```

Meta Programmed State Machine(s)

We program the 2 state machines (initiator/responder) at once:

Target C code:

```
error_code initiator_handshake_send(...) {
  if (step == 0) {
    return send_message0(...);
  else
    return send_message2(...);
}

error_code responder_handshake_send(...) {
  return send_message1(...);
}
```

F* code:

Meta parameter (i ∈ {0, 1, …})

Runtime parameter

```
// Pre: initiator==((i%2)==0) /\ i < num_handshake_messages
let rec handshake_send_i (initiator:bool) ... (i:nat) (step:UInt32.t) =
  if i+2 >= num_handshake_messages then
    ... // last possible send_message function
  else if step = size i then
    ... // instantiated send_message function
  else
    handshake_send ... (i+2) step // Increment i by 2
```

Meta Programmed State Machine(s)

We program the 2 state machines (initiator/responder) at once:

Target C code:

```
error_code initiator_handshake_send(...) {
  if (step == 0) {
    return send_message0(...);
  else
    return send_message2(...);
}

error_code responder_handshake_send(...) {
  return send_message1(...);
}
```

F* code:

Meta parameter (i ∈ {0, 1, …})

Runtime parameter

```
// Pre: initiator==((i%2)==0) /\ i < num_handshake_messages
let rec handshake_send_i (initiator:bool) ... (i:nat) (step:UInt32.t) =
  if i+2 >= num_handshake_messages then
    ... // last possible send_message function
  else if step = size i then
    ... // instantiated send_message function
  else
    handshake_send ... (i+2) step // Increment i by 2
```

Meta Programmed State Machine(s)

We program the 2 state machines (initiator/responder) at once:

Target C code:

```
error_code initiator_handshake_send(...) {
  if (step == 0) {
    return send_message0(...);
  else
    return send_message2(...);
}

error_code responder_handshake_send(...) {
  return send_message1(...);
}
```

F* code:

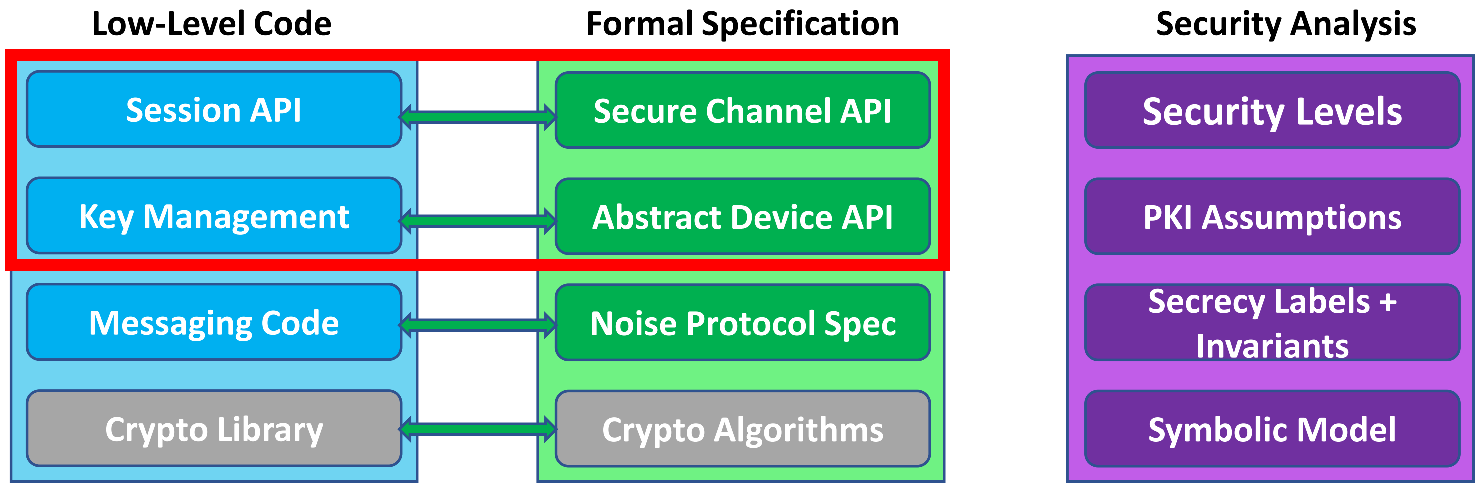
Meta parameter (i ∈ {0, 1, …})

Runtime parameter

```
// Pre: initiator==((i%2)==0) /\ i < num_handshake_messages
let rec handshake_send_i (initiator:bool) ... (i:nat) (step:UInt32.t) =
  if i+2 >= num_handshake_messages then
    ... // last possible send_message function
  else if step = size i then
    ... // instantiated send_message function
  else
    handshake_send ... (i+2) step // Increment i by 2
```

```
let initiator_handshake_send ... step = handshake_send true ... 0 step
let responder_handshake_send ... step = handshake_send false ... 1 step
```

What does the high-level API give us?



- State Machines
- ➡ Peer Management
- Key Storage & Validation
- Message Encapsulation

Devices and Peers (IKpsk2)

Device contains our **static identity** and stores remote **peers information** (linked list, no recursive functions):

Initialization and **premessages** phase:

```
// Alice
device* dv;
dv = create_device("Alice", alice_private_key, alice_public_key);

bob = device_add_peer(dv, "Bob", bob_public_key, alice_bob_psk);
charlie = device_add_peer(dv, "Charlie",
                          charlie_public_key,
                          alice_charlie_psk);
...
```

```
// Bob
device* dv;
dv = create_device("Bob", bob_private_key, bob_public_key);

...
```

Devices and Peers (IKpsk2)

Device contains our **static identity** and stores remote **peers information** (linked list, no recursive functions):

Initialization and **premessages** phase:

```
// Alice
device* dv;
dv = create_device("Alice", alice_private_key, alice_public_key);

bob = device_add_peer(dv, "Bob", bob_public_key, alice_bob_psk);
charlie = device_add_peer(dv, "Charlie",
                          charlie_public_key,
                          alice_charlie_psk);
...
```

```
// Bob
device* dv;
dv = create_device("Bob", bob_private_key, bob_public_key);

...
```

Handshake phase:

```
// Alice: talk to Bob
session *sn;
sn = create_initiator(dv, bob_id);
uint8_t out[...];
send_message(sn, "Hello Bob!", out, outlen);
... // Send message over the network
```

```
// On Bob's side
session *sn;
sn = create_responder(dv); // We don't know who we talk to yet
uint8_t msg[...];
... // Receive message over the network
receive_message(sn, out, msg_len); // Discover it is Alice
```

Devices and Peers (IKpsk2)

Device contains our **static identity** and stores remote **peers information** (linked list, no recursive functions):

Initialization and **premessages** phase:

```
// Alice
device* dv;
dv = create_device("Alice", alice_private_key, alice_public_key);

bob = device_add_peer(dv, "Bob", bob_public_key, alice_bob_psk);
charlie = device_add_peer(dv, "Charlie",
                          charlie_public_key,
                          alice_charlie_psk);
...
```

```
// Bob
device* dv;
dv = create_device("Bob", bob_private_key, bob_public_key);

...
```

IKpsk2:

← s **initiator knows responder from beginning**

...

→ e, es, s, ss

← e, ee, se, psk

Handshake phase:

```
// Alice: talk to Bob
session *sn;
sn = create_initiator(dv, bob_id);
uint8_t out[...];
send_message(sn, "Hello Bob!", out, outlen);
... // Send message over the network
```

```
// On Bob's side
session *sn;
sn = create_responder(dv); // We don't know who we talk to yet
uint8_t msg[...];
... // Receive message over the network
receive_message(sn, out, msg_len); // Discover it is Alice
```

Devices and Peers (IKpsk2)

Device contains our **static identity** and stores remote **peers information** (linked list, no recursive functions):

Initialization and **premessages** phase:

```
// Alice
device* dv;
dv = create_device("Alice", alice_private_key, alice_public_key);

bob = device_add_peer(dv, "Bob", bob_public_key, alice_bob_psk);
charlie = device_add_peer(dv, "Charlie",
                          charlie_public_key,
                          alice_charlie_psk);
...
```

```
// Bob
device* dv;
dv = create_device("Bob", bob_private_key, bob_public_key);

...
```

IKpsk2:

← s **initiator knows responder from beginning**

...

Responder learns initiator's identity

→ e, es, s, ss

← e, ee, se, psk

Handshake phase:

```
// Alice: talk to Bob
session *sn;
sn = create_initiator(dv, bob_id);
uint8_t out[...];
send_message(sn, "Hello Bob!", out, outlen);
... // Send message over the network
```

```
// On Bob's side
session *sn;
sn = create_responder(dv); // We don't know who we talk to yet
uint8_t msg[...];
... // Receive message over the network
receive_message(sn, out, msg_len); // Discover it is Alice
```

Devices and Peers (IKpsk2)

Device contains our **static identity** and stores remote **peers information** (linked list, no recursive functions):

Initialization and **premessages** phase:

```
// Alice
device* dv;
dv = create_device("Alice", alice_private_key, alice_public_key);

bob = device_add_peer(dv, "Bob", bob_public_key, alice_bob_psk);
charlie = device_add_peer(dv, "Charlie",
                          charlie_public_key,
                          alice_charlie_psk);
...
```

```
// Bob
device* dv;
dv = create_device("Bob", bob_private_key, bob_public_key);

...
```

IKpsk2:

← s **initiator knows responder from beginning**

... **Responder learns initiator's identity**

→ e, es, s, ss

← e, ee, se, psk

Handshake phase:

```
// Alice: talk to Bob
session *sn;
sn = create_initiator(dv, bob_id);
uint8_t out[...];
send_message(sn, "Hello Bob!", out, outlen);
... // Send message over the network
```

```
// On Bob's side
session *sn;
sn = create_responder(dv); // We don't know who we talk to yet
uint8_t msg[...];
... // Receive message over the network
receive_message(sn, out, msg_len); // Discover it is Alice
```

peer_id parameter: varies with pattern and role

Devices and Peers (IKpsk2)

Device contains our **static identity** and stores remote **peers information** (linked list, no recursive functions):

Initialization and **premessages** phase:

```
// Alice
device* dv;
dv = create_device("Alice", alice_private_key, alice_public_key);

bob = device_add_peer(dv, "Bob", bob_public_key, alice_bob_psk);
charlie = device_add_peer(dv, "Charlie",
                          charlie_public_key,
                          alice_charlie_psk);
...
```

```
// Bob
device* dv;
dv = create_device("Bob", bob_private_key, bob_public_key);

...
```

psk parameter only if pattern uses it

IKpsk2:

← s — initiator knows responder from beginning

...

Responder learns initiator’s identity

→ e, es, s, ss

← e, ee, se, psk

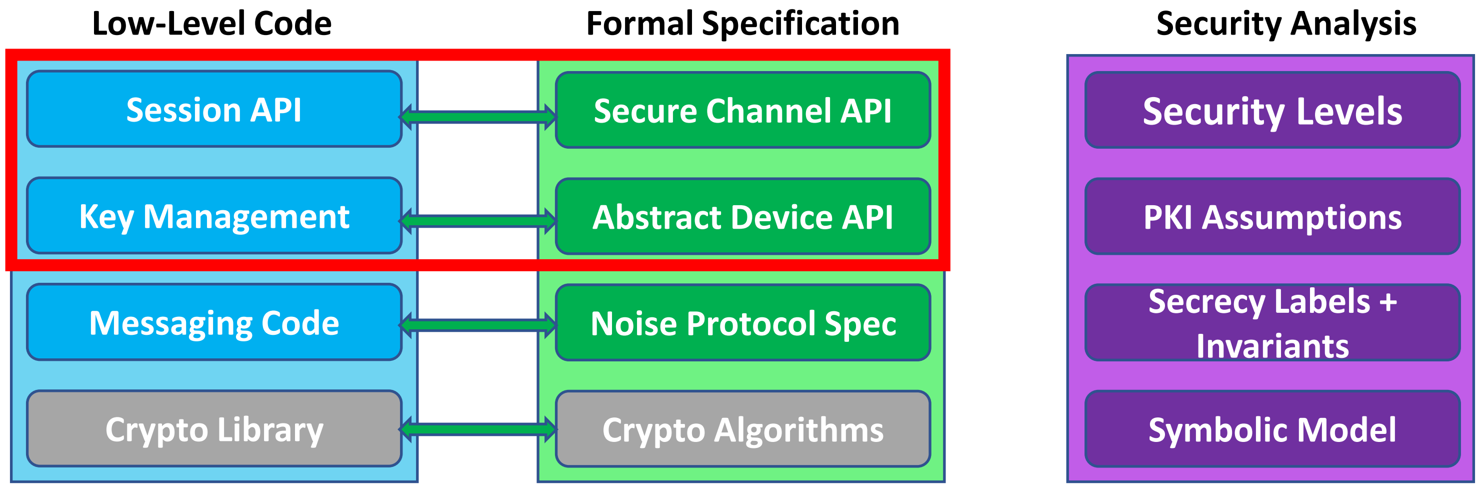
Handshake phase:

```
// Alice: talk to Bob
session *sn;
sn = create_initiator(dv, bob_id);
uint8_t out[...];
send_message(sn, "Hello Bob!", out, outlen);
... // Send message over the network
```

```
// On Bob's side
session *sn;
sn = create_responder(dv); // We don't know who we talk to yet
uint8_t msg[...];
... // Receive message over the network
receive_message(sn, out, msg_len); // Discover it is Alice
```

peer_id parameter: varies with pattern and role

What does the high-level API give us?



- State Machines
- Peer Management
- Key Storage & Validation
- Message Encapsulation

Key Storage and Validation

IKpsk2:

← s

. . .

→ e, es, s, ss

← e, ee, se, psk

Wireguard VPN: all remote static keys **must have been registered** in the device before

XX:

→ e

← e, ee, s, es

→ s, se

WhatsApp: we actually **transmit** keys, which must be validated by some external mean

Key Storage and Validation

IKpsk2:

← s

. . .

→ e, es, s, ss

← e, ee, se, psk

Wireguard VPN: all remote static keys **must have been registered** in the device before

XX:

→ e

← e, ee, s, es

→ s, se

WhatsApp: we actually **transmit** keys, which must be validated by some external mean

Key Storage and Validation

IKpsk2:

← s

. . .

→ e, es, s, ss

← e, ee, se, psk

Wireguard VPN: all remote static keys **must have been registered** in the device before

XX:

→ e

← e, ee, s, es

→ s, se

WhatsApp: we actually **transmit** keys, which must be validated by some external mean

Key Storage and Validation

IKpsk2:

← s

. . .

→ e, es, s, ss

← e, ee, se, psk

XX:

→ e

← e, ee, s, es

→ s, se

Wireguard VPN: all remote static keys **must have been registered** in the device before

WhatsApp: we actually **transmit** keys, which must be validated by some external mean

We parameterize our implementation with:

- **Policy** (`bool`): can we accept unknown remote keys? (Wireguard: `false` / WhatsApp: `true`)
- **Certification** function: `certification_state → public_key → payload → option peer_name`

Key Storage and Validation

IKpsk2:

← s

. . .

→ e, es, s, ss

← e, ee, se, psk

Wireguard VPN: all remote static keys **must have been registered** in the device before

XX:

→ e

← e, ee, s, es

→ s, se

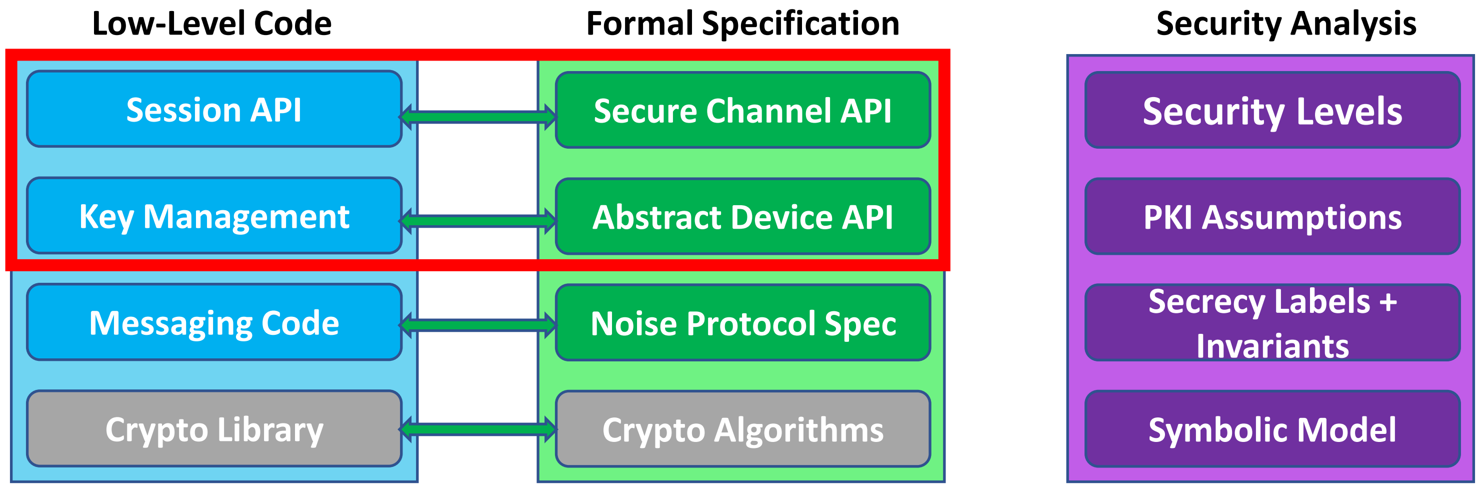
WhatsApp: we actually **transmit** keys, which must be validated by some external mean

We parameterize our implementation with:

- **Policy** (`bool`): can we accept unknown remote keys? (Wireguard: `false` / WhatsApp: `true`)
- **Certification** function: `certification_state → public_key → payload → option peer_name`

Long-term keys storage (on disk): serialization/deserialization functions for device static identity and peers (random nonces + device/peer name as authentication data).

What does the high-level API give us?



- State Machines
- Peer Management
- Key Storage & Validation
- Message Encapsulation

Message Encapsulation – Security Levels

Every payload has an **authentication level** (≤ 2) and a **confidentiality level** (≤ 5):

IKpsk2	**Payload Conf. Level**	
	→	←
← s		
. . .		
→ e, es, s, ss	2	-
← e, ee, se, psk	-	4
→	5	-
←	-	5

Message Encapsulation – Security Levels

Every payload has an **authentication level** (≤ 2) and a **confidentiality level** (≤ 5):

IKpsk2	Payload Conf. Level	
	→	←
← s		
. . .		
→ e, es, s, ss	2	-
← e, ee, se, psk	-	4
→	5	-
←	-	5

XX	Payload Conf. Level	
	→	←
→ e	0	-
← e, ee, s, es	-	1
→ s, se	5	-
←	-	5
→	5	-
…		

Message Encapsulation – Security Levels

Every payload has an **authentication level** (≤ 2) and a **confidentiality level** (≤ 5):

IKpsk2	Payload Conf. Level	
	→	←
← s		
. . .		
→ e, es, s, ss	2	-
← e, ee, se, psk	-	4
→	5	-
←	-	5

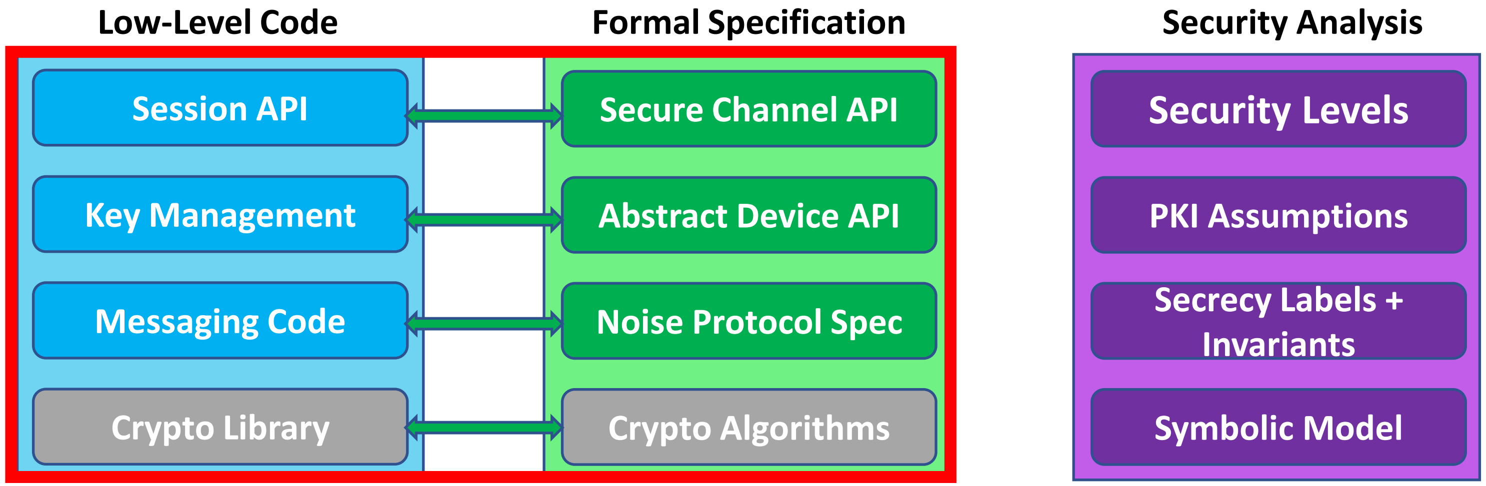
XX	Payload Conf. Level	
	→	←
→ e	0	-
← e, ee, s, es	-	1
→ s, se	5	-
←	-	5
→	5	-
…		

We protect the user from sending secret data/trusting received data **too early** (dynamic checks on **user-friendly auth./conf. levels**):

```
encap_message_t *pack_with_conf_level(
  uint8_t requested_conf_level, // <--- confidentiality
  const char *session_name, const char *peer_name, uint32_t msg_len, uint8_t *msg);

bool unpack_message_with_auth_level(
  uint32 t *out_msg_len, uint8 t **out_msg, char *session_name, char *peer_name,
  uint8_t requested_auth_level, // <--- authentication
  encap_message_t *msg);
```

Generated Code & Performance



Generated Code (IKpsk2)

Some signatures:

```
Noise_peer_t
*Noise_device_add_peer(Noise_device_t *dvp, uint8_t *pinfo, uint8_t *rs, uint8_t *psk);

void Noise_device_remove_peer(Noise_device_t *dvp, uint32_t pid);

Noise_peer_t *Noise_device_lookup_peer_by_id(Noise_device_t *dvp, uint32_t id);

Noise_peer_t *Noise_device_lookup_peer_by_static(Noise_device_t *dvp, uint8_t *s);

Noise_session_t *Noise_session_create_initiator(Noise_device_t *dvp, uint32_t pid);

Noise_session_t *Noise_session_create_responder(Noise_device_t *dvp);

void Noise_session_free(Noise_session_t *sn);

Noise_rcode
Noise_session_write(
  Noise_encap_message_t *payload,
  Noise_session_t *sn_p,
  uint32_t *out_len,
  uint8_t **out
);

Noise_rcode
Noise_session_read(
  Noise_encap_message_t **payload_out,
  Noise_session_t *sn_p,
  uint32_t inlen,
  uint8_t *input
);
```

`session_write` (length checks, security level checks…):

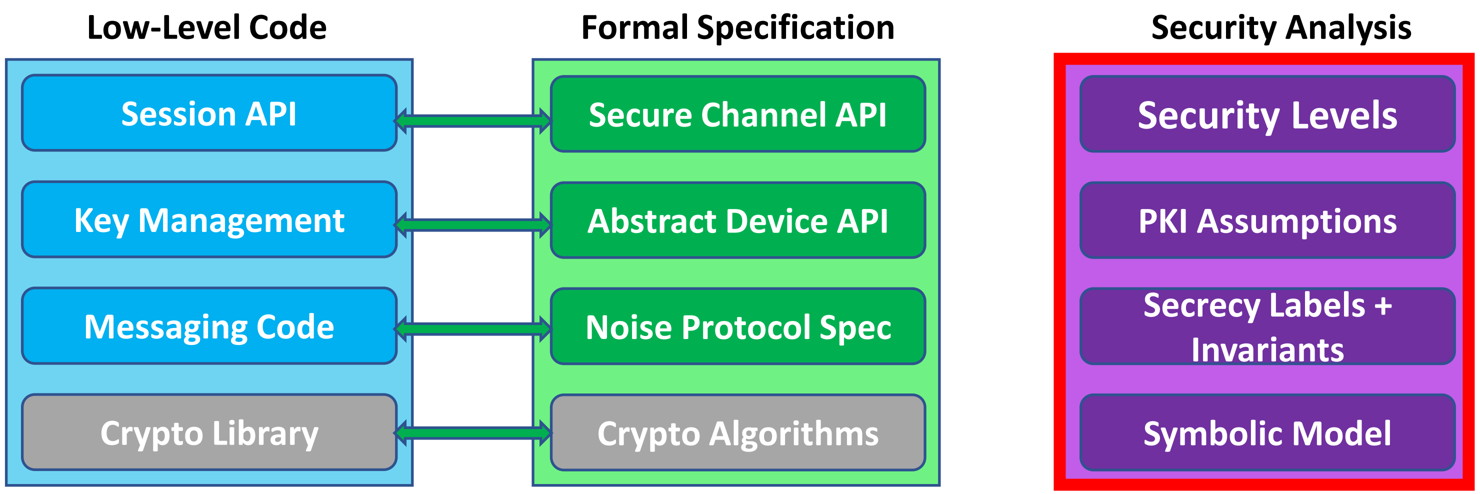
```
if (sn.tag == Noise_DS_Initiator)
{
  Noise_init_state_t sn_state = sn.val.case_DS_Initiator.state;
  if (sn_state.tag == Noise_IMS_Transport)
  {
    Noise_encap_message_t encap_payload = payload[0U];
    bool next_length_ok;
    if (encap_payload.em_message_len <= (uint32_t)4294967279U)
    {
      out_len[0U] = encap_payload.em_message_len + (uint32_t)16U;
      next_length_ok = true;
    }
    else
      next_length_ok = false;
    if (next_length_ok)
    {
      bool sec_ok;
      if (encap_payload.em_message_len == (uint32_t)0U)
        sec_ok = true;
      else
      {
        uint8_t clevel = (uint8_t)5U;
        if (encap_payload.em_ac_level.tag == Noise_Conf_level)
        {
          uint8_t req_level = encap_payload.em_ac_level.val.case_Conf_level;
          sec_ok =
            (req_level >= (uint8_t)2U && clevel >= req_level)
            || (req_level == (uint8_t)1U && (clevel == req_level || clevel >= (uint8_t)3U))
            || req_level == (uint8_t)0U;
        }
        else
          sec_ok = false;
      }
      if (sec_ok)
```

Performance

Pattern	Noise*	Custom	Cacophony	NoiseExpl.	Noise-C
X	6677	N/A	2272	4955	5603
NX	5385	N/A	2392	4046	5065
XX	3917	N/A	1593	3149	3577
IK	3143	N/A	1357	2459	2822
IKpsk2	3138	3756	1194	2431	N/A

Performance Comparison, in handshakes / second. Benchmark performed on a Dell XPS13 laptop (Intel Core i7-10510U) with Ubuntu 18.04

Security Analysis



Security Analysis - Dolev-Yao*

DY*: framework for symbolic analysis developed in F*.
We do the security analysis **once and for all**.

Security Analysis - Dolev-Yao*

DY*: framework for symbolic analysis developed in F*.
We do the security analysis **once and for all**.

We **formalize the Noise security levels with predicates**, and prove that those predicates are satisfied at the proper steps of the proper handshakes:

Level	Confidentiality Predicate (over i, idx, and l)
0	$\top$
1	can_flow i (CanRead [S idx.p idx.sid] ⊔ idx.peer_eph_label) l
2	can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l
3	can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l ∧ can_flow i (CanRead [S idx.p idx.sid] ⊔ idx.peer_eph_label) l
4	can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l ∧ can_flow i (CanRead [S idx.p idx.sid] ⊔ idx.peer_eph_label) l ∧ (compromised_before i (P idx.p) ∨ compromised_before i (P idx.peer) ∨ (∃sid'. peer_eph_label == CanRead [S idx.peer sid']))
5	can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l ∧ can_flow i (CanRead [S idx.p idx.sid] ⊔ idx.peer_eph_label) l ∧ (compromised_before i (S idx.p idx.sid) ∨ compromised_before i (P idx.peer) ∨ (∃sid'. peer_eph_label == CanRead [S idx.peer sid']))

Level	Authentication Predicate (over i, idx, and l)
0	$\top$
1	can_flow i (CanRead [P idx.p; P idx.peer]) l
2	can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l

Security Analysis - Dolev-Yao*

DY*: framework for symbolic analysis developed in F*.
We do the security analysis **once and for all**.

We **formalize the Noise security levels with predicates**, and prove that those predicates are satisfied at the proper steps of the proper handshakes:

Level	Confidentiality Predicate (over i, idx, and l)
0	⊤
1	can_flow i (CanRead [S idx.p idx.sid] ⊔ idx.peer_eph_label) l
2	can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l
3	can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l ∧ can_flow i (CanRead [S idx.p idx.sid] ⊔ idx.peer_eph_label) l
4	can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l ∧ can_flow i (CanRead [S idx.p idx.sid] ⊔ idx.peer_eph_label) l ∧ (compromised_before i (P idx.p) ∨ compromised_before i (P idx.peer) ∨ (∃sid'. peer_eph_label == CanRead [S idx.peer sid']))
5	can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l ∧ can_flow i (CanRead [S idx.p idx.sid] ⊔ idx.peer_eph_label) l ∧ (compromised_before i (S idx.p idx.sid) ∨ compromised_before i (P idx.peer) ∨ (∃sid'. peer_eph_label == CanRead [S idx.peer sid']))

Strong forward-secrecy

Level	Authentication Predicate (over i, idx, and l)
0	⊤
1	can_flow i (CanRead [P idx.p; P idx.peer]) l
2	can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l

Security Analysis - Dolev-Yao*

Define labels for the data-types:

- `CanRead [P "Alice"]` : static data that can only be read by principal `"Alice"`
- `CanRead [S "Bob" sid]` : ephemeral data that can only be read by principal `"Bob"` at session `sid`

Security Analysis - Dolev-Yao*

Define labels for the data-types:

- `CanRead [P "Alice"]` : static data that can only be read by principal `"Alice"`
- `CanRead [S "Bob" sid]` : ephemeral data that can only be read by principal `"Bob"` at session `sid`

Annotate the data types to give them usages and labels:

- `dh_private_key l` : private key of label `l`
- `dh_public_key l` : public key associated to a private key of label `l`

Security Analysis - Dolev-Yao*

Define labels for the data-types:

- `CanRead [P "Alice"]` : static data that can only be read by principal `"Alice"`
- `CanRead [S "Bob" sid]` : ephemeral data that can only be read by principal `"Bob"` at session `sid`

Annotate the data types to give them usages and labels:

- `dh_private_key l` : private key of label `l`
- `dh_public_key l` : public key associated to a private key of label `l`

```
// DH signature (simplified)
val dh (l1 : label) (priv : dh_private_key l1)
       (l2 : label) (pub : dh_public_key l2) :
  dh_result (join l1 l2) // l1 ⊔ l2
```

Security Analysis - Dolev-Yao*

Define labels for the data-types:

- CanRead [P "Alice"] : static data that can only be read by principal "Alice"
- CanRead [S "Bob" sid] : ephemeral data that can only be read by principal "Bob" at session sid

Annotate the data types to give them usages and labels:

- dh_private_key l : private key of label l
- dh_public_key l : public key associated to a private key of label l

```
// DH signature (simplified)
val dh (l1 : label) (priv : dh_private_key l1)
       (l2 : label) (pub : dh_public_key l2) :
  dh_result (join l1 l2) // l1 ⊔ l2
```

For now: those annotations are purely **syntactic**

Security Analysis - Example

Alice **Bob**

→ e, es, s, ss, [d]

```
let ck0 = hash "Noise_IKpsk2_..." in
// e
...
// es
let dh_es = dh e rs in
let ck1, sk1 = kdf2 ck0 dh_es in
// s
...
// ss
let dh_ss = dh s rs in
let ck2, sk2 = kdf2 ck1 dh_ss in
// d (plain text)
let cipher =
  aead_encrypt sk2 ... plain
in
...
// Output
concat ... cipher
```

Security Analysis - Example

Alice **Bob**

→ e, es, s, ss, [d]

```
let ck0 = hash "Noise_IKpsk2_..." in
// e
...
// es
let dh_es = dh e rs in
let ck1, sk1 = kdf2 ck0 dh_es in
// s
...
// ss
let dh_ss = dh s rs in
let ck2, sk2 = kdf2 ck1 dh_ss in
// d (plain text)
let cipher =
  aead_encrypt sk2 ... plain
in
...
// Output
concat ... cipher
```

```
l_es := ((CanRead [S "Alice" sn]) ⊔ (CanRead [P "Bob"]))
dh_es : dh_result l_es
```

Security Analysis - Example

Alice **Bob**

→ e, es, s, ss, [d]

```
let ck0 = hash "Noise_IKpsk2_..." in
// e
...
// es
let dh_es = dh e rs in
let ck1, sk1 = kdf2 ck0 dh_es in
// s
...
// ss
let dh_ss = dh s rs in
let ck2, sk2 = kdf2 ck1 dh_ss in
// d (plain text)
let cipher =
  aead_encrypt sk2 ... plain
in
...
// Output
concat ... cipher
```

```
l_es := ((CanRead [S "Alice" sn]) ⊔ (CanRead [P "Bob"]))
dh_es : dh_result l_es
```

```
l_ss := ((CanRead [P "Alice"])    ⊔ (CanRead [P "Bob"]))
dh_ss : dh_result l_ss
```

Security Analysis - Example

Alice **Bob**

→ e, es, s, ss, [d]

```
let ck0 = hash "Noise_IKpsk2_..." in
// e
...
// es
let dh_es = dh e rs in
let ck1, sk1 = kdf2 ck0 dh_es in
// s
...
// ss
let dh_ss = dh s rs in
let ck2, sk2 = kdf2 ck1 dh_ss in
// d (plain text)
let cipher =
  aead_encrypt sk2 ... plain
in
...
// Output
concat ... cipher
```

```
l_es := ((CanRead [S "Alice" sn]) ⊔ (CanRead [P "Bob"]))
dh_es : dh_result l_es
```

```
l_ss := ((CanRead [P "Alice"])    ⊔ (CanRead [P "Bob"]))
dh_ss : dh_result l_ss
```

```
ck0 : chaining_key public
ck1 : chaining_key (public ⊓ l_es)
ck2 : chaining_key ((public ⊓ l_es) ⊓ l_ss)
```

Security Analysis - Example

Alice **Bob**

→ e, es, s, ss, [d]

```
let ck0 = hash "Noise_IKpsk2_..." in
// e
...
// es
let dh_es = dh e rs in
let ck1, sk1 = kdf2 ck0 dh_es in
// s
...
// ss
let dh_ss = dh s rs in
let ck2, sk2 = kdf2 ck1 dh_ss in
// d (plain text)
let cipher =
  aead_encrypt sk2 ... plain
in
...
// Output
concat ... cipher
```

```
l_es := ((CanRead [S "Alice" sn]) ⊔ (CanRead [P "Bob"]))
dh_es : dh_result l_es
```

```
l_ss := ((CanRead [P "Alice"])    ⊔ (CanRead [P "Bob"]))
dh_ss : dh_result l_ss
```

```
ck0 : chaining_key public
ck1 : chaining_key (public ⊓ l_es)
ck2 : chaining_key ((public ⊓ l_es) ⊓ l_ss)
```

```
val aead_encrypt
    (#l : label)
    (sk : aead_key l)   // encryption key
    (iv : msg public)   // nonce
    (plain : msg l)     // plaintext
    (ad : msg public) : // authentication data
    msg public
```

Security Analysis - Example

Alice **Bob**

→ e, es, s, ss, [d]

```
let ck0 = hash "Noise_IKpsk2_..." in
// e
...
// es
let dh_es = dh e rs in
let ck1, sk1 = kdf2 ck0 dh_es in
// s
...
// ss
let dh_ss = dh s rs in
let ck2, sk2 = kdf2 ck1 dh_ss in
// d (plain text)
let cipher =
  aead_encrypt sk2 ... plain
in
...
// Output
concat ... cipher
```

```
l_es := ((CanRead [S "Alice" sn]) ⊔ (CanRead [P "Bob"]))
dh_es : dh_result l_es
```

```
l_ss := ((CanRead [P "Alice"])    ⊔ (CanRead [P "Bob"]))
dh_ss : dh_result l_ss
```

```
ck0 : chaining_key public
ck1 : chaining_key (public ⊓ l_es)
ck2 : chaining_key ((public ⊓ l_es) ⊓ l_ss)
```

```
val aead_encrypt
    (#l : label)
  → (sk : aead_key l)    // encryption key
    (iv : msg public)    // nonce
  → (plain : msg l)      // plaintext
    (ad : msg public) :  // authentication data
    msg public
```

Security Analysis - Example

Alice **Bob**

→ e, es, s, ss, [d]

```
let ck0 = hash "Noise_IKpsk2_..." in
// e
...
// es
let dh_es = dh e rs in
let ck1, sk1 = kdf2 ck0 dh_es in
// s
...
// ss
let dh_ss = dh s rs in
let ck2, sk2 = kdf2 ck1 dh_ss in
// d (plain text)
let cipher =
  aead_encrypt sk2 ... plain
in
...
// Output
concat ... cipher
```

```
l_es := ((CanRead [S "Alice" sn]) ⊔ (CanRead [P "Bob"]))
dh_es : dh_result l_es
```

```
l_ss := ((CanRead [P "Alice"])    ⊔ (CanRead [P "Bob"]))
dh_ss : dh_result l_ss
```

```
ck0 : chaining_key public
ck1 : chaining_key (public ⊓ l_es)
ck2 : chaining_key ((public ⊓ l_es) ⊓ l_ss)
```

```
val aead_encrypt
    (#l : label)
  → (sk : aead_key l)    // encryption key
    (iv : msg public)    // nonce
  → (plain : msg l)      // plaintext
    (ad : msg public) :  // authentication data
    msg public
```

We can then send the encrypted message: register a **Send** event in a global trace

Security Analysis: `can_flow`

- Labels are purely **syntactic**
- **Semantics** of DY* are given through a `can_flow` predicate which states properties about a global trace of events
- The content of a message sent over the network is **compromised** if its label flows to `public`

Security Analysis: `can_flow`

- Labels are purely **syntactic**
- **Semantics** of DY* are given through a `can_flow` predicate which states properties about a global trace of events
- The content of a message sent over the network is **compromised** if its label flows to `public`

- Labels can flow to more secret labels (i is a timestamp):

```
can_flow i (CanRead [P p1]) (CanRead [P p1] ⊓ CanRead [P p2])
```

Security Analysis: `can_flow`

- Labels are purely **syntactic**
- **Semantics** of DY* are given through a `can_flow` predicate which states properties about a global trace of events
- The content of a message sent over the network is **compromised** if its label flows to `public`

- Labels can flow to more secret labels (i is a timestamp):

```
can_flow i (CanRead [P p1]) (CanRead [P p1] ⊓ CanRead [P p2])
```

- The attacker can **dynamically compromise** a participant's current state: event `Compromise p ...`
- A label is compromised (and data with this label) if it flows to `public` :

```
compromised_before i (P p) ==> can_flow i (CanRead [P p]) public
compromised_before i (S p sid) ==> can_flow i (CanRead [S p sid]) public
...
```

Security Analysis: `can_flow`

- Labels are purely **syntactic**
- **Semantics** of DY* are given through a `can_flow` predicate which states properties about a global trace of events
- The content of a message sent over the network is **compromised** if its label flows to `public`

- Labels can flow to more secret labels (i is a timestamp):

```
can_flow i (CanRead [P p1]) (CanRead [P p1] ⊓ CanRead [P p2])
```

- The attacker can **dynamically compromise** a participant's current state: event `Compromise p ...`
- A label is compromised (and data with this label) if it flows to `public` :

```
compromised_before i (P p) ==> can_flow i (CanRead [P p]) public
compromised_before i (S p sid) ==> can_flow i (CanRead [S p sid]) public
...
```

- If a label flows to `public` we can deduce the existence of compromise events :

```
can_flow i (CanRead [P p]) public ==> compromised_before i (P p)
```

Security Analysis – Security Predicates

Confidentiality level 5 (**strong forward secrecy**):

```
can_flow i (CanRead [S p sid] ⊔ CanRead [P peer]) l /\
... /\
(... \/
 ...)
```

Security Analysis – Security Predicates

Confidentiality level 5 (**strong forward secrecy**):

```
can_flow i (CanRead [S p sid] ⊔ CanRead [P peer]) l /\
can_flow i (CanRead [S p sid] ⊔ get_dh_label re) l /\
(... \/
 ...)
```

Security Analysis – Security Predicates

Confidentiality level 5 (**strong forward secrecy**):

```
can_flow i (CanRead [S p sid] ⊔ CanRead [P peer]) l /\
can_flow i (CanRead [S p sid] ⊔ get_dh_label re) l /\
(... \/
 ...)
```

We initially have no information about re : we link it to the remote static key (which has been certified)

Security Analysis – Security Predicates

Confidentiality level 5 (**strong forward secrecy**):

```
can_flow i (CanRead [S p sid] ⊔ CanRead [P peer]) l /\
can_flow i (CanRead [S p sid] ⊔ get_dh_label re) l /\
(compromised_before i (S p sid) \/ compromised_before i (P peer) \/
 (∃ sid'. get_dh_label re == CanRead [S peer sid']))
```

We initially have no information about re : we link it to the remote static key (which has been certified)

Security Analysis – Security Predicates

Confidentiality level 5 (**strong forward secrecy**):

```
can_flow i (CanRead [S p sid] ⊔ CanRead [P peer]) l /\
can_flow i (CanRead [S p sid] ⊔ get_dh_label re) l /\
(compromised_before i (S p sid) \/ compromised_before i (P peer) \/
 (∃ sid'. get_dh_label re == CanRead [S peer sid']))
```

We initially have no information about re : we link it to the remote static key (which has been certified)

Authentication invariant (an aead encrypt/decrypt):

```
... \/
begin match opt_rs, opt_re with
| Some rs, Some re ->
  ∃ peer' sid'. get_dh_label rs = CanRead [P peer'] /\
                get_dh_label re = CanRead [S peer' sid']
| _ -> True
end
```

Security Analysis – Security Predicates

Confidentiality level 5 (**strong forward secrecy**):

```
can_flow i (CanRead [S p sid] ⊔ CanRead [P peer]) l /\
can_flow i (CanRead [S p sid] ⊔ get_dh_label re) l /\
(compromised_before i (S p sid) \/ compromised_before i (P peer) \/
 (∃ sid'. get_dh_label re == CanRead [S peer sid']))
```

We initially have no information about re : we link it to the remote static key (which has been certified)

Authentication invariant (an aead encrypt/decrypt):

```
can_flow i l public \/
begin match opt_rs, opt_re with
| Some rs, Some re ->
  ∃ peer' sid'. get_dh_label rs = CanRead [P peer'] /\
                get_dh_label re = CanRead [S peer' sid']
| _ -> True
end
```

Security Analysis – Security Predicates

Confidentiality level 5 (**strong forward secrecy**):

```
can_flow i (CanRead [S p sid] ⊔ CanRead [P peer]) l /\
can_flow i (CanRead [S p sid] ⊔ get_dh_label re) l /\
(compromised_before i (S p sid) \/ compromised_before i (P peer) \/
 (∃ sid'. get_dh_label re == CanRead [S peer sid']))
```

We initially have no information about re : we link it to the remote static key (which has been certified)

Authentication invariant (an aead encrypt/decrypt):

```
can_flow i l public \/
begin match opt_rs, opt_re with
| Some rs, Some re ->
  ∃ peer' sid'. get_dh_label rs = CanRead [P peer'] /\
                get_dh_label re = CanRead [S peer' sid']
| _ -> True
end
```

Certification of remote static key gives:

```
get_dh_label rs = CanRead [P peer]
```

Security Analysis - Dolev-Yao*

DY*: framework for symbolic analysis developed in F*.
We do the security analysis **once and for all**.

1. We **add annotations** to types to reflect security properties:

```
// DH signature (simplified)
val dh (l1 : label) (priv : dh_private_key l1)
       (l2 : label) (pub : dh_public_key l2) :
  dh_result (join l1 l2) // label: l1 ⊔ l2
```

Security Analysis - Dolev-Yao*

DY*: framework for symbolic analysis developed in F*.
We do the security analysis **once and for all**.

1. We **add annotations** to types to reflect security properties:

```
// DH signature (simplified)
val dh (l1 : label) (priv : dh_private_key l1)
       (l2 : label) (pub : dh_public_key l2) :
  dh_result (join l1 l2) // label: l1 ⊔ l2
```

2. We **generate target labels** for every step of the handshake:

IKpsk2 (from the responder's point of view)

← s

...

→ e, es, s, ss, [d] l1 = (peer_eph_label ⊔ CanRead [P p]) ⊓ (. . .)

← e, ee, se, psk, [d] l2 = (peer_eph_label ⊔ CanRead [P p]) ⊓ (. . .) ⊓
(peer_eph_label ⊔ CanRead [S p sid]) ⊓ (. . .)

. . .

Security Analysis - Dolev-Yao*

DY*: framework for symbolic analysis developed in F*.
We do the security analysis **once and for all**.

1. We **add annotations** to types to reflect security properties:

```
// DH signature (simplified)
val dh (l1 : label) (priv : dh_private_key l1)
       (l2 : label) (pub : dh_public_key l2) :
  dh_result (join l1 l2) // label: l1 ⊔ l2
```

2. We **generate target labels** for every step of the handshake:

IKpsk2 (from the responder's point of view)

← s

. . .

→ e, es, s, ss, [d]   l1 = (peer_eph_label ⊔ CanRead [P p]) ⊓ (. . .)

← e, ee, se, psk, [d]   l2 = (peer_eph_label ⊔ CanRead [P p]) ⊓ (. . .) ⊓
(peer_eph_label ⊔ CanRead [S p sid]) ⊓ (. . .)

. . .

3. We prove that the **handshake state meets** at each stage of the protocol the **corresponding security label**

Security Analysis - Dolev-Yao*

DY*: framework for symbolic analysis developed in F*.
We do the security analysis **once and for all**.

1. We **add annotations** to types to reflect security properties:

```
// DH signature (simplified)
val dh (l1 : label) (priv : dh_private_key l1)
       (l2 : label) (pub : dh_public_key l2) :
  dh_result (join l1 l2) // label: l1 ⊔ l2
```

2. We **generate target labels** for every step of the handshake:

IKpsk2 (from the responder's point of view)

← s

. . .

→ e, es, s, ss, [d]   l1 = (peer_eph_label ⊔ CanRead [P p]) ⊓ (. . .)

← e, ee, se, psk, [d]   l2 = (peer_eph_label ⊔ CanRead [P p]) ⊓ (. . .) ⊓
(peer_eph_label ⊔ CanRead [S p sid]) ⊓ (. . .)

. . .

3. We prove that the **handshake state meets** at each stage of the protocol the **corresponding security label**

4. We prove an **ephemeral authentication invariant** to link the remote ephemeral to the remote static (while remote static is validated by certification function)

Security Analysis - Dolev-Yao*

DY*: framework for symbolic analysis developed in F*.
We do the security analysis **once and for all**.

1. We **add annotations** to types to reflect security properties:

```
// DH signature (simplified)
val dh (l1 : label) (priv : dh_private_key l1)
       (l2 : label) (pub : dh_public_key l2) :
  dh_result (join l1 l2) // label: l1 ⊔ l2
```

2. We **generate target labels** for every step of the handshake:

IKpsk2 (from the responder's point of view)

← s

…

→ e, es, s, ss, [d] l1 = (peer_eph_label ⊔ CanRead [P p]) ⊓ (. . .)

← e, ee, se, psk, [d] l2 = (peer_eph_label ⊔ CanRead [P p]) ⊓ (. . .) ⊓
(peer_eph_label ⊔ CanRead [S p sid]) ⊓ (. . .)

. . .

3. We prove that the **handshake state meets** at each stage of the protocol the **corresponding security label**

4. We prove an **ephemeral authentication invariant** to link the remote ephemeral to the remote static (while remote static is validated by certification function)

5. We **formalize the Noise security levels** with predicates over labels:

Level	Confidentiality Predicate (over i, idx, and l)
0	⊤
1	can_flow i (CanRead [S idx.p idx.sid] ⊔ idx.peer_eph_label) l
2	can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l
3	can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l ∧ can_flow i (CanRead [S idx.p idx.sid] ⊔ idx.peer_eph_label) l
4	can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l ∧ can_flow i (CanRead [S idx.p idx.sid] ⊔ idx.peer_eph_label) l ∧ (compromised_before i (P idx.p) ∨ compromised_before i (P idx.peer) ∨ (∃sid'. peer_eph_label == CanRead [S idx.peer sid']))
5	can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l ∧ can_flow i (CanRead [S idx.p idx.sid] ⊔ idx.peer_eph_label) l ∧ (compromised_before i (S idx.p idx.sid) ∨ compromised_before i (P idx.peer) ∨ (∃sid'. peer_eph_label == CanRead [S idx.peer sid']))

Level	Authentication Predicate (over i, idx, and l)
0	⊤
1	can_flow i (CanRead [P idx.p; P idx.peer]) l
2	can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l

Security Analysis - Dolev-Yao*

DY*: framework for symbolic analysis developed in F*.
We do the security analysis **once and for all**.

1. We **add annotations** to types to reflect security properties:

```
// DH signature (simplified)
val dh (l1 : label) (priv : dh_private_key l1)
       (l2 : label) (pub : dh_public_key l2) :
  dh_result (join l1 l2) // label: l1 ⊔ l2
```

2. We **generate target labels** for every step of the handshake:

IKpsk2 (from the responder's point of view)

← s

. . .

→ e, es, s, ss, [d] l1 = (peer_eph_label ⊔ CanRead [P p]) ⊓ (. . .)

← e, ee, se, psk, [d] l2 = (peer_eph_label ⊔ CanRead [P p]) ⊓ (. . .) ⊓
(peer_eph_label ⊔ CanRead [S p sid]) ⊓ (. . .)

. . .

3. We prove that the **handshake state meets** at each stage of the protocol the **corresponding security label**

4. We prove an **ephemeral authentication invariant** to link the remote ephemeral to the remote static (while remote static is validated by certification function)

5. We **formalize the Noise security levels** with predicates over labels:

Level	Confidentiality Predicate (over i, idx, and l)
0	⊤
1	can_flow i (CanRead [S idx.p idx.sid] ⊔ idx.peer_eph_label) l
2	can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l
3	can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l ∧ can_flow i (CanRead [S idx.p idx.sid] ⊔ idx.peer_eph_label) l
4	can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l ∧ can_flow i (CanRead [S idx.p idx.sid] ⊔ idx.peer_eph_label) l ∧ (compromised_before i (P idx.p) ∨ compromised_before i (P idx.peer) ∨ (∃sid'. peer_eph_label == CanRead [S idx.peer sid']))
5	can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l ∧ can_flow i (CanRead [S idx.p idx.sid] ⊔ idx.peer_eph_label) l ∧ (compromised_before i (S idx.p idx.sid) ∨ compromised_before i (P idx.peer) ∨ (∃sid'. peer_eph_label == CanRead [S idx.peer sid']))

Level	Authentication Predicate (over i, idx, and l)
0	⊤
1	can_flow i (CanRead [P idx.p; P idx.peer]) l
2	can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l

6. We prove that those **security predicates are satisfied** by the target labels by combining 3. and 4.

Security Analysis - Dolev-Yao*

DY*: framework for symbolic analysis developed in F*.
We do the security analysis **once and for all**.

1. We **add annotations** to types to reflect security properties:

```
// DH signature (simplified)
val dh (l1 : label) (priv : dh_private_key l1)
       (l2 : label) (pub : dh_public_key l2) :
  dh_result (join l1 l2) // label: l1 ⊔ l2
```

2. We **generate target labels** for every step of the handshake:

IKpsk2 (from the responder's point of view)

← s

...

Strong forward-secrecy

→ e, es, s, ss, [d]   l1 = (peer_eph_label ⊔ CanRead [P p]) ⊓ (. . .)

← e, ee, se, psk, [d]   l2 = (peer_eph_label ⊔ CanRead [P p]) ⊓ (. . .) ⊓
(peer_eph_label ⊔ CanRead [S p sid]) ⊓ (. . .)

. . .

3. We prove that the **handshake state meets** at each stage of the protocol the **corresponding security label**

4. We prove an **ephemeral authentication invariant** to link the remote ephemeral to the remote static (while remote static is validated by certification function)

5. We **formalize the Noise security levels** with predicates over labels:

Level	Confidentiality Predicate (over i, idx, and l)
0	⊤
1	can_flow i (CanRead [S idx.p idx.sid] ⊔ idx.peer_eph_label) l
2	can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l
3	can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l ∧ can_flow i (CanRead [S idx.p idx.sid] ⊔ idx.peer_eph_label) l
4	can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l ∧ can_flow i (CanRead [S idx.p idx.sid] ⊔ idx.peer_eph_label) l ∧ (compromised_before i (P idx.p) ∨ compromised_before i (P idx.peer) ∨ (∃sid'. peer_eph_label == CanRead [S idx.peer sid']))
5	can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l ∧ can_flow i (CanRead [S idx.p idx.sid] ⊔ idx.peer_eph_label) l ∧ (compromised_before i (S idx.p idx.sid) ∨ compromised_before i (P idx.peer) ∨ (∃sid'. peer_eph_label == CanRead [S idx.peer sid']))

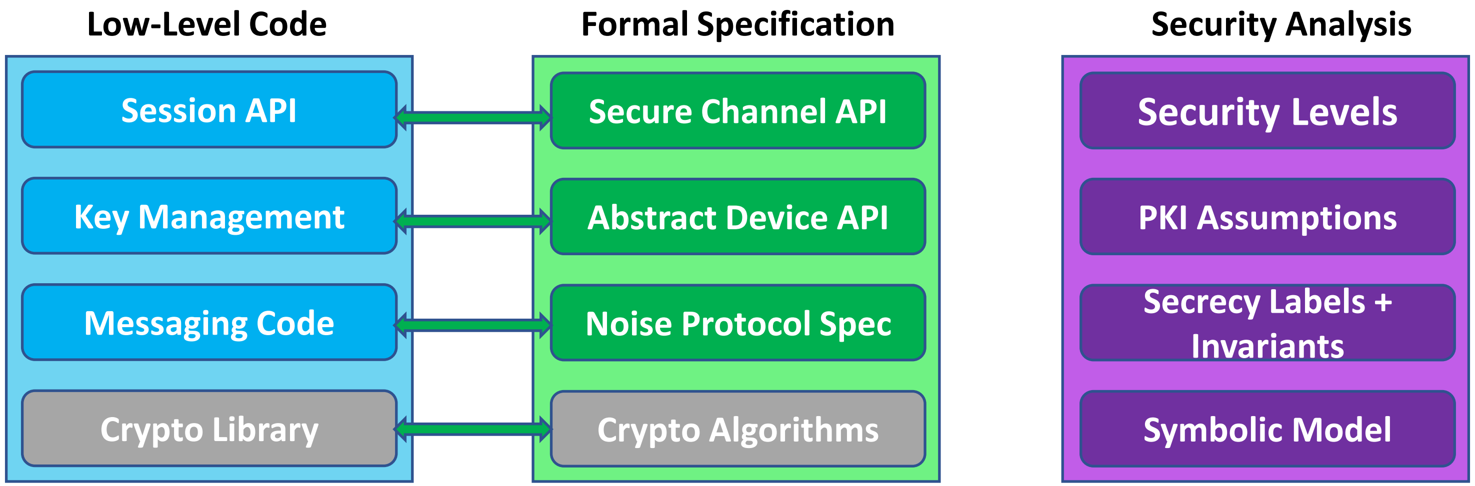
Level	Authentication Predicate (over i, idx, and l)
0	⊤
1	can_flow i (CanRead [P idx.p; P idx.peer]) l
2	can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l

6. We prove that those **security predicates are satisfied** by the target labels by combining 3. and 4.

Conclusion

We introduced **Noise***, which is:

- A compiler from Noise protocol patterns to efficient, verified C code, executed in F* normalizer
- A complete, verified library stack exposed through a high-level, defensive API
- A symbolic security analysis generically performed on protocol and API



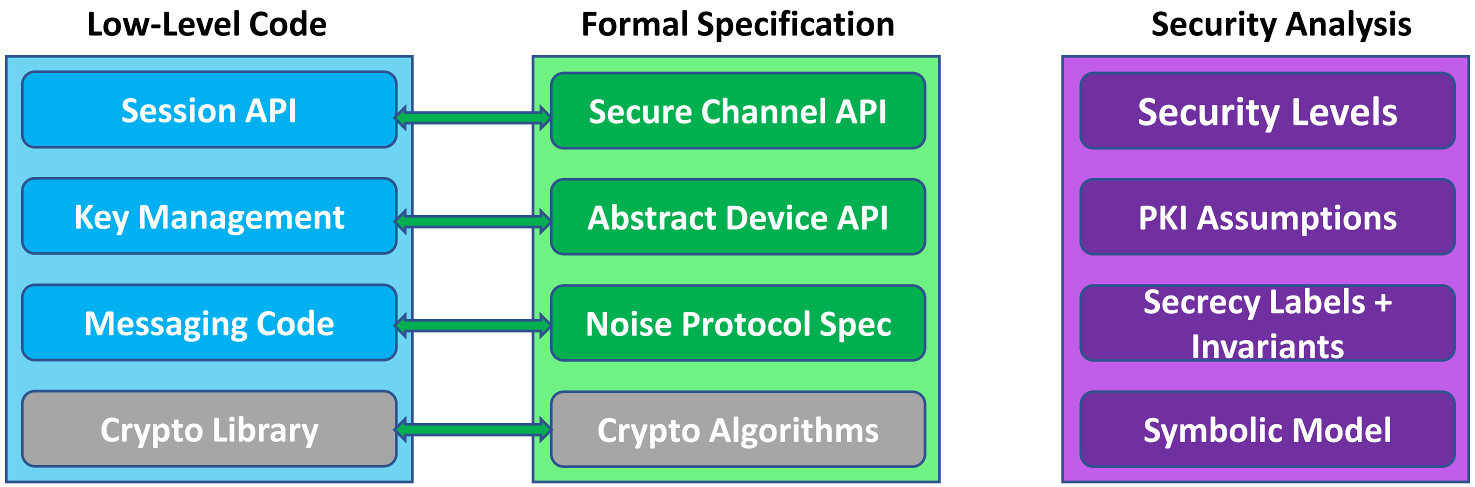
Conclusion

We introduced **Noise***, which is:

- A compiler from Noise protocol patterns to efficient, verified C code, executed in F* normalizer
- A complete, verified library stack exposed through a high-level, defensive API
- A symbolic security analysis generically performed on protocol and API

Noise* showcases techniques useful for:

- **Fully verified software stack**
- **Automated production** of code where we don't sacrifice **precision** or **performance**



Security Analysis - Dolev-Yao*

DY*: framework for symbolic analysis developed in F*.
We do the security analysis **once and for all**.

1. We **add annotations** to types to reflect security properties:

```
// DH signature (simplified)
val dh (l1 : label) (priv : dh_private_key l1)
       (l2 : label) (pub : dh_public_key l2) :
  dh_result (join l1 l2) // label: l1 ⊔ l2
```

Security Analysis - Dolev-Yao*

DY*: framework for symbolic analysis developed in F*.
We do the security analysis **once and for all**.

1. We **add annotations** to types to reflect security properties:

```
// DH signature (simplified)
val dh' (priv : dh_private_key Alice)
        (pub : dh_public_key Bob) :
  dh_result (join Alice Bob) // Alice ⊔ Bob
```

Security Analysis - Dolev-Yao*

DY*: framework for symbolic analysis developed in F*.
We do the security analysis **once and for all**.

1. We **add annotations** to types to reflect security properties:

```
// DH signature (simplified)
val dh' (priv : dh_private_key Alice)
        (pub : dh_public_key Bob) :
  dh_result (join Alice Bob) // Alice ⊔ Bob
```

2. We **generate target labels** for every step of the handshake:

IKpsk2 (from the responder's point of view)

← s

. . .

→ e, es, s, ss, [d] l1 = (peer_eph_label ⊔ CanRead [P p]) ⊓ (. . .)

← e, ee, se, psk, [d] l2 = (peer_eph_label ⊔ CanRead [P p]) ⊓ (. . .) ⊓
(peer_eph_label ⊔ CanRead [S p sid]) ⊓ (. . .)

. . .

Security Analysis - Dolev-Yao*

DY*: framework for symbolic analysis developed in F*.
We do the security analysis **once and for all**.

1. We **add annotations** to types to reflect security properties:

```
// DH signature (simplified)
val dh' (priv : dh_private_key Alice)
        (pub : dh_public_key Bob) :
  dh_result (join Alice Bob) // Alice ⊔ Bob
```

2. We **generate target labels** for every step of the handshake:

IKpsk2 (from the responder's point of view)

← s

...

→ e, es, s, ss, [d]   l1 = (peer_eph_label ⊔ CanRead [P p]) ⊓ (. . .)

← e, ee, se, psk, [d]   l2 = (peer_eph_label ⊔ CanRead [P p]) ⊓ (. . .) ⊓
    (peer_eph_label ⊔ CanRead [S p sid]) ⊓ (. . .)

...

3. We prove that the **handshake state meets** at each stage of the protocol the **corresponding security label**

Security Analysis - Dolev-Yao*

DY*: framework for symbolic analysis developed in F*.
We do the security analysis **once and for all**.

1. We **add annotations** to types to reflect security properties:

```
// DH signature (simplified)
val dh’ (priv : dh_private_key Alice)
        (pub : dh_public_key Bob) :
  dh_result (join Alice Bob) // Alice ⊔ Bob
```

2. We **generate target labels** for every step of the handshake:

IKpsk2 (from the responder’s point of view)

← s

. . .

→ e, es, s, ss, [d]   l1 = (peer_eph_label ⊔ CanRead [P p]) ⊓ (. . .)

← e, ee, se, psk, [d]   l2 = (peer_eph_label ⊔ CanRead [P p]) ⊓ (. . .) ⊓
(peer_eph_label ⊔ CanRead [S p sid]) ⊓ (. . .)

. . .

3. We prove that the **handshake state meets** at each stage of the protocol the **corresponding security label**

4. We prove an **ephemeral authentication invariant** to link the remote ephemeral to the remote static (while remote static is validated by certification function)

Security Analysis - Dolev-Yao*

DY*: framework for symbolic analysis developed in F*.
We do the security analysis **once and for all**.

1. We **add annotations** to types to reflect security properties:

```
// DH signature (simplified)
val dh' (priv : dh_private_key Alice)
        (pub : dh_public_key Bob) :
  dh_result (join Alice Bob) // Alice ⊔ Bob
```

2. We **generate target labels** for every step of the handshake:

IKpsk2 (from the responder's point of view)

← s

...

→ e, es, s, ss, [d] l1 = (peer_eph_label ⊔ CanRead [P p]) ⊓ (. . .)

← e, ee, se, psk, [d] l2 = (peer_eph_label ⊔ CanRead [P p]) ⊓ (. . .) ⊓
(peer_eph_label ⊔ CanRead [S p sid]) ⊓ (. . .)

. . .

3. We prove that the **handshake state meets** at each stage of the protocol the **corresponding security label**

4. We prove an **ephemeral authentication invariant** to link the remote ephemeral to the remote static (while remote static is validated by certification function)

5. We **formalize the Noise security levels** with predicates over labels:

Level	Confidentiality Predicate (over i, idx, and l)
0	⊤
1	can_flow i (CanRead [S idx.p idx.sid] ⊔ idx.peer_eph_label) l
2	can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l
3	can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l ∧ can_flow i (CanRead [S idx.p idx.sid] ⊔ idx.peer_eph_label) l
4	can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l ∧ can_flow i (CanRead [S idx.p idx.sid] ⊔ idx.peer_eph_label) l ∧ (compromised_before i (P idx.p) ∨ compromised_before i (P idx.peer) ∨ (∃sid'. peer_eph_label == CanRead [S idx.peer sid']))
5	can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l ∧ can_flow i (CanRead [S idx.p idx.sid] ⊔ idx.peer_eph_label) l ∧ (compromised_before i (S idx.p idx.sid) ∨ compromised_before i (P idx.peer) ∨ (∃sid'. peer_eph_label == CanRead [S idx.peer sid']))

Level	Authentication Predicate (over i, idx, and l)
0	⊤
1	can_flow i (CanRead [P idx.p; P idx.peer]) l
2	can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l

Security Analysis - Dolev-Yao*

DY*: framework for symbolic analysis developed in F*.
We do the security analysis **once and for all**.

1. We **add annotations** to types to reflect security properties:

```
// DH signature (simplified)
val dh' (priv : dh_private_key Alice)
        (pub : dh_public_key Bob) :
  dh_result (join Alice Bob) // Alice ⊔ Bob
```

2. We **generate target labels** for every step of the handshake:

IKpsk2 (from the responder's point of view)

← s

…

→ e, es, s, ss, [d]   l1 = (peer_eph_label ⊔ CanRead [P p]) ⊓ (. . .)

← e, ee, se, psk, [d]   l2 = (peer_eph_label ⊔ CanRead [P p]) ⊓ (. . .) ⊓
(peer_eph_label ⊔ CanRead [S p sid]) ⊓ (. . .)

. . .

3. We prove that the **handshake state meets** at each stage of the protocol the **corresponding security label**

4. We prove an **ephemeral authentication invariant** to link the remote ephemeral to the remote static (while remote static is validated by certification function)

5. We **formalize the Noise security levels** with predicates over labels:

Level	Confidentiality Predicate (over i, idx, and l)
0	⊤
1	can_flow i (CanRead [S idx.p idx.sid] ⊔ idx.peer_eph_label) l
2	can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l
3	can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l ∧ can_flow i (CanRead [S idx.p idx.sid] ⊔ idx.peer_eph_label) l
4	can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l ∧ can_flow i (CanRead [S idx.p idx.sid] ⊔ idx.peer_eph_label) l ∧ (compromised_before i (P idx.p) ∨ compromised_before i (P idx.peer) ∨ (∃sid'. peer_eph_label == CanRead [S idx.peer sid']))
5	can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l ∧ can_flow i (CanRead [S idx.p idx.sid] ⊔ idx.peer_eph_label) l ∧ (compromised_before i (S idx.p idx.sid) ∨ compromised_before i (P idx.peer) ∨ (∃sid'. peer_eph_label == CanRead [S idx.peer sid']))

Level	Authentication Predicate (over i, idx, and l)
0	⊤
1	can_flow i (CanRead [P idx.p; P idx.peer]) l
2	can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l

6. We prove that those **security predicates are satisfied** by the target labels by combining 3. and 4.

Security Analysis - Dolev-Yao*

DY*: framework for symbolic analysis developed in F*.
We do the security analysis **once and for all**.

1. We **add annotations** to types to reflect security properties:

```
// DH signature (simplified)
val dh' (priv : dh_private_key Alice)
        (pub : dh_public_key Bob) :
  dh_result (join Alice Bob) // Alice ⊔ Bob
```

2. We **generate target labels** for every step of the handshake:

IKpsk2 (from the responder's point of view)

← s

...

Strong forward-secrecy

→ e, es, s, ss, [d]   l1 = (peer_eph_label ⊔ CanRead [P p]) ⊓ (. . .)

← e, ee, se, psk, [d]   l2 = (peer_eph_label ⊔ CanRead [P p]) ⊓ (. . .) ⊓

(peer_eph_label ⊔ CanRead [S p sid]) ⊓ (. . .)

. . .

3. We prove that the **handshake state meets** at each stage of the protocol the **corresponding security label**

4. We prove an **ephemeral authentication invariant** to link the remote ephemeral to the remote static (while remote static is validated by certification function)

5. We **formalize the Noise security levels** with predicates over labels:

Level	Confidentiality Predicate (over i, idx, and l)
0	⊤
1	can_flow i (CanRead [S idx.p idx.sid] ⊔ idx.peer_eph_label) l
2	can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l
3	can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l ∧ can_flow i (CanRead [S idx.p idx.sid] ⊔ idx.peer_eph_label) l
4	can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l ∧ can_flow i (CanRead [S idx.p idx.sid] ⊔ idx.peer_eph_label) l ∧ (compromised_before i (P idx.p) ∨ compromised_before i (P idx.peer) ∨ (∃sid'. peer_eph_label == CanRead [S idx.peer sid']))
5	can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l ∧ can_flow i (CanRead [S idx.p idx.sid] ⊔ idx.peer_eph_label) l ∧ (compromised_before i (S idx.p idx.sid) ∨ compromised_before i (P idx.peer) ∨ (∃sid'. peer_eph_label == CanRead [S idx.peer sid']))

Level	Authentication Predicate (over i, idx, and l)
0	⊤
1	can_flow i (CanRead [P idx.p; P idx.peer]) l
2	can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l

6. We prove that those **security predicates are satisfied** by the target labels by combining 3. and 4.

Security Analysis - Dolev-Yao*

DY*: framework for symbolic analysis developed in F*

Define labels for the data-types:

- `CanRead [P p]` : only principal p can read
- `CanRead [S p sid]` : only principal p at session `sid`

Annotate the data types to give them usages and labels:

- `dh_private_key l` : private key of label `l`
- `dh_public_key l` : public key associated to a private key of label `l`

```
// DH signature (simplified)
val dh (l1 : label) (priv : dh_private_key l1)
       (l2 : label) (pub : dh_public_key l2) :
  dh_result (join l1 l2) // l1 ⊔ l2
```

Security Analysis - Example

Alice **Bob**

→ e, es, s, ss

```
let ck0 = hash "Noise_IKpsk2_25519_..." in
// e
...
// es
let dh_es = dh e rs in
let ck1, sk1 = kdf2 ck1 dh_es in
// s
...
// ss
let dh_ss = dh s rs in
let ck2, sk2 = kdf2 ck2 dh_ss in
// d (plain text)
let cipher = aead_encrypt sk2 ... plain in
...
// Output
... @ cipher
```

Alice must have the following keys:

```
e: dh_private (CanRead [S "Alice" sn])
s: dh_private (CanRead [P "Alice"])
rs: dh_public (CanRead [P "Bob"])
```

```
// DH signature (simplified)
val dh (l1 : label) (priv : dh_private_key l1)
       (l2 : label) (pub : dh_public_key l2) :
  dh_result (join l1 l2) // label: l1 ⊔ l2
```

All annotations are syntactic (labels are syntactic)

```
dh_es: dh_result ((CanRead [S "Alice" sn]) ⊔ (CanRead [P "Bob"]))

dh_ss: dh_result ((CanRead [P "Alice"])    ⊔ (CanRead [P "Bob"]))
```

Security Analysis - Example

```
let ck0 = hash "Noise_IKpsk2_..." in
// e
...
// es
let dh_es = dh e rs in
let ck1, sk1 = kdf2 ck0 dh_es in
// s
...
// ss
let dh_ss = dh s rs in
let ck2, sk2 = kdf2 ck2 dh_ss in
// d (plain text)
let cipher =
  aead_encrypt sk2 ... plain
in
...
// Output
... @ cipher
```

```
l_es := ((CanRead [S "Alice" sn]) ⊔ (CanRead [P "Bob"]))
l_ss := ((CanRead [P "Alice"])    ⊔ (CanRead [P "Bob"]))
```

```
dh_es : dh_result l_es
dh_ss : dh_result l_ss
```

```
ck0 : chaining_key public
ck1 : chaining_key (public ⊓ l_es)
ck2 : chaining_key ((public ⊓ l_es) ⊓ l_ss)
```

```
val aead_encrypt
 ➡(sk : aead_key l)    // encryption key
   (iv : msg public)    // nonce
 ➡(plain : msg l)      // plaintext
   (ad : msg public) : // authentication data
   msg public
```

We can then send the encrypted message: insert a **Send** even in a global trace

Security Analysis – Target Labels

IKpsk2 (from the responder's point of view)

← s

…

→ e, es, s, ss, [d]   (peer_eph_label ⊔ CanRead [P p]) ⊓ (CanRead [P peer] ⊔ CanRead [P p])

← e, ee, se, psk, [d]   (peer_eph_label ⊔ CanRead [P p]) ⊓ (CanRead [P peer] ⊔ CanRead [P p]) ⊓ (peer_eph_label ⊔CanRead [S p sid]) ⊓ (CanRead [P peer] ⊔ CanRead [S p sid]) ⊓ (CanRead [P peer] ⊔ CanRead [P p])

← [d]

→ [d]

← [d]

Security Analysis – Target Labels

IKpsk2 (from the responder's point of view)

← s

…

→ e, es, s, ss, [d] (peer_eph_label ⊔ CanRead [P p]) ⊓ (CanRead [P peer] ⊔ CanRead [P p])

← e, ee, se, psk, [d] (peer_eph_label ⊔ CanRead [P p]) ⊓ (CanRead [P peer] ⊔ CanRead [P p]) ⊓
(peer_eph_label ⊔CanRead [S p sid]) ⊓ (CanRead [P peer] ⊔ CanRead [S p sid]) ⊓ (CanRead [P peer] ⊔ CanRead [P p])

← [d]

→ [d]

← [d]

Security Analysis – Target Labels

IKpsk2 (from the responder's point of view)

← s

Responder learns information about the initiator's ephemeral by linking it to the initiator's static key

…

→ e, es, s, ss, [d] (peer_eph_label ⊔ CanRead [P p]) ⊓ (CanRead [P peer] ⊔ CanRead [P p])

← e, ee, se, psk, [d] (peer_eph_label ⊔ CanRead [P p]) ⊓ (CanRead [P peer] ⊔ CanRead [P p]) ⊓ (peer_eph_label ⊔CanRead [S p sid]) ⊓ (CanRead [P peer] ⊔ CanRead [S p sid]) ⊓ (CanRead [P peer] ⊔ CanRead [P p])

← [d]

→ [d]

← [d]

Security Analysis – Target Labels

IKpsk2 (from the responder's point of view)

← s

…

→ e, es, s, ss, [d]

← e, ee, se, psk, [d]

← [d]

→ [d]

← [d]

Responder learns information about the initiator's ephemeral by linking it to the initiator's static key

(peer_eph_label ⊔ CanRead [P p]) ⊓ (CanRead [P peer] ⊔ CanRead [P p])

(peer_eph_label ⊔ CanRead [P p]) ⊓ (CanRead [P peer] ⊔ CanRead [P p]) ⊓
(peer_eph_label ⊔CanRead [S p sid]) ⊓ (CanRead [P peer] ⊔ CanRead [S p sid]) ⊓ (CanRead [P peer] ⊔ CanRead [P p])

Authentication invariant gives us information upon receiving messages

Upon receiving a message: we get information from the **current label** and previously used keys.
The authentication invariant is verified whenever encrypting data (and retrieved when decrypting).

Security Analysis: Annotating the Types

"Computational" Definitions

```
type cipher_state = {
  sk : option aead_key;
  n : nat;
}
```

```
type symmetric_state (nc : config) = {
  c_state : cipher_state;
  ck : chaining_key nc;
  h : hash nc;
}
```

```
type handshake_state (nc : config) = {
  sym_state : symmetric_state nc;
  static : option (keypair nc);
  ephemeral : option (keypair nc);
  remote_static : option (public_key nc);
  remote_ephemeral : option (public_key nc);
  preshared : option preshared_key;
}
```

Symbolic Definitions

```
type cipher_state (i:nat) (l:label) = {
  k : option (aead_key i l);
  n: nat;
}
```

```
type symmetric_state (nc:config) (i:nat) (l:label) = {
  c_state : cipher_state i l;
  ck : chaining_key nc i l;
  h : hash nc i;
  ...; // Omitted ghost fields
}
```

```
type handshake_state (nc:config) (i:nat) (l:label) (idx:index) = {
  sym_state : symmetric_state nc i l idx.p idx.si;
  static : option (static_keypair nc i (readers [P idx.p]));
  ephemeral : option (ephemeral_keypair nc i (readers [S p idx.si]));
  remote_static : option (not_validated_key nc i); // Externally validated
  remote_ephemeral : option (not_validated_key nc i); // Externally validated
  preshared : psk:option (preshared_key i idx.p idx.peer);
}
```

We then rewrite the protocol and the API specification functions by using those annotated types (and symbolic functions)

Security Analysis: `can_flow`

- Labels are purely syntactic
- Semantics of DY* are given through a `can_flow` predicate which states properties about a global trace of events

- Labels can flow to more secret labels (i is a timestamp):

```
can_flow i (CanRead [P p1]) (CanRead [P p1] ⊓ CanRead [P p2])
```

- The attacker can **dynamically compromise** a participant's current state: event `Compromise p sid`
- If a label is compromised, it flows to `public` (actually an equivalence):

```
compromised_before i (P p) ==> can_flow (CanRead [P p]) public
compromised_before i (S p sid) ==> can_flow (CanRead [S p sid]) public
...
```

Security Analysis : Noise Security Goals

"Strong Forward Secrecy" (level 5)

Level	Confidentiality Predicate (over i, idx, and l)
0	$\top$
1	can_flow i (CanRead [S idx.p idx.sid] $\sqcup$ idx.peer_eph_label) l
2	can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l
3	can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l $\wedge$ can_flow i (CanRead [S idx.p idx.sid] $\sqcup$ idx.peer_eph_label) l
4	can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l $\wedge$ can_flow i (CanRead [S idx.p idx.sid] $\sqcup$ idx.peer_eph_label) l $\wedge$ (compromised_before i (P idx.p) $\vee$ compromised_before i (P idx.peer) $\vee$ ($\exists$sid'. peer_eph_label == CanRead [S idx.peer sid']))
5	can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l $\wedge$ can_flow i (CanRead [S idx.p idx.sid] $\sqcup$ idx.peer_eph_label) l $\wedge$ (compromised_before i (S idx.p idx.sid) $\vee$ compromised_before i (P idx.peer) $\vee$ ($\exists$sid'. peer_eph_label == CanRead [S idx.peer sid']))

Level	Authentication Predicate (over i, idx, and l)
0	$\top$
1	can_flow i (CanRead [P idx.p; P idx.peer]) l
2	can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l

Security Analysis : API

- **Authentication** (slightly simplified):

```
// Slightly simplified:
val unpack_message_with_auth_level :
  #mi:mindex ->
  alevel:auth_level -> // ← Requested authentication level
  msg:encap_message mi ->
  Pure (option (lvl_bytes mi.i mi.l)) (requires True)
  (ensures (fun res ->
    match res with
    | Some b -> expected_auth_label alevel mi mi.l // ← Security guarantees
    | None -> True
   ))
```

- **Confidentiality:** slightly more subtle, but information can be exposed

Linked Lists and Automated Framing

- In C: high-level comments discouraging from looking up peers and performing peer removal at the same time
- In F*: possible to formally reason about peer disjointness

```
// Peer insertion (simplified)
val device_add_peer dv peer_name rs psk : ST (peer_p idc)
  (requires (fun h0 -> ...))
  (ensures (fun h0 p h1 ->

    peer_invariant h1 p dv /\
    loc_includes (device_footprint dv) (peer_footprint p) /\

    peer_get_id h1 p = device_get_peers_counter h0 dv /\
    device_get_peers_counter h1 dv = device_get_peers_counter h0 dv + 1 /\

    device_no_removal dv h0 h1 /\
    ...))
```

```
// Peer deletion (simplified)
val device_remove_peer dv pid : ST unit
  (requires (fun h0 -> ...))
  (ensures (fun h0 p h1 ->
    device_removed_peer dv pid h0 h1 /\
    ...)
```

Linked Lists and Automated Framing

- In C: high-level comments discouraging from looking up peers and performing peer removal at the same time
- In F*: possible to formally reason about peer disjointness

```
// Peer insertion (simplified)
val device_add_peer dv peer_name rs psk : ST (peer_p idc)
  (requires (fun h0 -> ...))
  (ensures (fun h0 p h1 ->
                                     Peer belongs to device
    peer_invariant h1 p dv /\
    loc_includes (device_footprint dv) (peer_footprint p) /\

    peer_get_id h1 p = device_get_peers_counter h0 dv /\
    device_get_peers_counter h1 dv = device_get_peers_counter h0 dv + 1 /\

    device_no_removal dv h0 h1 /\
    ...))
```

```
// Peer deletion (simplified)
val device_remove_peer dv pid : ST unit
  (requires (fun h0 -> ...))
  (ensures (fun h0 p h1 ->
    device_removed_peer dv pid h0 h1 /\
    ...)
```

Linked Lists and Automated Framing

- In C: high-level comments discouraging from looking up peers and performing peer removal at the same time
- In F*: possible to formally reason about peer disjointness

```
// Peer insertion (simplified)
val device_add_peer dv peer_name rs psk : ST (peer_p idc)
  (requires (fun h0 -> ...))
  (ensures (fun h0 p h1 ->
    peer_invariant h1 p dv /\
    loc_includes (device_footprint dv) (peer_footprint p) /\
    peer_get_id h1 p = device_get_peers_counter h0 dv /\
    device_get_peers_counter h1 dv = device_get_peers_counter h0 dv + 1 /\
    device_no_removal dv h0 h1 /\
    ...))
```

Peer belongs to device

Peer has unique id

```
// Peer deletion (simplified)
val device_remove_peer dv pid : ST unit
  (requires (fun h0 -> ...))
  (ensures (fun h0 p h1 ->
    device_removed_peer dv pid h0 h1 /\
    ...)
```

Linked Lists and Automated Framing

- In C: high-level comments discouraging from looking up peers and performing peer removal at the same time
- In F*: possible to formally reason about peer disjointness

```
// Peer insertion (simplified)
val device_add_peer dv peer_name rs psk : ST (peer_p idc)
  (requires (fun h0 -> ...))
  (ensures (fun h0 p h1 ->
    peer_invariant h1 p dv /\
    loc_includes (device_footprint dv) (peer_footprint p) /\
    peer_get_id h1 p = device_get_peers_counter h0 dv /\
    device_get_peers_counter h1 dv = device_get_peers_counter h0 dv + 1 /\
    device_no_removal dv h0 h1 /\
    ...))
```

Peer belongs to device

Peer has unique id

No peer removed

```
// Peer deletion (simplified)
val device_remove_peer dv pid : ST unit
  (requires (fun h0 -> ...))
  (ensures (fun h0 p h1 ->
    device_removed_peer dv pid h0 h1 /\
    ...)
```

Linked Lists and Automated Framing

- In C: high-level comments discouraging from looking up peers and performing peer removal at the same time
- In F*: possible to formally reason about peer disjointness

```
// Peer insertion (simplified)
val device_add_peer dv peer_name rs psk : ST (peer_p idc)
  (requires (fun h0 -> ...))
  (ensures (fun h0 p h1 ->
    peer_invariant h1 p dv /\
    loc_includes (device_footprint dv) (peer_footprint p) /\
    peer_get_id h1 p = device_get_peers_counter h0 dv /\
    device_get_peers_counter h1 dv = device_get_peers_counter h0 dv + 1 /\
    device_no_removal dv h0 h1 /\
    ...))
```

Peer belongs to device

Peer has unique id

No peer removed

```
// Peer deletion (simplified)
val device_remove_peer dv pid : ST unit
  (requires (fun h0 -> ...))
  (ensures (fun h0 p h1 ->
    device_removed_peer dv pid h0 h1 /\
    ...)
```

Peer with id pid was removed

Linked Lists and Automated Framing

- In C: high-level comments discouraging from looking up peers and performing peer removal at the same time
- In F*: possible to formally reason about peer disjointness

```
// Peer insertion (simplified)
val device_add_peer dv peer_name rs psk : ST (peer_p idc)
  (requires (fun h0 -> ...))
  (ensures (fun h0 p h1 ->
    peer_invariant h1 p dv /\
    loc_includes (device_footprint dv) (peer_footprint p) /\
    peer_get_id h1 p = device_get_peers_counter h0 dv /\
    device_get_peers_counter h1 dv = device_get_peers_counter h0 dv + 1 /\
    device_no_removal dv h0 h1 /\
    ...))
```

Peer belongs to device

Peer has unique id

No peer removed

```
// Peer deletion (simplified)
val device_remove_peer dv pid : ST unit
  (requires (fun h0 -> ...))
  (ensures (fun h0 p h1 ->
    device_removed_peer dv pid h0 h1 /\
    ...)
```

Peer with id pid was removed

"Simple" framing lemma: if modified memory is disjoint from peer, peer is left unchanged

Linked Lists and Automated Framing

- In C: high-level comments discouraging from looking up peers and performing peer removal at the same time
- In F*: possible to formally reason about peer disjointness

```
// Peer insertion (simplified)
val device_add_peer dv peer_name rs psk : ST (peer_p idc)
  (requires (fun h0 -> ...))
  (ensures (fun h0 p h1 ->
    peer_invariant h1 p dv /\
    loc_includes (device_footprint dv) (peer_footprint p) /\
    peer_get_id h1 p = device_get_peers_counter h0 dv /\
    device_get_peers_counter h1 dv = device_get_peers_counter h0 dv + 1 /\
    device_no_removal dv h0 h1 /\
    ...))
```

Peer belongs to device

Peer has unique id

No peer removed

```
// Peer deletion (simplified)
val device_remove_peer dv pid : ST unit
  (requires (fun h0 -> ...))
  (ensures (fun h0 p h1 ->
    device_removed_peer dv pid h0 h1 /\
    ...)
```

Peer with id pid was removed

"Simple" framing lemma: if modified memory is disjoint from peer, peer is left unchanged

Advanced framing lemmas:

```
// Frame lemma (insertion)
val peer_no_removal_frame_invariant p dv h0 h1 : Lemma
  (requires (
    peer_invariant h0 p dvp /\
    device_no_removal dvp h0 h1))
  (ensures (
    peer_invariant h1 p dvp /\
    peer_v h0 p == peer_v h1 p))
  [SMTPat (peer_invariant h0 p dvp);
   SMTPat (device_no_removal dvp h0 h1)]
```

```
// Frame lemma (removal)
  val peer_removed_peer_frame_invariant pid p dvp h0 h1 :
  Lemma
  (requires (
    peer_invariant h0 p dvp /\
    device_removed_peer dvp pid h0 h1 /\
    peer_get_pid h0 p <> Some pid))
  (ensures (
    peer_invariant h1 p dvp /\
    peer_v h0 p == peer_v h1 p))
  [SMTPat (peer_invariant h0 p dvp);
   SMTPat (device_removed_peer dvp pid h0 h1)]
```

Linked Lists and Automated Framing

- In C: high-level comments discouraging from looking up peers and performing peer removal at the same time
- In F*: possible to formally reason about peer disjointness

```
// Peer insertion (simplified)
val device_add_peer dv peer_name rs psk : ST (peer_p idc)
  (requires (fun h0 -> ...))
  (ensures (fun h0 p h1 ->
   peer_invariant h1 p dv /\
   loc_includes (device_footprint dv) (peer_footprint p) /\
   peer_get_id h1 p = device_get_peers_counter h0 dv /\
   device_get_peers_counter h1 dv = device_get_peers_counter h0 dv + 1 /\
   device_no_removal dv h0 h1 /\
   ...))
```

Peer belongs to device

Peer has unique id

No peer removed

```
// Peer deletion (simplified)
val device_remove_peer dv pid : ST unit
  (requires (fun h0 -> ...))
  (ensures (fun h0 p h1 ->
   device_removed_peer dv pid h0 h1 /\
   ...)
```

Peer with id pid was removed

"Simple" framing lemma: if modified memory is disjoint from peer, peer is left unchanged

Advanced framing lemmas:

```
// Frame lemma (insertion)
val peer_no_removal_frame_invariant p dv h0 h1 : Lemma
  (requires (
    peer_invariant h0 p dvp /\
    device_no_removal dvp h0 h1))
  (ensures (
    peer_invariant h1 p dvp /\
    peer_v h0 p == peer_v h1 p))
  [SMTPat (peer_invariant h0 p dvp);
   SMTPat (device_no_removal dvp h0 h1)]
```

```
// Frame lemma (removal)
  val peer_removed_peer_frame_invariant pid p dvp h0 h1 :
  Lemma
  (requires (
    peer_invariant h0 p dvp /\
    device_removed_peer dvp pid h0 h1 /\
    peer_get_pid h0 p <> Some pid))
  (ensures (
    peer_invariant h1 p dvp /\
    peer_v h0 p == peer_v h1 p))
  [SMTPat (peer_invariant h0 p dvp);
   SMTPat (device_removed_peer dvp pid h0 h1)]
```

Linked Lists and Automated Framing

- In C: high-level comments discouraging from looking up peers and performing peer removal at the same time
- In F*: possible to formally reason about peer disjointness

```
// Peer insertion (simplified)
val device_add_peer dv peer_name rs psk : ST (peer_p idc)
  (requires (fun h0 -> ...))
  (ensures (fun h0 p h1 ->
    peer_invariant h1 p dv /\
    loc_includes (device_footprint dv) (peer_footprint p) /\
    peer_get_id h1 p = device_get_peers_counter h0 dv /\
    device_get_peers_counter h1 dv = device_get_peers_counter h0 dv + 1 /\
    device_no_removal dv h0 h1 /\
    ...))
```

Peer belongs to device

Peer has unique id

No peer removed

```
// Peer deletion (simplified)
val device_remove_peer dv pid : ST unit
  (requires (fun h0 -> ...))
  (ensures (fun h0 p h1 ->
    device_removed_peer dv pid h0 h1 /\
    ...)
```

Peer with id pid was removed

"Simple" framing lemma: if modified memory is disjoint from peer, peer is left unchanged

Advanced framing lemmas:

```
// Frame lemma (insertion)
val peer_no_removal_frame_invariant p dv h0 h1 : Lemma
  (requires (
    peer_invariant h0 p dvp /\
    device_no_removal dvp h0 h1))
  (ensures (
    peer_invariant h1 p dvp /\
    peer_v h0 p == peer_v h1 p))
  [SMTPat (peer_invariant h0 p dvp);
   SMTPat (device_no_removal dvp h0 h1)]
```

```
// Frame lemma (removal)
  val peer_removed_peer_frame_invariant pid p dvp h0 h1 :
  Lemma
  (requires (
    peer_invariant h0 p dvp /\
    device_removed_peer dvp pid h0 h1 /\
    peer_get_pid h0 p <> Some pid))
  (ensures (
    peer_invariant h1 p dvp /\
    peer_v h0 p == peer_v h1 p))
  [SMTPat (peer_invariant h0 p dvp);
   SMTPat (device_removed_peer dvp pid h0 h1)]
```

Linked Lists and Automated Framing

- In C: high-level comments discouraging from looking up peers and performing peer removal at the same time
- In F*: possible to formally reason about peer disjointness

```
// Peer insertion (simplified)
val device_add_peer dv peer_name rs psk : ST (peer_p idc)
  (requires (fun h0 -> ...))
  (ensures (fun h0 p h1 ->
   peer_invariant h1 p dv /\
   loc_includes (device_footprint dv) (peer_footprint p) /\
   peer_get_id h1 p = device_get_peers_counter h0 dv /\
   device_get_peers_counter h1 dv = device_get_peers_counter h0 dv + 1 /\
   device_no_removal dv h0 h1 /\
   ...))
```

Peer belongs to device

Peer has unique id

No peer removed

```
// Peer deletion (simplified)
val device_remove_peer dv pid : ST unit
  (requires (fun h0 -> ...))
  (ensures (fun h0 p h1 ->
   device_removed_peer dv pid h0 h1 /\
   ...)
```

Peer with id pid was removed

"Simple" framing lemma: if modified memory is disjoint from peer, peer is left unchanged

Advanced framing lemmas:

```
// Frame lemma (insertion)
val peer_no_removal_frame_invariant p dv h0 h1 : Lemma
  (requires (
    peer_invariant h0 p dvp /\
    device_no_removal dvp h0 h1))
  (ensures (
    peer_invariant h1 p dvp /\
    peer_v h0 p == peer_v h1 p))
  [SMTPat (peer_invariant h0 p dvp);
   SMTPat (device_no_removal dvp h0 h1)]
```

SMT Pattern

```
// Frame lemma (removal)
  val peer_removed_peer_frame_invariant pid p dvp h0 h1 :
  Lemma
  (requires (
    peer_invariant h0 p dvp /\
    device_removed_peer dvp pid h0 h1 /\
    peer_get_pid h0 p <> Some pid))
  (ensures (
    peer_invariant h1 p dvp /\
    peer_v h0 p == peer_v h1 p))
  [SMTPat (peer_invariant h0 p dvp);
   SMTPat (device_removed_peer dvp pid h0 h1)]
```

Linked Lists and Automated Framing

- In C: high-level comments discouraging from looking up peers and performing peer removal at the same time
- In F*: possible to formally reason about peer disjointness

```
// Peer insertion (simplified)
val device_add_peer dv peer_name rs psk : ST (peer_p idc)
  (requires (fun h0 -> ...))
  (ensures (fun h0 p h1 ->
    peer_invariant h1 p dv /\
    loc_includes (device_footprint dv) (peer_footprint p) /\
    peer_get_id h1 p = device_get_peers_counter h0 dv /\
    device_get_peers_counter h1 dv = device_get_peers_counter h0 dv + 1 /\
    device_no_removal dv h0 h1 /\
    ...))
```

Peer belongs to device

Peer has unique id

No peer removed

```
// Peer deletion (simplified)
val device_remove_peer dv pid : ST unit
  (requires (fun h0 -> ...))
  (ensures (fun h0 p h1 ->
    device_removed_peer dv pid h0 h1 /\
    ...)
```

Peer with id pid was removed

"Simple" framing lemma: if modified memory is disjoint from peer, peer is left unchanged

Advanced framing lemmas:

```
// Frame lemma (insertion)
val peer_no_removal_frame_invariant p dv h0 h1 : Lemma
  (requires (
    peer_invariant h0 p dvp /\
    device_no_removal dvp h0 h1))
  (ensures (
    peer_invariant h1 p dvp /\
    peer_v h0 p == peer_v h1 p))
  [SMTPat (peer_invariant h0 p dvp);
   SMTPat (device_no_removal dvp h0 h1)]
```

SMT Pattern

```
// Frame lemma (removal)
  val peer_removed_peer_frame_invariant pid p dvp h0 h1 :
  Lemma
  (requires (
    peer_invariant h0 p dvp /\
    device_removed_peer dvp pid h0 h1 /\
    peer_get_pid h0 p <> Some pid))
  (ensures (
    peer_invariant h1 p dvp /\
    peer_v h0 p == peer_v h1 p))
  [SMTPat (peer_invariant h0 p dvp);
   SMTPat (device_removed_peer dvp pid h0 h1)]
```

Linked Lists and Automated Framing

- In C: high-level comments discouraging from looking up peers and performing peer removal at the same time
- In F*: possible to formally reason about peer disjointness

```
// Peer insertion (simplified)
val device_add_peer dv peer_name rs psk : ST (peer_p idc)
  (requires (fun h0 -> ...))
  (ensures (fun h0 p h1 ->
   peer_invariant h1 p dv /\
   loc_includes (device_footprint dv) (peer_footprint p) /\
   peer_get_id h1 p = device_get_peers_counter h0 dv /\
   device_get_peers_counter h1 dv = device_get_peers_counter h0 dv + 1 /\
   device_no_removal dv h0 h1 /\
   ...))
```

Peer belongs to device

Peer has unique id

No peer removed

```
// Peer deletion (simplified)
val device_remove_peer dv pid : ST unit
  (requires (fun h0 -> ...))
  (ensures (fun h0 p h1 ->
   device_removed_peer dv pid h0 h1 /\
   ...)
```

Peer with id pid was removed

"Simple" framing lemma: if modified memory is disjoint from peer, peer is left unchanged

Advanced framing lemmas:

```
// Frame lemma (insertion)
val peer_no_removal_frame_invariant p dv h0 h1 : Lemma
  (requires (
    peer_invariant h0 p dvp /\
    device_no_removal dvp h0 h1))
  (ensures (
    peer_invariant h1 p dvp /\
    peer_v h0 p == peer_v h1 p))
  [SMTPat (peer_invariant h0 p dvp);
   SMTPat (device_no_removal dvp h0 h1)]
```

SMT Pattern

```
// Frame lemma (removal)
  val peer_removed_peer_frame_invariant pid p dvp h0 h1 :
  Lemma
  (requires (
    peer_invariant h0 p dvp /\
    device_removed_peer dvp pid h0 h1 /\
    peer_get_pid h0 p <> Some pid))
  (ensures (
    peer_invariant h1 p dvp /\
    peer_v h0 p == peer_v h1 p))
  [SMTPat (peer_invariant h0 p dvp);
   SMTPat (device_removed_peer dvp pid h0 h1)]
```

Noise Patterns: IKpsk2

IKpsk2:

← s

. . .

→ e, es, s, ss

← e, ee, se, psk

Noise Patterns: IKpsk2

IKpsk2:

← s

. . .

→ e, es, s, ss

← e, ee, se, psk

} Premessages

Noise Patterns: IKpsk2

IKpsk2:

← s

. . .

→ e, es, s, ss

← e, ee, se, psk

Premessages: ← s, . . .

Handshake Messages: → e, es, s, ss and ← e, ee, se, psk

Noise Patterns: IKpsk2

IKpsk2:

← s

. . .

→ e, es, s, ss, [d0]

← e, ee, se, psk, [d1]

Premessages: ← s, . . .

Handshake Messages: → e, es, s, ss, [d0]; ← e, ee, se, psk, [d1]

Noise Patterns: IKpsk2

IKpsk2:	
← s . . .	Premessages
→ e, es, s, ss, [d0] ← e, ee, se, psk, [d1]	Handshake Messages
↔ [d2, d3, …]	Transport

Noise Patterns: IKpsk2

Initiator **Responder**

IKpsk2:

← s . . .	Premessages
→ e, es, s, ss, [d0] ← e, ee, se, psk, [d1]	Handshake Messages
↔ [d2, d3, …]	Transport

Noise Patterns: IKpsk2

Initiator **Responder**

IKpsk2:

← s

. . .

→ e, es, s, ss, [d0]

← e, ee, se, psk, [d1]

↔ [d2, d3, …]

Premessages

Handshake Messages

Transport

Noise Patterns: IKpsk2

Initiator **Responder**

IKpsk2:

← s
...
→ e, es, s, ss, [d0]
← e, ee, se, psk, [d1]
↔ [d2, d3, …]

Premessages

Handshake Messages

Transport

Noise Patterns: IKpsk2

Initiator **Responder**

IKpsk2:

← s
. . .
→ e, es, s, ss, [d0]
← e, ee, se, psk, [d1]
↔ [d2, d3, …]

Premessages

Handshake Messages

Transport

Noise Patterns: IKpsk2

Initiator **Responder**

IKpsk2:

← s
. . .

→ e, es, s, ss, [d0]

← e, ee, se, psk, [d1]

↔ [d2, d3, …]

Premessages: ← s, . . .

Handshake Messages: → e, es, s, ss, [d0]; ← e, ee, se, psk, [d1]

Transport: ↔ [d2, d3, …]

Noise Patterns: IKpsk2

responder sends **S_r_pub** before handshake
(uses Public Key Infrastructure…)

Initiator **Responder**

IKpsk2:

← s
. . .
→ e, es, s, ss, [d0]
← e, ee, se, psk, [d1]
↔ [d2, d3, …]

Premessages: ← s, . . .
Handshake Messages: → e, es, s, ss, [d0]; ← e, ee, se, psk, [d1]
Transport: ↔ [d2, d3, …]

Noise Patterns: IKpsk2

Initial chaining Key

(..., ck0) := initialize(protocol_name, ...);
message := []

(..., ck0) := initialize(protocol_name, ...);

Initiator **Responder**

IKpsk2:

← s
. . .
→ e, es, s, ss, [d0]
← e, ee, se, psk, [d1]
↔ [d2, d3, ...]

Premessages

Handshake Messages

Transport

Noise Patterns: IKpsk2

Initial chaining Key

(..., ck0) := initialize(protocol_name, ...);
message := []

e:
(E_i_priv, E_i_pub) := generate_keypair ()
message := message ++ E_i_pub
... // omitted: hash, nonce

(..., ck0) := initialize(protocol_name, ...);

Initiator **Responder**

IKpsk2:

← s
. . .
→ e, es, s, ss, [d0]
← e, ee, se, psk, [d1]
↔ [d2, d3, ...]

Premessages

Handshake Messages

Transport

Noise Patterns: IKpsk2

Initial chaining Key

```
(...,  ck0) := initialize(protocol_name, ...);
message := []

e:
  (E_i_priv, E_i_pub) := generate_keypair ()
  message := message ++ E_i_pub
  ... // omitted: hash, nonce
```

Initiator **Responder**

```
(..., ck0) := initialize(protocol_name, ...);

e:
  (E_i_pub, message) := read_e (message)
  ...
```

IKpsk2:

← s
. . .
→ e, es, s, ss, [d0]
← e, ee, se, psk, [d1]
↔ [d2, d3, ...]

Premessages

Handshake Messages

Transport

Noise Patterns: IKpsk2

Initial chaining Key

(..., ck0) := initialize(protocol_name, ...);
message := []

e:
(E_i_priv, E_i_pub) := generate_keypair ()
message := message ++ E_i_pub
... // omitted: hash, nonce

es: DH(Ephemeral, Static)
dh0 := DH(E_i_priv, S_r_pub)

Initiator **Responder**

IKpsk2:

← s
. . .
} Premessages

→ e, es, s, ss, [d0]
← e, ee, se, psk, [d1]
} Handshake Messages

↔ [d2, d3, …]
} Transport

(..., ck0) := initialize(protocol_name, ...);

e:
(E_i_pub, message) := read_e (message)
...

Noise Patterns: IKpsk2

Initial chaining Key

(..., ck0) := initialize(protocol_name, ...);
message := []

e:
(E_i_priv, **E_i_pub**) := generate_keypair ()
message := message ++ **E_i_pub**
... // omitted: hash, nonce

es: DH(Ephemeral, Static)
dh0 := DH(E_i_priv, **S_r_pub**)

Initiator **Responder**

IKpsk2:

← s
. . .
} Premessages

→ e, es, s, ss, [d0]
← e, ee, se, psk, [d1]
} Handshake Messages

↔ [d2, d3, ...] } Transport

(..., ck0) := initialize(protocol_name, ...);

e:
(**E_i_pub**, message) := read_e (message)
...

es:
dh0 := DH(S_r_priv, **E_i_pub**)

Noise Patterns: IKpsk2

Initial chaining Key

(..., ck0) := initialize(protocol_name, ...);
message := []

e:
(E_i_priv, E_i_pub) := generate_keypair ()
message := message ++ E_i_pub
... // omitted: hash, nonce

es: DH(Ephemeral, Static)
dh0 := DH(E_i_priv, S_r_pub)

Initiator

Responder

IKpsk2:

← s
. . .
} Premessages

→ e, es, s, ss, [d0]
← e, ee, se, psk, [d1]
} Handshake Messages

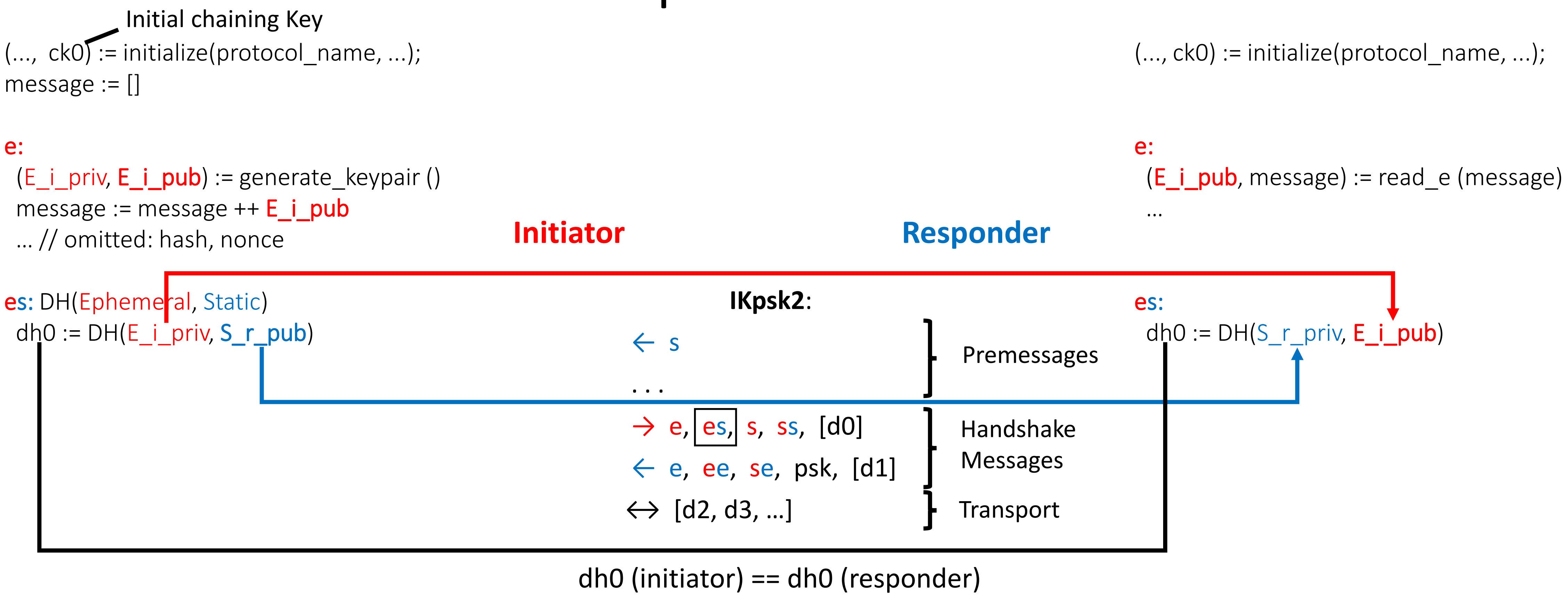
↔ [d2, d3, ...]
} Transport

(..., ck0) := initialize(protocol_name, ...);

e:
(E_i_pub, message) := read_e (message)
...

es:
dh0 := DH(S_r_priv, E_i_pub)

Noise Patterns: IKpsk2



Noise Patterns: IKpsk2

Initial chaining Key

(..., ck0) := initialize(protocol_name, ...);
message := []

e:
(E_i_priv, E_i_pub) := generate_keypair ()
message := message ++ E_i_pub
... // omitted: hash, nonce

es: DH(Ephemeral, Static)
dh0 := DH(E_i_priv, S_r_pub)

Initiator **Responder**

IKpsk2:

← s
. . .
} Premessages

→ e, es, s, ss, [d0]
← e, ee, se, psk, [d1]
} Handshake Messages

↔ [d2, d3, ...] } Transport

(..., ck0) := initialize(protocol_name, ...);

e:
(E_i_pub, message) := read_e (message)
...

es:
dh0 := DH(S_r_priv, E_i_pub)

Noise Patterns: IKpsk2

Initial chaining Key

(..., ck0) := initialize(protocol_name, ...);
message := []

e:
(E_i_priv, E_i_pub) := generate_keypair ()
message := message ++ E_i_pub
... // omitted: hash, nonce

es: DH(Ephemeral, Static)
dh0 := DH(E_i_priv, S_r_pub)
(ck1,sk1) := mix_key(ck0, dh0)
...

Initial chaining key

Diffie-Hellman result

Initiator **Responder**

IKpsk2:

← s
. . .
→ e, es, s, ss, [d0]
← e, ee, se, psk, [d1]
↔ [d2, d3, ...]

Premessages

Handshake Messages

Transport

(..., ck0) := initialize(protocol_name, ...);

e:
(E_i_pub, message) := read_e (message)
...

es:
dh0 := DH(S_r_priv, E_i_pub)
(ck1,sk1) := mix_key(ck0, dh0)
...

Noise Patterns: IKpsk2

Initial chaining Key

(..., ck0) := initialize(protocol_name, ...);
message := []

e:
(E_i_priv, E_i_pub) := generate_keypair ()
message := message ++ E_i_pub
... // omitted: hash, nonce

es: DH(Ephemeral, Static)
dh0 := DH(E_i_priv, S_r_pub)
(ck1,sk1) := mix_key(ck0, dh0)
...

Diffie-Hellman result
Initial chaining key
New chaining key
Symmetric encrypt/decrypt key

Initiator **Responder**

IKpsk2:

← s
...
→ e, es, s, ss, [d0]
← e, ee, se, psk, [d1]
↔ [d2, d3, ...]

Premessages
Handshake Messages
Transport

(..., ck0) := initialize(protocol_name, ...);

e:
(E_i_pub, message) := read_e (message)
...

es:
dh0 := DH(S_r_priv, E_i_pub)
(ck1,sk1) := mix_key(ck0, dh0)
...

Noise Patterns: IKpsk2

Initial chaining Key

(..., ck0) := initialize(protocol_name, ...);
message := []

e:
(E_i_priv, E_i_pub) := generate_keypair ()
message := message ++ E_i_pub
... // omitted: hash, nonce

es: DH(Ephemeral, Static)
dh0 := DH(E_i_priv, S_r_pub)
(ck1,sk1) := mix_key(ck0, dh0)
...

Initiator **Responder**

IKpsk2:

← s
. . .
→ e, es, s, ss, [d0]
← e, ee, se, psk, [d1]
↔ [d2, d3, ...]

Premessages

Handshake Messages

Transport

(..., ck0) := initialize(protocol_name, ...);

e:
(E_i_pub, message) := read_e (message)
...

es:
dh0 := DH(S_r_priv, E_i_pub)
(ck1,sk1) := mix_key(ck0, dh0)
...

Noise Patterns: IKpsk2

Initial chaining Key

(..., ck0) := initialize(protocol_name, ...);
message := []

e:
 (E_i_priv, E_i_pub) := generate_keypair ()
 message := message ++ E_i_pub
 ... // omitted: hash, nonce

es: DH(Ephemeral, Static)
 dh0 := DH(E_i_priv, S_r_pub)
 (ck1,sk1) := mix_key(ck0, dh0)
 ...

s:
 enc_s := aead_encrypt(sk1, ..., S_i_pub)
 message := message ++ enc_s
 ...

Initiator **Responder**

IKpsk2:

← s . . .	Premessages
→ e, es, s, ss, [d0] ← e, ee, se, psk, [d1]	Handshake Messages
↔ [d2, d3, ...]	Transport

(..., ck0) := initialize(protocol_name, ...);

e:
 (E_i_pub, message) := read_e (message)
 ...

es:
 dh0 := DH(S_r_priv, E_i_pub)
 (ck1,sk1) := mix_key(ck0, dh0)
 ...

Noise Patterns: IKpsk2

Initial chaining Key

(..., ck0) := initialize(protocol_name, ...);
message := []

e:
(E_i_priv, E_i_pub) := generate_keypair ()
message := message ++ E_i_pub
... // omitted: hash, nonce

es: DH(Ephemeral, Static)
dh0 := DH(E_i_priv, S_r_pub)
(ck1,sk1) := mix_key(ck0, dh0)
...

s:
enc_s := aead_encrypt(sk1, ..., S_i_pub)
message := message ++ enc_s
...

Initiator **Responder**

IKpsk2:

← s
. . .
Premessages

→ e, es, s, ss, [d0]
← e, ee, se, psk, [d1]
Handshake Messages

↔ [d2, d3, ...]
Transport

(..., ck0) := initialize(protocol_name, ...);

e:
(E_i_pub, message) := read_e (message)
...

es:
dh0 := DH(S_r_priv, E_i_pub)
(ck1,sk1) := mix_key(ck0, dh0)
...

s:
(enc_s, message) := read_enc_s (message)
S_i_pub := aead_decrypt(sk1, ..., enc_s)
...

Uses authentication data: may fail (if message is corrupted or if not same symmetric key)

Noise Patterns: IKpsk2

Initial chaining Key

(..., ck0) := initialize(protocol_name, ...);
message := []

e:
(E_i_priv, E_i_pub) := generate_keypair ()
message := message ++ E_i_pub
... // omitted: hash, nonce

es: DH(Ephemeral, Static)
dh0 := DH(E_i_priv, S_r_pub)
(ck1,sk1) := mix_key(ck0, dh0)
...

s:
enc_s := aead_encrypt(sk1, ..., S_i_pub)
message := message ++ enc_s
...

Initiator **Responder**

IKpsk2:

← s
. . .
} Premessages

→ e, es, s, ss, [d0]
← e, ee, se, psk, [d1]
} Handshake Messages

↔ [d2, d3, ...]
} Transport

(..., ck0) := initialize(protocol_name, ...);

e:
(E_i_pub, message) := read_e (message)
...

es:
dh0 := DH(S_r_priv, E_i_pub)
(ck1,sk1) := mix_key(ck0, dh0)
...

s:
(enc_s, message) := read_enc_s (message)
S_i_pub := aead_decrypt(sk1, ..., enc_s)
...

Noise Patterns: IKpsk2

Initial chaining Key

(..., ck0) := initialize(protocol_name, ...);
message := []

e:
(E_i_priv, E_i_pub) := generate_keypair ()
message := message ++ E_i_pub
... // omitted: hash, nonce

es: DH(Ephemeral, Static)
dh0 := DH(E_i_priv, S_r_pub)
(ck1,sk1) := mix_key(ck0, dh0)
...

s:
enc_s := aead_encrypt(sk1, ..., S_i_pub)
message := message ++ enc_s
...

ss: DH(Static, Static)
dh1 := DH(S_i_priv, S_r_pub)
(ck2,sk2) := mix_key(ck1, dh1)
...

Initiator **Responder**

IKpsk2:

← s
...
} Premessages

→ e, es, s, ss, [d0]
← e, ee, se, psk, [d1]
} Handshake Messages

↔ [d2, d3, ...]
} Transport

(..., ck0) := initialize(protocol_name, ...);

e:
(E_i_pub, message) := read_e (message)
...

es:
dh0 := DH(S_r_priv, E_i_pub)
(ck1,sk1) := mix_key(ck0, dh0)
...

s:
(enc_s, message) := read_enc_s (message)
S_i_pub := aead_decrypt(sk1, ..., enc_s)
...

ss:
dh1 := DH(S_r_priv, S_i_pub)
(ck2,sk2) := mix_key(ck1, dh1)
..

Noise Patterns: IKpsk2

Initial chaining Key

(..., ck0) := initialize(protocol_name, ...);
message := []

e:
(E_i_priv, E_i_pub) := generate_keypair ()
message := message ++ E_i_pub
... // omitted: hash, nonce

es: DH(Ephemeral, Static)
dh0 := DH(E_i_priv, S_r_pub)
(ck1,sk1) := mix_key(ck0, dh0)
...

s:
enc_s := aead_encrypt(sk1, ..., S_i_pub)
message := message ++ enc_s
...

ss: DH(Static, Static)
dh1 := DH(S_i_priv, S_r_pub)
(ck2,sk2) := mix_key(ck1, dh1)
...

[d0]:
cipher := aead_encrypt(sk2, ..., d0)
message := message ++ cipher
...

Initiator **Responder**

IKpsk2:

← s
. . .
→ e, es, s, ss, [d0]
← e, ee, se, psk, [d1]
↔ [d2, d3, ...]

Premessages
Handshake Messages
Transport

(..., ck0) := initialize(protocol_name, ...);

e:
(E_i_pub, message) := read_e (message)
...

es:
dh0 := DH(S_r_priv, E_i_pub)
(ck1,sk1) := mix_key(ck0, dh0)
...

s:
(enc_s, message) := read_enc_s (message)
S_i_pub := aead_decrypt(sk1, ..., enc_s)
...

ss:
dh1 := DH(S_r_priv, S_i_pub)
(ck2,sk2) := mix_key(ck1, dh1)
..

[d0]:
d0 := aead_decrypt(sk2, ..., message)
...
...

Noise Patterns: IKpsk2

Initial chaining Key

(..., ck0) := initialize(protocol_name, ...);
message := []

e:
(E_i_priv, E_i_pub) := generate_keypair ()
message := message ++ E_i_pub
... // omitted: hash, nonce

es: DH(Ephemeral, Static)
dh0 := DH(E_i_priv, S_r_pub)
(ck1,sk1) := mix_key(ck0, dh0)
...

s:
enc_s := aead_encrypt(sk1, ..., S_i_pub)
message := message ++ enc_s
...

ss: DH(Static, Static)
dh1 := DH(S_i_priv, S_r_pub)
(ck2,sk2) := mix_key(ck1, dh1)
...

[d0]:
cipher := aead_encrypt(sk2, ..., d0)
message := message ++ cipher
...

Initiator **Responder**

IKpsk2:

← s
. . .
} Premessages

→ e, es, s, ss, [d0]
← e, ee, se, psk, [d1]
} Handshake Messages

↔ [d2, d3, ...] } Transport

- we exchange keys
- we derive same values for ck, sk at every step

(..., ck0) := initialize(protocol_name, ...);

e:
(E_i_pub, message) := read_e (message)
...

es:
dh0 := DH(S_r_priv, E_i_pub)
(ck1,sk1) := mix_key(ck0, dh0)
...

s:
(enc_s, message) := read_enc_s (message)
S_i_pub := aead_decrypt(sk1, ..., enc_s)
...

ss:
dh1 := DH(S_r_priv, S_i_pub)
(ck2,sk2) := mix_key(ck1, dh1)
..

[d0]:
d0 := aead_decrypt(sk2, ..., message)
...
...

Noise Patterns: IKpsk2

Initiator

(..., ck0) := initialize(protocol_name, ...); ← Initial chaining Key
message := []

e:
 (E_i_priv, E_i_pub) := generate_keypair ()
 message := message ++ E_i_pub
 ... // omitted: hash, nonce

es: DH(Ephemeral, Static)
 dh0 := DH(E_i_priv, S_r_pub)
 (ck1,sk1) := mix_key(ck0, dh0)
 ...

s:
 enc_s := aead_encrypt(sk1, ..., S_i_pub)
 message := message ++ enc_s
 ...

ss: DH(Static, Static)
 dh1 := DH(S_i_priv, S_r_pub)
 (ck2,sk2) := mix_key(ck1, dh1)
 ...

[d0]:
 cipher := aead_encrypt(sk2, ..., d0)
 message := message ++ cipher
 ...

IKpsk2:

← s
. . .
} Premessages

→ e, es, s, ss, [d0]
← e, ee, se, psk, [d1]
} Handshake Messages

↔ [d2, d3, ...]
} Transport

- we exchange keys
- we derive same values for ck, sk at every step

Responder

(..., ck0) := initialize(protocol_name, ...);

e:
 (E_i_pub, message) := read_e (message)
 ...

es:
 dh0 := DH(S_r_priv, E_i_pub)
 (ck1,sk1) := mix_key(ck0, dh0)
 ...

s:
 (enc_s, message) := read_enc_s (message)
 S_i_pub := aead_decrypt(sk1, ..., enc_s)
 ...

ss:
 dh1 := DH(S_r_priv, S_i_pub)
 (ck2,sk2) := mix_key(ck1, dh1)
 ..

[d0]:
 d0 := aead_decrypt(sk2, ..., message)
 ...
 ...

Noise Patterns: IKpsk2

Initial chaining Key

(..., ck0) := initialize(protocol_name, ...);
message := []

e:
(E_i_priv, E_i_pub) := generate_keypair ()
message := message ++ E_i_pub
... // omitted: hash, nonce

es: DH(Ephemeral, Static)
dh0 := DH(E_i_priv, S_r_pub)
(ck1,sk1) := mix_key(ck0, dh0)
...

s:
enc_s := aead_encrypt(sk1, ..., S_i_pub)
message := message ++ enc_s
...

ss: DH(Static, Static)
dh1 := DH(S_i_priv, S_r_pub)
(ck2,sk2) := mix_key(ck1, dh1)
...

[d0]:
cipher := aead_encrypt(sk2, ..., d0)
message := message ++ cipher
...

Initiator **Responder**

IKpsk2:

← s
. . .
} Premessages

→ e, es, s, ss, [d0]
← e, ee, se, psk, [d1]
} Handshake Messages

↔ [d2, d3, ...]
} Transport

- we exchange keys
- we derive same values for ck, sk at every step

(..., ck0) := initialize(protocol_name, ...);

e:
(E_i_pub, message) := read_e (message)
...

es:
dh0 := DH(S_r_priv, E_i_pub)
(ck1,sk1) := mix_key(ck0, dh0)
...

s:
(enc_s, message) := read_enc_s (message)
S_i_pub := aead_decrypt(sk1, ..., enc_s)
...

ss:
dh1 := DH(S_r_priv, S_i_pub)
(ck2,sk2) := mix_key(ck1, dh1)
..

[d0]:
d0 := aead_decrypt(sk2, ..., message)
...
...

Noise Patterns: IKpsk2

Initial chaining Key

Initiator

(..., ck0) := initialize(protocol_name, ...);
message := []

e:
(E_i_priv, E_i_pub) := generate_keypair ()
message := message ++ E_i_pub
... // omitted: hash, nonce

es: DH(Ephemeral, Static)
dh0 := DH(E_i_priv, S_r_pub)
(ck1,sk1) := mix_key(ck0, dh0)
...

s:
enc_s := aead_encrypt(sk1, ..., S_i_pub)
message := message ++ enc_s
...

ss: DH(Static, Static)
dh1 := DH(S_i_priv, S_r_pub)
(ck2,sk2) := mix_key(ck1, dh1)
...

[d0]:
cipher := aead_encrypt(sk2, ..., d0)
message := message ++ cipher
...

IKpsk2:

← s
. . .
} Premessages

→ e, es, s, ss, [d0]
← e, ee, se, psk, [d1]
} Handshake Messages

↔ [d2, d3, ...]
} Transport

- we exchange keys
- we derive same values for ck, sk at every step

Responder

(..., ck0) := initialize(protocol_name, ...);

e:
(E_i_pub, message) := read_e (message)
...

es:
dh0 := DH(S_r_priv, E_i_pub)
(ck1,sk1) := mix_key(ck0, dh0)
...

s:
(enc_s, message) := read_enc_s (message)
S_i_pub := aead_decrypt(sk1, ..., enc_s)
...

ss:
dh1 := DH(S_r_priv, S_i_pub)
(ck2,sk2) := mix_key(ck1, dh1)
..

[d0]:
d0 := aead_decrypt(sk2, ..., message)
...
...

Noise Patterns: IKpsk2

Initial chaining Key

Initiator | **Responder**

Initiator:

```
(..., ck0) := initialize(protocol_name, ...);
message := []

e:
  (E_i_priv, E_i_pub) := generate_keypair ()
  message := message ++ E_i_pub
  ... // omitted: hash, nonce

es: DH(Ephemeral, Static)
  dh0 := DH(E_i_priv, S_r_pub)
  (ck1,sk1) := mix_key(ck0, dh0)
  ...

s:
  enc_s := aead_encrypt(sk1, ..., S_i_pub)
  message := message ++ enc_s
  ...

ss: DH(Static, Static)
  dh1 := DH(S_i_priv, S_r_pub)
  (ck2,sk2) := mix_key(ck1, dh1)
  ...

[d0]:
  cipher := aead_encrypt(sk2, ..., d0)
  message := message ++ cipher
  ...
```

IKpsk2:

← s . . .	Premessages
→ e, es, s, ss, [d0] ← e, ee, se, psk, [d1]	Handshake Messages
↔ [d2, d3, …]	Transport

- we exchange keys
- we derive same values for ck, sk at every step

Responder:

```
(..., ck0) := initialize(protocol_name, ...);

e:
  (E_i_pub, message) := read_e (message)
  ...

es:
  dh0 := DH(S_r_priv, E_i_pub)
  (ck1,sk1) := mix_key(ck0, dh0)
  ...

s:
  (enc_s, message) := read_enc_s (message)
  S_i_pub := aead_decrypt(sk1, ..., enc_s)
  ...

ss:
  dh1 := DH(S_r_priv, S_i_pub)
  (ck2,sk2) := mix_key(ck1, dh1)
  ..

[d0]:
  d0 := aead_decrypt(sk2, ..., message)
  ...
  ...
```

Noise Patterns: IKpsk2

Initial chaining Key

(..., ck0) := initialize(protocol_name, ...);
message := []

e:
(E_i_priv, E_i_pub) := generate_keypair ()
message := message ++ E_i_pub
... // omitted: hash, nonce

es: DH(Ephemeral, Static)
dh0 := DH(E_i_priv, S_r_pub)
(ck1,sk1) := mix_key(ck0, dh0)
...

s:
enc_s := aead_encrypt(sk1, ..., S_i_pub)
message := message ++ enc_s
...

ss: DH(Static, Static)
dh1 := DH(S_i_priv, S_r_pub)
(ck2,sk2) := mix_key(ck1, dh1)
...

[d0]:
cipher := aead_encrypt(sk2, ..., d0)
message := message ++ cipher
...

Initiator **Responder**

IKpsk2:

← s
. . .
} Premessages

→ e, es, s, ss, [d0]
← e, ee, se, psk, [d1]
} Handshake Messages

↔ [d2, d3, ...]
} Transport

- we exchange keys
- we derive same values for ck, sk at every step

(..., ck0) := initialize(protocol_name, ...);

e:
(E_i_pub, message) := read_e (message)
...

es:
dh0 := DH(S_r_priv, E_i_pub)
(ck1,sk1) := mix_key(ck0, dh0)
...

s:
(enc_s, message) := read_enc_s (message)
S_i_pub := aead_decrypt(sk1, ..., enc_s)
...

ss:
dh1 := DH(S_r_priv, S_i_pub)
(ck2,sk2) := mix_key(ck1, dh1)
..

[d0]:
d0 := aead_decrypt(sk2, ..., message)
...
...

Noise Patterns: IKpsk2

Initial chaining Key

(..., ck0) := initialize(protocol_name, ...);
message := []

e:
 (E_i_priv, E_i_pub) := generate_keypair ()
 message := message ++ E_i_pub
 ... // omitted: hash, nonce

es: DH(Ephemeral, Static)
 dh0 := DH(E_i_priv, S_r_pub)
 (ck1,sk1) := mix_key(ck0, dh0)
 ...

s:
 enc_s := aead_encrypt(sk1, ..., S_i_pub)
 message := message ++ enc_s
 ...

ss: DH(Static, Static)
 dh1 := DH(S_i_priv, S_r_pub)
 (ck2,sk2) := mix_key(ck1, dh1)
 ...

[d0]:
 cipher := aead_encrypt(sk2, ..., d0)
 message := message ++ cipher
 ...

Initiator **Responder**

IKpsk2:

← s
. . .
} Premessages

→ e, es, s, ss, [d0]
← e, ee, se, psk, [d1]
} Handshake Messages

↔ [d2, d3, ...]
} Transport

- we exchange keys
- we derive same values for ck, sk at every step

(..., ck0) := initialize(protocol_name, ...);

e:
 (E_i_pub, message) := read_e (message)
 ...

es:
 dh0 := DH(S_r_priv, E_i_pub)
 (ck1,sk1) := mix_key(ck0, dh0)
 ...

s:
 (enc_s, message) := read_enc_s (message)
 S_i_pub := aead_decrypt(sk1, ..., enc_s)
 ...

ss:
 dh1 := DH(S_r_priv, S_i_pub)
 (ck2,sk2) := mix_key(ck1, dh1)
 ..

[d0]:
 d0 := aead_decrypt(sk2, ..., message)
 ...
 ...

Noise Patterns: IKpsk2

Initial chaining Key

(..., ck0) := initialize(protocol_name, ...);
message := []

e:
(E_i_priv, E_i_pub) := generate_keypair ()
message := message ++ E_i_pub
... // omitted: hash, nonce

es: DH(Ephemeral, Static)
dh0 := DH(E_i_priv, S_r_pub)
(ck1 sk1) := mix_key(ck0, dh0)
...

ck1, sk1: DH(E_i, S_r)

s:
enc_s := aead_encrypt(sk1, ..., S_i_pub)
message := message ++ enc_s
...

ss: DH(Static, Static)
dh1 := DH(S_i_priv, S_r_pub)
(ck2 sk2) := mix_key(ck1, dh1)
...

ck2, sk2: DH(E_i, S_r), DH(S_i, S_r)

[d0]:
cipher := aead_encrypt(sk2, ..., d0)
message := message ++ cipher
...

Initiator **Responder**

IKpsk2:

← s
. . .
} Premessages

→ e, es, s, ss, [d0]
← e, ee, se, psk, [d1]
} Handshake Messages

↔ [d2, d3, ...]
} Transport

- we exchange keys
- we derive same values for ck, sk at every step
- ck, sk become “more secret” at every DH

(..., ck0) := initialize(protocol_name, ...);

e:
(E_i_pub, message) := read_e (message)
...

es:
dh0 := DH(S_r_priv, E_i_pub)
(ck1,sk1) := mix_key(ck0, dh0)
...

s:
(enc_s, message) := read_enc_s (message)
S_i_pub := aead_decrypt(sk1, ..., enc_s)
...

ss:
dh1 := DH(S_r_priv, S_i_pub)
(ck2,sk2) := mix_key(ck1, dh1)
...

[d0]:
d0 := aead_decrypt(sk2, ..., message)
...
...