Noise Explorer: IKpsk1

May 8, 2019

1 Message A

1.1 Message Pattern Analysis

Message `A` is the first message in the `IKpsk1` Noise Handshake Pattern. It is sent from the initiator to the responder. In this detailed analysis, we attempt to give you some insight into the protocol logic underlying this message. The insight given here does not fully extend down to fully illustrate the exact state transformations conducted by the formal model, but it does describe them at least informally in order to help illustrate how Message `A` affects the protocol.

1.1.1 Sending Message A

In the applied pi calculus, the initiator's process prepares Message `A` using the following function:

```
letfun writeMessage_a(me:principal, them:principal, hs:handshakestate, payload:
    ↪ bitstring, sid:sessionid) =
        let (ss:symmetricstate, s:keypair, e:keypair, rs:key, re:key, psk:key,
            ↪ initiator:bool) = handshakestateunpack(hs) in
        let (ne:bitstring, ns:bitstring, ciphertext:bitstring) = (empty, empty,
            ↪ empty) in
        let e = generate_keypair(key_e(me, them, sid)) in
        let ne = key2bit(getpublickey(e)) in
        let ss = mixHash(ss, ne) in
        let ss = mixKey(ss, getpublickey(e)) in
        let ss = mixKey(ss, dh(e, rs)) in
        let s = generate_keypair(key_s(me)) in
        let (ss:symmetricstate, ns:bitstring) = encryptAndHash(ss, key2bit(
            ↪ getpublickey(s))) in
        let ss = mixKey(ss, dh(s, rs)) in
        let ss = mixKeyAndHash(ss, psk) in
        let (ss:symmetricstate, ciphertext:bitstring) = encryptAndHash(ss,
            ↪ payload) in
        let hs = handshakestatepack(ss, s, e, rs, re, psk, initiator) in
        let message_buffer = concat3(ne, ns, ciphertext) in
        (hs, message_buffer).
```

How each token is processed by the initiator:

- `e`: Signals that the initiator is sending a fresh ephemeral key share as part of this message. This token adds the following state transformations to `writeMessage_a`:
  - `mixHash`, which hashes the new key into the session hash.
- `es`: Signals that the initiator is calculating a Diffie-Hellman shared secret derived from the initiator's ephemeral key and the responder's static key as part of this message. This token adds the following state transformations to `writeMessage_a`:
  - `mixKey`, which calls the HKDF function using, as input, the existing `SymmetricState` key, and `dh(e, rs)`, the Diffie-Hellman share calculated from the initiator's ephemeral key and the responder's static key.
- `s`: Signals that the initiator is sending a static key share as part of this message. This token adds the following state transformations to `writeMessage_a`:
  - `encryptAndHash` is called on the static public key. If any prior Diffie-Hellman shared secret was established between the sender and the recipient, this allows the initiator to communicate their long-term identity with some degree of confidentiality.
- `ss`: Signals that the initiator is calculating a Diffie-Hellman shared secret derived from the initiator's static key and the responder's static key as part of this message. This token adds the following state transformations to `writeMessage_a`:
  - `mixKey`, which calls the HKDF function using, as input, the existing `SymmetricState` key, and `dh(s, rs)`, the Diffie-Hellman share calculated from the initiator's static key and the responder's static key.
- `psk`: Signals that the initiator is calculating a new session secret that adds a pre-shared symmetric key as part of this message. This token adds the following state transformations to `writeMessage_a`:
  - `mixKeyAndHash`, which mixes and hashes the PSK value into the state and then initializes a new state seeded by the result.

Message `A`'s payload, which is modeled as the output of the function `msg_a(initiatorIdentity, responderIdentity, sessionId)`, is encrypted as `ciphertext2`. This invokes the following operations:

- `encryptAndHash`, which performs an authenticated encryption with added data (AEAD) on the payload, with the session hash as the added data (`encryptWithAd`) and `mixHash`, which hashes the encrypted payload into the next session hash.

1.1.2 Receiving Message A

In the applied pi calculus, the initiator's process prepares Message `A` using the following function:

```
letfun readMessage_a(me:principal, them:principal, hs:handshakestate, message:
    ↪ bitstring, sid:sessionid) =
        let (ss:symmetricstate, s:keypair, e:keypair, rs:key, re:key, psk:key,
            ↪ initiator:bool) = handshakestateunpack(hs) in
```

```
let (ne:bitstring, ns:bitstring, ciphertext:bitstring) = deconcat3(
    ↪ message) in
let valid1 = true in
let re = bit2key(ne) in
let ss = mixHash(ss, key2bit(re)) in
let ss = mixKey(ss, re) in
let ss = mixKey(ss, dh(s, re)) in
let (ss:symmetricstate, ne:bitstring, valid1:bool) = decryptAndHash(ss,
    ↪ ns) in
let rs = bit2key(ne) in
let ss = mixKey(ss, dh(s, rs)) in
let ss = mixKeyAndHash(ss, psk) in
let (ss:symmetricstate, plaintext:bitstring, valid2:bool) =
    ↪ decryptAndHash(ss, ciphertext) in
if ((valid1 && valid2) && (rs = getpublickey(generate_keypair(key_s(them
    ↪ ))))) then (
        let hs = handshakestatepack(ss, s, e, rs, re, psk, initiator) in
        (hs, plaintext, true)
).
```

How each token is processed by the responder:

- `e`: Signals that the responder is receiving a fresh ephemeral key share as part of this message. This token adds the following state transformations to `readMessage_a`:
  - `mixHash`, which hashes the new key into the session hash.
- `es`: Signals that the responder is calculating a Diffie-Hellman shared secret derived from the initiator's ephemeral key and the responder's static key as part of this message. This token adds the following state transformations to `readMessage_a`:
  - `mixKey`, which calls the HKDF function using, as input, the existing `SymmetricState` key, and `dh(e, rs)`, the Diffie-Hellman share calculated from the initiator's ephemeral key and the responder's static key.
- `s`: Signals that the responder is receiving a static key share as part of this message. This token adds the following state transformations to `readMessage_a`:
  - `encryptAndHash` is called on the static public key. If any prior Diffie-Hellman shared secret was established between the sender and the recipient, this allows the initiator to communicate their long-term identity with some degree of confidentiality.
- `ss`: Signals that the responder is calculating a Diffie-Hellman shared secret derived from the initiator's static key and the responder's static key as part of this message. This token adds the following state transformations to `readMessage_a`:
  - `mixKey`, which calls the HKDF function using, as input, the existing `SymmetricState` key, and `dh(s, rs)`, the Diffie-Hellman share calculated from the initiator's static key and the responder's static key.
- `psk`: Signals that the responder is calculating a new session secret that adds a pre-shared symmetric key as part of this message. This token adds the following state transformations to `readMessage_a`:

  - `mixKeyAndHash`, which mixes and hashes the PSK value into the state and then initializes a new state seeded by the result.

Message `A`'s payload, which is modeled as the output of the function `msg_a(initiatorIdentity, responderIdentity, sessionId)`, is encrypted as `ciphertext2`. This invokes the following operations:

- `decryptAndHash`, which performs an authenticated decryption with added data (AEAD) on the payload, with the session hash as the added data (`decryptWithAd`) and `mixHash`, which hashes the encrypted payload into the next session hash.

1.1.3 Queries and Results

Message `A` is tested against four authentication queries and five confidentiality queries.

Authentication Grade 1: Passed

```
RESULT event(RecvMsg(bob,alice,stagepack_a(sid_b),m)) ==> event(SendMsg(alice,
↪ c_1342,stagepack_a(sid_a),m)) || (event(LeakS(phase0,alice)) && event(
↪ LeakPsk(phase0,alice,bob))) || (event(LeakS(phase0,bob)) && event(LeakPsk(
↪ phase0,alice,bob))) is true.
```

In this query, we test for *sender authentication* and *message integrity*. If Bob receives a valid message from Alice, then Alice must have sent that message to *someone*, or Alice had their static key and PSK compromised before the session began, or Bob had their static key and PSK compromised before the session began.

Authentication Grade 2: Failed

```
RESULT event(RecvMsg(bob,alice,stagepack_a(sid_b),m)) ==> event(SendMsg(alice,
↪ c_1342,stagepack_a(sid_a),m)) || event(LeakS(phase0,alice)) cannot be
↪ proved.
```

In this query, we test for *sender authentication* and is *Key Compromise Impersonation* resistance. If Bob receives a valid message from Alice, then Alice must have sent that message to *someone*, or Alice had their static key compromised before the session began.

Authentication Grade 3: Passed

```
RESULT event(RecvMsg(bob,alice,stagepack_a(sid_b),m)) ==> event(SendMsg(alice,
↪ bob,stagepack_a(sid_a),m)) || (event(LeakS(phase0,alice)) && event(LeakPsk
↪ (phase0,alice,bob))) || (event(LeakS(phase0,bob)) && event(LeakPsk(phase0,
↪ alice,bob))) is true.
```

In this query, we test for *sender and receiver authentication* and *message integrity*. If Bob receives a valid message from Alice, then Alice must have sent that message to *Bob specifically*, or Alice had their static key and PSK compromised before the session began, or Bob had their static key and PSK compromised before the session began.

Authentication Grade 4: Failed

```
RESULT event(RecvMsg(bob,alice,stagepack_a(sid_b),m)) ==> event(SendMsg(alice,
↪ bob,stagepack_a(sid_a),m)) || event(LeakS(phase0,alice)) cannot be proved.
```

In this query, we test for *sender and receiver authentication* and is *Key Compromise Impersonation* resistance. If Bob receives a valid message from Alice, then Alice must have sent that message to *Bob specifically*, or Alice had their static key compromised before the session began.

Confidentiality Grade 1: Passed

```
RESULT attacker_p1(msg_a(alice,bob,sid_a)) ==> (event(LeakS(phase0,bob)) &&
↪ event(LeakPsk(phase0,alice,bob))) || (event(LeakS(phase0,bob)) && event(
↪ LeakPsk(phase1,alice,bob))) || (event(LeakS(phase1,bob)) && event(LeakPsk(
↪ phase0,alice,bob))) || (event(LeakS(phase1,bob)) && event(LeakPsk(phase1,
↪ alice,bob))) is true.
```

In this query, we test for *message secrecy* by checking if a passive attacker is able to retrieve the payload plaintext only by compromising Bob's static key and PSK either before or after the protocol session.

Confidentiality Grade 2: Passed

```
RESULT attacker_p1(msg_a(alice,bob,sid_a)) ==> (event(LeakS(phase0,bob)) &&
↪ event(LeakPsk(phase0,alice,bob))) || (event(LeakS(phase0,bob)) && event(
↪ LeakPsk(phase1,alice,bob))) || (event(LeakS(phase1,bob)) && event(LeakPsk(
↪ phase0,alice,bob))) || (event(LeakS(phase1,bob)) && event(LeakPsk(phase1,
↪ alice,bob))) is true.
```

In this query, we test for *message secrecy* by checking if an active attacker is able to retrieve the payload plaintext only by compromising Bob's static key and PSK either before or after the protocol session.

Confidentiality Grade 3: Failed

```
RESULT attacker_p1(msg_a(alice,bob,sid_a)) ==> (event(LeakS(phase0,bob)) &&
↪ event(LeakPsk(phase0,alice,bob))) || (event(LeakS(px,bob)) && event(
↪ LeakPsk(py,alice,bob)) && event(LeakS(pz,alice))) cannot be proved.
```

In this query, we test for *forward secrecy* by checking if a passive attacker is able to retrieve the payload plaintext only by compromising Bob's static key and PSK before the protocol session, or after the protocol session along with Alice's static public key (at any time.)

Confidentiality Grade 4: Failed

```
RESULT attacker_p1(msg_a(alice,bob,sid_a)) ==> (event(LeakS(phase0,bob)) &&
↪ event(LeakPsk(phase0,alice,bob))) || (event(LeakS(px,bob)) && event(
↪ LeakPsk(py,alice,bob)) && event(LeakS(pz,alice))) cannot be proved.
```

In this query, we test for *weak forward secrecy* by checking if an active attacker is able to retrieve the payload plaintext only by compromising Bob's static key and PSK before the protocol session, or after the protocol session along with Alice's static public key (at any time.)

Confidentiality Grade 5: Failed

```
RESULT attacker_p1(msg_a(alice,bob,sid_a)) ==> (event(LeakS(phase0,bob)) &&
  ↪ event(LeakPsk(phase0,alice,bob))) cannot be proved.
```

In this query, we test for *strong forward secrecy* by checking if an active attacker is able to retrieve the payload plaintext only by compromising Bob's static key and PSK before the protocol session.

2 Message B

2.1 Message Pattern Analysis

Message `B` is the second message in the `IKpsk1` Noise Handshake Pattern. It is sent from the responder to the initiator. In this detailed analysis, we attempt to give you some insight into the protocol logic underlying this message. The insight given here does not fully extend down to fully illustrate the exact state transformations conducted by the formal model, but it does describe them at least informally in order to help illustrate how Message `B` affects the protocol.

2.1.1 Sending Message B

In the applied pi calculus, the initiator's process prepares Message `B` using the following function:

```
letfun writeMessage_b(me:principal, them:principal, hs:handshakestate, payload:
  ↪ bitstring, sid:sessionid) =
      let (ss:symmetricstate, s:keypair, e:keypair, rs:key, re:key, psk:key,
        ↪ initiator:bool) = handshakestateunpack(hs) in
      let (ne:bitstring, ns:bitstring, ciphertext:bitstring) = (empty, empty,
        ↪ empty) in
      let e = generate_keypair(key_e(me, them, sid)) in
      let ne = key2bit(getpublickey(e)) in
      let ss = mixHash(ss, ne) in
      let ss = mixKey(ss, getpublickey(e)) in
      let ss = mixKey(ss, dh(e, re)) in
      let ss = mixKey(ss, dh(e, rs)) in
      let (ss:symmetricstate, ciphertext:bitstring) = encryptAndHash(ss,
        ↪ payload) in
      let hs = handshakestatepack(ss, s, e, rs, re, psk, initiator) in
      let message_buffer = concat3(ne, ns, ciphertext) in
      let (ssi:symmetricstate, cs1:cipherstate, cs2:cipherstate) = split(ss)
        ↪ in
      (hs, message_buffer, cs1, cs2).
```

How each token is processed by the responder:

- `e`: Signals that the responder is sending a fresh ephemeral key share as part of this message. This token adds the following state transformations to `writeMessage_b`:
  - `mixHash`, which hashes the new key into the session hash.

- `ee`: Signals that the responder is calculating a Diffie-Hellman shared secret derived from the initiator's ephemeral key and the responder's ephemeral key as part of this message. This token adds the following state transformations to `writeMessage_b`:
  - `mixKey`, which calls the HKDF function using, as input, the existing `SymmetricState` key, and `dh(e, re)`, the Diffie-Hellman share calculated from the initiator's ephemeral key and the responder's ephemeral key.
- `se`: Signals that the responder is calculating a Diffie-Hellman shared secret derived from the initiator's static key and the responder's ephemeral key as part of this message. This token adds the following state transformations to `writeMessage_b`:
  - `mixKey`, which calls the HKDF function using, as input, the existing `SymmetricState` key, and `dh(s, re)`, the Diffie-Hellman share calculated from the initiator's static key and the responder's ephemeral key.

If a static public key was communicated as part of this message, it would have been encrypted as `ciphertext1`. However, since the initiator does not communicate a static public key here, that value is left empty.

Message `B`'s payload, which is modeled as the output of the function `msg_a(initiatorIdentity, responderIdentity, sessionId)`, is encrypted as `ciphertext2`. This invokes the following operations:

- `encryptAndHash`, which performs an authenticated encryption with added data (AEAD) on the payload, with the session hash as the added data (`encryptWithAd`) and `mixHash`, which hashes the encrypted payload into the next session hash.

2.1.2 Receiving Message B

In the applied pi calculus, the initiator's process prepares Message `B` using the following function:

```
letfun readMessage_b(me:principal, them:principal, hs:handshakestate, message:
    ↪ bitstring, sid:sessionid) =
        let (ss:symmetricstate, s:keypair, e:keypair, rs:key, re:key, psk:key,
            ↪ initiator:bool) = handshakestateunpack(hs) in
        let (ne:bitstring, ns:bitstring, ciphertext:bitstring) = deconcat3(
            ↪ message) in
        let valid1 = true in
        let re = bit2key(ne) in
        let ss = mixHash(ss, key2bit(re)) in
        let ss = mixKey(ss, re) in
        let ss = mixKey(ss, dh(e, re)) in
        let ss = mixKey(ss, dh(s, re)) in
        let (ss:symmetricstate, plaintext:bitstring, valid2:bool) =
            ↪ decryptAndHash(ss, ciphertext) in
        if ((valid1 && valid2)) then (
                let hs = handshakestatepack(ss, s, e, rs, re, psk, initiator) in
                let (ssi:symmetricstate, cs1:cipherstate, cs2:cipherstate) =
                    ↪ split(ss) in
                (hs, plaintext, true, cs1, cs2)
        ).
```

How each token is processed by the initiator:

- `e`: Signals that the initiator is receiving a fresh ephemeral key share as part of this message. This token adds the following state transformations to `readMessage_b`:
  - `mixHash`, which hashes the new key into the session hash.
- `ee`: Signals that the initiator is calculating a Diffie-Hellman shared secret derived from the initiator's ephemeral key and the responder's ephemeral key as part of this message. This token adds the following state transformations to `readMessage_b`:
  - `mixKey`, which calls the HKDF function using, as input, the existing `SymmetricState` key, and `dh(e, re)`, the Diffie-Hellman share calculated from the initiator's ephemeral key and the responder's ephemeral key.
- `se`: Signals that the initiator is calculating a Diffie-Hellman shared secret derived from the initiator's static key and the responder's ephemeral key as part of this message. This token adds the following state transformations to `readMessage_b`:
  - `mixKey`, which calls the HKDF function using, as input, the existing `SymmetricState` key, and `dh(s, re)`, the Diffie-Hellman share calculated from the initiator's static key and the responder's ephemeral key.

If a static public key was communicated as part of this message, it would have been encrypted as `ciphertext1`. However, since the initiator does not communicate a static public key here, that value is left empty.

Message `B`'s payload, which is modeled as the output of the function `msg_a(initiatorIdentity, responderIdentity, sessionId)`, is encrypted as `ciphertext2`. This invokes the following operations:

- `decryptAndHash`, which performs an authenticated decryption with added data (AEAD) on the payload, with the session hash as the added data (`decryptWithAd`) and `mixHash`, which hashes the encrypted payload into the next session hash.

2.1.3 Queries and Results

Message `B` is tested against four authentication queries and five confidentiality queries.

Authentication Grade 1: Passed

```
RESULT event(RecvMsg(alice,bob,stagepack_b(sid_a),m)) ==> event(SendMsg(bob,
↪ c_1342,stagepack_b(sid_b),m)) || (event(LeakS(phase0,bob)) && event(
↪ LeakPsk(phase0,alice,bob))) || (event(LeakS(phase0,alice)) && event(
↪ LeakPsk(phase0,alice,bob))) is true.
```

In this query, we test for *sender authentication* and *message integrity.* If Alice receives a valid message from Bob, then Bob must have sent that message to *someone*, or Bob had their static key and PSK compromised before the session began, or Alice had their static key and PSK compromised before the session began.

Authentication Grade 2: Passed

```
RESULT event(RecvMsg(alice,bob,stagepack_b(sid_a),m)) ==> event(SendMsg(bob,
  ↪ c_1342,stagepack_b(sid_b),m)) || event(LeakS(phase0,bob)) is true.
```

In this query, we test for *sender authentication* and is *Key Compromise Impersonation* resistance. If Alice receives a valid message from Bob, then Bob must have sent that message to *someone*, or Bob had their static key compromised before the session began.

Authentication Grade 3: Passed

```
RESULT event(RecvMsg(alice,bob,stagepack_b(sid_a),m)) ==> event(SendMsg(bob,
  ↪ alice,stagepack_b(sid_b),m)) || (event(LeakS(phase0,bob)) && event(LeakPsk
  ↪ (phase0,alice,bob))) || (event(LeakS(phase0,alice)) && event(LeakPsk(
  ↪ phase0,alice,bob))) is true.
```

In this query, we test for *sender and receiver authentication* and *message integrity*. If Alice receives a valid message from Bob, then Bob must have sent that message to *Alice specifically*, or Bob had their static key and PSK compromised before the session began, or Alice had their static key and PSK compromised before the session began.

Authentication Grade 4: Passed

```
RESULT event(RecvMsg(alice,bob,stagepack_b(sid_a),m)) ==> event(SendMsg(bob,
  ↪ alice,stagepack_b(sid_b),m)) || event(LeakS(phase0,bob)) is true.
```

In this query, we test for *sender and receiver authentication* and is *Key Compromise Impersonation* resistance. If Alice receives a valid message from Bob, then Bob must have sent that message to *Alice specifically*, or Bob had their static key compromised before the session began.

Confidentiality Grade 1: Passed

```
RESULT attacker_p1(msg_b(bob,alice,sid_b)) ==> (event(LeakS(phase0,alice)) &&
  ↪ event(LeakPsk(phase0,alice,bob))) || (event(LeakS(phase0,alice)) && event(
  ↪ LeakPsk(phase1,alice,bob))) || (event(LeakS(phase1,alice)) && event(
  ↪ LeakPsk(phase0,alice,bob))) || (event(LeakS(phase1,alice)) && event(
  ↪ LeakPsk(phase1,alice,bob))) is true.
```

In this query, we test for *message secrecy* by checking if a passive attacker is able to retrieve the payload plaintext only by compromising Alice's static key and PSK either before or after the protocol session.

Confidentiality Grade 2: Passed

```
RESULT attacker_p1(msg_b(bob,alice,sid_b)) ==> (event(LeakS(phase0,alice)) &&
  ↪ event(LeakPsk(phase0,alice,bob))) || (event(LeakS(phase0,alice)) && event(
  ↪ LeakPsk(phase1,alice,bob))) || (event(LeakS(phase1,alice)) && event(
  ↪ LeakPsk(phase0,alice,bob))) || (event(LeakS(phase1,alice)) && event(
  ↪ LeakPsk(phase1,alice,bob))) is true.
```

In this query, we test for *message secrecy* by checking if an active attacker is able to retrieve the payload plaintext only by compromising Alice's static key and PSK either before or after the protocol session.

Confidentiality Grade 3: Passed

```
RESULT attacker_p1(msg_b(bob,alice,sid_b)) ==> (event(LeakS(phase0,alice)) &&
  ↪ event(LeakPsk(phase0,alice,bob))) || (event(LeakS(px,alice)) && event(
  ↪ LeakPsk(py,alice,bob)) && event(LeakS(pz,bob))) is true.
```

In this query, we test for *forward secrecy* by checking if a passive attacker is able to retrieve the payload plaintext only by compromising Alice's static key and PSK before the protocol session, or after the protocol session along with Bob's static public key (at any time.)

Confidentiality Grade 4: Passed

```
RESULT attacker_p1(msg_b(bob,alice,sid_b)) ==> (event(LeakS(phase0,alice)) &&
  ↪ event(LeakPsk(phase0,alice,bob))) || (event(LeakS(px,alice)) && event(
  ↪ LeakPsk(py,alice,bob)) && event(LeakS(pz,bob))) is true.
```

In this query, we test for *weak forward secrecy* by checking if an active attacker is able to retrieve the payload plaintext only by compromising Alice's static key and PSK before the protocol session, or after the protocol session along with Bob's static public key (at any time.)

Confidentiality Grade 5: Failed

```
RESULT attacker_p1(msg_b(bob,alice,sid_b)) ==> (event(LeakS(phase0,alice)) &&
  ↪ event(LeakPsk(phase0,alice,bob))) cannot be proved.
```

In this query, we test for *strong forward secrecy* by checking if an active attacker is able to retrieve the payload plaintext only by compromising Alice's static key and PSK before the protocol session.

3 Message C

3.1 Message Pattern Analysis

Message `C` is the third message in the `IKpsk1` Noise Handshake Pattern. It is sent from the initiator to the responder. In this detailed analysis, we attempt to give you some insight into the protocol logic underlying this message. The insight given here does not fully extend down to fully illustrate the exact state transformations conducted by the formal model, but it does describe them at least informally in order to help illustrate how Message `C` affects the protocol.

3.1.1 Sending Message C

In the applied pi calculus, the initiator's process prepares Message `C` using the following function:

```
letfun writeMessage_c(me:principal, them:principal, hs:handshakestate, payload:
  ↪ bitstring, sid:sessionid) =
        let (ss:symmetricstate, s:keypair, e:keypair, rs:key, re:key, psk:key,
          ↪ initiator:bool) = handshakestateunpack(hs) in
        let (ne:bitstring, ns:bitstring, ciphertext:bitstring) = (empty, empty,
          ↪ empty) in
        let (ss:symmetricstate, ciphertext:bitstring) = encryptAndHash(ss,
          ↪ payload) in
        let hs = handshakestatepack(ss, s, e, rs, re, psk, initiator) in
```

```
        let message_buffer = concat3(ne, ns, ciphertext) in
        (hs, message_buffer).
```

Since Message `C` contains no tokens, it is considered purely an "AppData" type message meant to transfer encrypted payloads. If a static public key was communicated as part of this message, it would have been encrypted as `ciphertext1`. However, since the initiator does not communicate a static public key here, that value is left empty.

Message `C`'s payload, which is modeled as the output of the function `msg_a(initiatorIdentity, responderIdentity, sessionId)`, is encrypted as `ciphertext2`. This invokes the following operations:

- `encryptAndHash`, which performs an authenticated encryption with added data (AEAD) on the payload, with the session hash as the added data (`encryptWithAd`) and `mixHash`, which hashes the encrypted payload into the next session hash.

3.1.2 Receiving Message C

In the applied pi calculus, the initiator's process prepares Message `C` using the following function:

```
letfun readMessage_c(me:principal, them:principal, hs:handshakestate, message:
    ↪ bitstring, sid:sessionid) =
        let (ss:symmetricstate, s:keypair, e:keypair, rs:key, re:key, psk:key,
            ↪ initiator:bool) = handshakestateunpack(hs) in
        let (ne:bitstring, ns:bitstring, ciphertext:bitstring) = deconcat3(
            ↪ message) in
        let valid1 = true in
        let (ss:symmetricstate, plaintext:bitstring, valid2:bool) =
            ↪ decryptAndHash(ss, ciphertext) in
        if ((valid1 && valid2)) then (
                let hs = handshakestatepack(ss, s, e, rs, re, psk, initiator) in
                (hs, plaintext, true)
        ).
```

Since Message `C` contains no tokens, it is considered purely an "AppData" type message meant to transfer encrypted payloads. If a static public key was communicated as part of this message, it would have been encrypted as `ciphertext1`. However, since the initiator does not communicate a static public key here, that value is left empty.

Message `C`'s payload, which is modeled as the output of the function `msg_a(initiatorIdentity, responderIdentity, sessionId)`, is encrypted as `ciphertext2`. This invokes the following operations:

- `decryptAndHash`, which performs an authenticated decryption with added data (AEAD) on the payload, with the session hash as the added data (`decryptWithAd`) and `mixHash`, which hashes the encrypted payload into the next session hash.

3.1.3 Queries and Results

Message `C` is tested against four authentication queries and five confidentiality queries.

Authentication Grade 1: Passed

```
RESULT event(RecvMsg(bob,alice,stagepack_c(sid_b),m)) ==> event(SendMsg(alice,
    ↪ c_1342,stagepack_c(sid_a),m)) || (event(LeakS(phase0,alice)) && event(
    ↪ LeakPsk(phase0,alice,bob))) || (event(LeakS(phase0,bob)) && event(LeakPsk(
    ↪ phase0,alice,bob))) is true.
```

In this query, we test for *sender authentication* and *message integrity*. If Bob receives a valid message from Alice, then Alice must have sent that message to *someone*, or Alice had their static key and PSK compromised before the session began, or Bob had their static key and PSK compromised before the session began.

Authentication Grade 2: Passed

```
RESULT event(RecvMsg(bob,alice,stagepack_c(sid_b),m)) ==> event(SendMsg(alice,
    ↪ c_1342,stagepack_c(sid_a),m)) || event(LeakS(phase0,alice)) is true.
```

In this query, we test for *sender authentication* and is *Key Compromise Impersonation* resistance. If Bob receives a valid message from Alice, then Alice must have sent that message to *someone*, or Alice had their static key compromised before the session began.

Authentication Grade 3: Passed

```
RESULT event(RecvMsg(bob,alice,stagepack_c(sid_b),m)) ==> event(SendMsg(alice,
    ↪ bob,stagepack_c(sid_a),m)) || (event(LeakS(phase0,alice)) && event(LeakPsk
    ↪ (phase0,alice,bob))) || (event(LeakS(phase0,bob)) && event(LeakPsk(phase0,
    ↪ alice,bob))) is true.
```

In this query, we test for *sender and receiver authentication* and *message integrity*. If Bob receives a valid message from Alice, then Alice must have sent that message to *Bob specifically*, or Alice had their static key and PSK compromised before the session began, or Bob had their static key and PSK compromised before the session began.

Authentication Grade 4: Passed

```
RESULT event(RecvMsg(bob,alice,stagepack_c(sid_b),m)) ==> event(SendMsg(alice,
    ↪ bob,stagepack_c(sid_a),m)) || event(LeakS(phase0,alice)) is true.
```

In this query, we test for *sender and receiver authentication* and is *Key Compromise Impersonation* resistance. If Bob receives a valid message from Alice, then Alice must have sent that message to *Bob specifically*, or Alice had their static key compromised before the session began.

Confidentiality Grade 1: Passed

```
RESULT attacker_p1(msg_c(alice,bob,sid_a)) ==> (event(LeakS(phase0,bob)) &&
    ↪ event(LeakPsk(phase0,alice,bob))) || (event(LeakS(phase0,bob)) && event(
    ↪ LeakPsk(phase1,alice,bob))) || (event(LeakS(phase1,bob)) && event(LeakPsk(
    ↪ phase0,alice,bob))) || (event(LeakS(phase1,bob)) && event(LeakPsk(phase1,
    ↪ alice,bob))) is true.
```

In this query, we test for *message secrecy* by checking if a passive attacker is able to retrieve the payload plaintext only by compromising Bob's static key and PSK either before or after the protocol session.

Confidentiality Grade 2: Passed

```
RESULT attacker_p1(msg_c(alice,bob,sid_a)) ==> (event(LeakS(phase0,bob)) &&
  ↪ event(LeakPsk(phase0,alice,bob))) || (event(LeakS(phase0,bob)) && event(
  ↪ LeakPsk(phase1,alice,bob))) || (event(LeakS(phase1,bob)) && event(LeakPsk(
  ↪ phase0,alice,bob))) || (event(LeakS(phase1,bob)) && event(LeakPsk(phase1,
  ↪ alice,bob))) is true.
```

In this query, we test for *message secrecy* by checking if an active attacker is able to retrieve the payload plaintext only by compromising Bob's static key and PSK either before or after the protocol session.

Confidentiality Grade 3: Passed

```
RESULT attacker_p1(msg_c(alice,bob,sid_a)) ==> (event(LeakS(phase0,bob)) &&
  ↪ event(LeakPsk(phase0,alice,bob))) || (event(LeakS(px,bob)) && event(
  ↪ LeakPsk(py,alice,bob)) && event(LeakS(pz,alice))) is true.
```

In this query, we test for *forward secrecy* by checking if a passive attacker is able to retrieve the payload plaintext only by compromising Bob's static key and PSK before the protocol session, or after the protocol session along with Alice's static public key (at any time.)

Confidentiality Grade 4: Passed

```
RESULT attacker_p1(msg_c(alice,bob,sid_a)) ==> (event(LeakS(phase0,bob)) &&
  ↪ event(LeakPsk(phase0,alice,bob))) || (event(LeakS(px,bob)) && event(
  ↪ LeakPsk(py,alice,bob)) && event(LeakS(pz,alice))) is true.
```

In this query, we test for *weak forward secrecy* by checking if an active attacker is able to retrieve the payload plaintext only by compromising Bob's static key and PSK before the protocol session, or after the protocol session along with Alice's static public key (at any time.)

Confidentiality Grade 5: Passed

```
RESULT attacker_p1(msg_c(alice,bob,sid_a)) ==> (event(LeakS(phase0,bob)) &&
  ↪ event(LeakPsk(phase0,alice,bob))) is true.
```

In this query, we test for *strong forward secrecy* by checking if an active attacker is able to retrieve the payload plaintext only by compromising Bob's static key and PSK before the protocol session.

4 Message D

4.1 Message Pattern Analysis

Message `D` is the fourth message in the `IKpsk1` Noise Handshake Pattern. It is sent from the responder to the initiator. In this detailed analysis, we attempt to give you some insight into the protocol logic underlying this message. The insight given here does not fully extend down to fully illustrate the exact state transformations conducted by the formal model, but it does describe them at least informally in order to help illustrate how Message `D` affects the protocol.

4.1.1 Sending Message D

In the applied pi calculus, the initiator's process prepares Message `D` using the following function:

```
letfun writeMessage_d(me:principal, them:principal, hs:handshakestate, payload:
    ↪ bitstring, sid:sessionid) =
        let (ss:symmetricstate, s:keypair, e:keypair, rs:key, re:key, psk:key,
            ↪ initiator:bool) = handshakestateunpack(hs) in
        let (ne:bitstring, ns:bitstring, ciphertext:bitstring) = (empty, empty,
            ↪ empty) in
        let (ss:symmetricstate, ciphertext:bitstring) = encryptAndHash(ss,
            ↪ payload) in
        let hs = handshakestatepack(ss, s, e, rs, re, psk, initiator) in
        let message_buffer = concat3(ne, ns, ciphertext) in
        (hs, message_buffer).
```

Since Message `D` contains no tokens, it is considered purely an "AppData" type message meant to transfer encrypted payloads. If a static public key was communicated as part of this message, it would have been encrypted as `ciphertext1`. However, since the initiator does not communicate a static public key here, that value is left empty.

Message `D`'s payload, which is modeled as the output of the function `msg_a(initiatorIdentity, responderIdentity, sessionId)`, is encrypted as `ciphertext2`. This invokes the following operations:

- `encryptAndHash`, which performs an authenticated encryption with added data (AEAD) on the payload, with the session hash as the added data (`encryptWithAd`) and `mixHash`, which hashes the encrypted payload into the next session hash.

4.1.2 Receiving Message D

In the applied pi calculus, the initiator's process prepares Message `D` using the following function:

```
letfun readMessage_d(me:principal, them:principal, hs:handshakestate, message:
    ↪ bitstring, sid:sessionid) =
        let (ss:symmetricstate, s:keypair, e:keypair, rs:key, re:key, psk:key,
            ↪ initiator:bool) = handshakestateunpack(hs) in
        let (ne:bitstring, ns:bitstring, ciphertext:bitstring) = deconcat3(
            ↪ message) in
        let valid1 = true in
        let (ss:symmetricstate, plaintext:bitstring, valid2:bool) =
            ↪ decryptAndHash(ss, ciphertext) in
        if ((valid1 && valid2)) then (
                let hs = handshakestatepack(ss, s, e, rs, re, psk, initiator) in
                (hs, plaintext, true)
        ).
```

Since Message `D` contains no tokens, it is considered purely an "AppData" type message meant to transfer encrypted payloads. If a static public key was communicated as part of this message, it would have been encrypted as `ciphertext1`. However, since the initiator does not communicate a static public key here, that value is left empty.

Message `D`'s payload, which is modeled as the output of the function `msg_a(initiatorIdentity, responderIdentity, sessionId)`, is encrypted as `ciphertext2`. This invokes the following operations:

- `decryptAndHash`, which performs an authenticated decryption with added data (AEAD) on the payload, with the session hash as the added data (`decryptWithAd`) and `mixHash`, which hashes the encrypted payload into the next session hash.

4.1.3 Queries and Results

Message `D` is tested against four authentication queries and five confidentiality queries.

Authentication Grade 1: Passed

```
RESULT event(RecvMsg(alice,bob,stagepack_d(sid_a),m)) ==> event(SendMsg(bob,
  ↪ c_1342,stagepack_d(sid_b),m)) || (event(LeakS(phase0,bob)) && event(
  ↪ LeakPsk(phase0,alice,bob))) || (event(LeakS(phase0,alice)) && event(
  ↪ LeakPsk(phase0,alice,bob))) is true.
```

In this query, we test for *sender authentication* and *message integrity.* If Alice receives a valid message from Bob, then Bob must have sent that message to *someone*, or Bob had their static key and PSK compromised before the session began, or Alice had their static key and PSK compromised before the session began.

Authentication Grade 2: Passed

```
RESULT event(RecvMsg(alice,bob,stagepack_d(sid_a),m)) ==> event(SendMsg(bob,
  ↪ c_1342,stagepack_d(sid_b),m)) || event(LeakS(phase0,bob)) is true.
```

In this query, we test for *sender authentication* and is *Key Compromise Impersonation* resistance. If Alice receives a valid message from Bob, then Bob must have sent that message to *someone*, or Bob had their static key compromised before the session began.

Authentication Grade 3: Passed

```
RESULT event(RecvMsg(alice,bob,stagepack_d(sid_a),m)) ==> event(SendMsg(bob,
  ↪ alice,stagepack_d(sid_b),m)) || (event(LeakS(phase0,bob)) && event(LeakPsk
  ↪ (phase0,alice,bob))) || (event(LeakS(phase0,alice)) && event(LeakPsk(
  ↪ phase0,alice,bob))) is true.
```

In this query, we test for *sender and receiver authentication* and *message integrity.* If Alice receives a valid message from Bob, then Bob must have sent that message to *Alice specifically*, or Bob had their static key and PSK compromised before the session began, or Alice had their static key and PSK compromised before the session began.

Authentication Grade 4: Passed

```
RESULT event(RecvMsg(alice,bob,stagepack_d(sid_a),m)) ==> event(SendMsg(bob,
  ↪ alice,stagepack_d(sid_b),m)) || event(LeakS(phase0,bob)) is true.
```

In this query, we test for *sender and receiver authentication* and is *Key Compromise Impersonation* resistance. If Alice receives a valid message from Bob, then Bob must have sent that message to *Alice specifically*, or Bob had their static key compromised before the session began.

Confidentiality Grade 1: Passed

```
RESULT attacker_p1(msg_d(bob,alice,sid_b)) ==> (event(LeakS(phase0,alice)) &&
  ↪ event(LeakPsk(phase0,alice,bob))) || (event(LeakS(phase0,alice)) && event(
  ↪ LeakPsk(phase1,alice,bob))) || (event(LeakS(phase1,alice)) && event(
  ↪ LeakPsk(phase0,alice,bob))) || (event(LeakS(phase1,alice)) && event(
  ↪ LeakPsk(phase1,alice,bob))) is true.
```

In this query, we test for *message secrecy* by checking if a passive attacker is able to retrieve the payload plaintext only by compromising Alice's static key and PSK either before or after the protocol session.

Confidentiality Grade 2: Passed

```
RESULT attacker_p1(msg_d(bob,alice,sid_b)) ==> (event(LeakS(phase0,alice)) &&
  ↪ event(LeakPsk(phase0,alice,bob))) || (event(LeakS(phase0,alice)) && event(
  ↪ LeakPsk(phase1,alice,bob))) || (event(LeakS(phase1,alice)) && event(
  ↪ LeakPsk(phase0,alice,bob))) || (event(LeakS(phase1,alice)) && event(
  ↪ LeakPsk(phase1,alice,bob))) is true.
```

In this query, we test for *message secrecy* by checking if an active attacker is able to retrieve the payload plaintext only by compromising Alice's static key and PSK either before or after the protocol session.

Confidentiality Grade 3: Passed

```
RESULT attacker_p1(msg_d(bob,alice,sid_b)) ==> (event(LeakS(phase0,alice)) &&
  ↪ event(LeakPsk(phase0,alice,bob))) || (event(LeakS(px,alice)) && event(
  ↪ LeakPsk(py,alice,bob)) && event(LeakS(pz,bob))) is true.
```

In this query, we test for *forward secrecy* by checking if a passive attacker is able to retrieve the payload plaintext only by compromising Alice's static key and PSK before the protocol session, or after the protocol session along with Bob's static public key (at any time.)

Confidentiality Grade 4: Passed

```
RESULT attacker_p1(msg_d(bob,alice,sid_b)) ==> (event(LeakS(phase0,alice)) &&
  ↪ event(LeakPsk(phase0,alice,bob))) || (event(LeakS(px,alice)) && event(
  ↪ LeakPsk(py,alice,bob)) && event(LeakS(pz,bob))) is true.
```

In this query, we test for *weak forward secrecy* by checking if an active attacker is able to retrieve the payload plaintext only by compromising Alice's static key and PSK before the protocol session, or after the protocol session along with Bob's static public key (at any time.)

Confidentiality Grade 5: Passed

```
RESULT attacker_p1(msg_d(bob,alice,sid_b)) ==> (event(LeakS(phase0,alice)) &&
  ↪ event(LeakPsk(phase0,alice,bob))) is true.
```

In this query, we test for *strong forward secrecy* by checking if an active attacker is able to retrieve the payload plaintext only by compromising Alice's static key and PSK before the protocol session.