Protocol specification languages: Tamarin + message formats
Protocol verification & my biases

• protocol verification in my everyday: **security protocols at spec level, properties like authentication**
  • e.g., TLS, WPA2, SSH, OCP-Stapling, PKCS#11, etc.
  • I work with Tamarin and ProVerif (protocol verifiers)
    − specifically for security: include Dolev-Yao attacker
    − unbounded sessions
    − no decidability
    − safety properties (authentication, secrecy)
• what I am missing:
  − general purpose verifiers (e.g., functional correctness)
  − program verification (e.g., analysing the implementation)
  − properties on low-level protocols (e.g., network layer)
  − liveness properties (e.g., resistance against DDoS)
Formal Verification

System $\models \varphi$

- Protocol, e.g., TLS, WPA2
- Safety property (authentication, secrecy)
Formal Verification

**System** $\models \varphi$

- Protocol, e.g., TLS, WPA2
- Safety property (authentication, secrecy)
Formal Verification

System \models \varphi

Protocol, e.g., TLS, WPA2

Safety property (authentication, secrecy)

---

Model M

---

Careful reading
Formal Verification

System $\models \varphi$

Protocol, e.g., TLS, WPA2
Safety property (authentication, secrecy)

RFC

Model M

$M \models \varphi$
Tamarin Prover

ProVerif
// Protocol (only initiator)
// Protocol (only initiator)

rule Init_1:
// Protocol (only initiator)

rule Init_1:
    [ Fr(eph), !Ltk($id_init, ltk) ]
// Protocol (only initiator)
rule Init_1:
  [ Fr(eph), !Ltk($id_init, ltk) ]
-->
// Protocol (only initiator)

rule Init_1:
  \[ Fr(eph), !Ltk($id_init, ltk) \]
  \rightarrow
  \[ Init_1($id_init, $id_resp, eph ) \]
// Protocol (only initiator)
rule Init_1:
  [ Fr(eph), !Ltk($id_init, ltk) ]
  -->
  [ Init_1($id_init, $id_resp, eph )
    , Out(<$id_init, $id_resp, 'g' ^ eph, sign(..) ) ]
// Protocol (only initiator)

rule Init_1:
  [ Fr(eph), !Ltk($id_init, ltk) ]
  -->
  [ Init_1($id_init, $id_resp, eph )
  , Out(<$id_init, $id_resp, 'g' ^ eph, sign(…)) ]
// Protocol (only initiator)

rule Init_1:
    [ Fr(eph), !Ltk($id_init, ltk) ]
    -->
    [ Init_1($id_init, $id_resp, eph )
    , Out(<$id_init, $id_resp, 'g' ^ eph, sign(…)) ]

rule Init_2:
// Protocol (only initiator)

rule Init_1:
    [ Fr(eph), !Ltk($id_init, ltk) ]
    -->
    [ Init_1($id_init, $id_resp, eph )
    , Out(<$id_init, $id_resp, 'g' ^ eph, sign(…)) ]

rule Init_2:
    [ Init_1($id_i, $id_r, eph )
    , Out(<$id_init, $id_resp, 'g' ^ eph, sign(…)) ]
// Protocol (only initiator)

rule Init_1:
    [ Fr(eph), !Ltk($id_init, ltk) ]
    -->
    [ Init_1($id_i, $id_r, eph )
      , Out(<$id_init, $id_r, 'g' ^ eph, sign(\_)) ]

rule Init_2:
    [ Init_1( $id_i, $id_r, eph )
      , In(<$id_i, $id_r, Y, signature>) ]
// Protocol (only initiator)

rule Init_1:
    [ Fr(eph), !Ltk($id_init, ltk) ]

--> 
    [ Init_1($id_i, $id_r, eph )
    , Out(<$id_init, $id_resp, 'g' ^ eph, sign(…)) ]

rule Init_2:
    [ Init_1( $id_i, $id_r, eph )
    , In(<$id_i, $id_r, Y, signature>)
    ]
// Protocol (only initiator)

rule Init_1:
  [ Fr(eph), !ltk($id_init, ltk) ]
  -->
  [ Init_1($id_init, $id_resp, eph )
    , Out(<$id_init, $id_resp, 'g' ^ eph, sign(…)) ]

rule Init_2:
  [ Init_1( $id_i, $id_r, eph )
    , In(<$id_i, $id_r, Y, signature>)
    ]
  --[ Eq(verify(…,signature),ok), SessionKey($id_i,$id_r, Y ^ eph) ]->
// Protocol (only initiator)

rule Init_1:
  [ Fr(eph), !Ltk($id_init, ltk) ]
  -->
  [ Init_1($id_init, $id_resp, eph ), Out(<$id_init,$id_resp,'g' ^ eph, sign(…)) ]

rule Init_2:
  [ Init_1( $id_i, $id_r, eph )
  , In(<$id_i, $id_r, Y, signature>)
  ]
  <--[ Eq(verify(…,signature),ok), SessionKey($id_i,$id_r, Y ^ eph) ]>
// Protocol (only initiator)

rule Init_1:
  [ Fr(eph), !Ltk($id_init, ltk) ]
  -->
  [ Init_1($id_init, $id_resp, eph )
    , Out(<$id_init, $id_resp, 'g' ^ eph, sign(…) ) ]

rule Init_2:
  [ Init_1( $id_i, $id_r, eph )
    , In(<$id_i, $id_r, Y, signature>)
    ]
  --[ Eq(verify(…,signature),ok), SessionKey($id_i,$id_r, Y ^ eph ) ]->
    □
// Protocol (only initiator)

rule Init_1:
   [ Fr(eph), !Lt($id_init, ltk) ]
   -->
   [ Init_1($id_init, $id_resp, eph )
      , Out( <$id_init, $id_resp, 'g' ^ eph, sign(…)> ) ]

rule Init_2:
   [ Init_1( $id_i, $id_r, eph )
      , In(<$id_i, $id_r, Y, signature>)
   ]
   --[ Eq(verify(…,signature),ok), SessionKey($id_i,$id_r, Y ^ eph) ]>

// setup
// Protocol (only initiator)

rule Init_1:
  [ Fr(eph), !Ltk($id_init, ltk) ]
  -->
  [ Init_1($id_init, $id_resp, eph )
    , Out(<$id_init, $id_resp, 'g' ^ eph, sign(…) ) ]

rule Init_2:
  [ Init_1( $id_i, $id_r, eph )
    , In(<$id_i, $id_r, Y, signature>)
    ]
  -->[ Eq(verify(…,signature),ok), SessionKey($id_i,$id_r, Y ^ eph ) ]->

// setup

rule Register_pk:
// Protocol (only initiator)

rule Init_1:
  [ Fr(eph), !Ltk($id_init, ltk) ]
  -->
  [ Init_1($id_init, $id_resp, eph ),
     Out(<$id_init, $id_resp, 'g' ^ eph, sign(…)> ) ]

rule Init_2:
  [ Init_1( $id_i, $id_r, eph ),
     In(<$id_i, $id_r, Y, signature>)
   ]
  --[ Eq(verify(…,signature),ok), SessionKey($id_i,$id_r, Y ^ eph) ]->

// setup

rule Register_pk:
  [ Fr(ltk) ]
// Protocol (only initiator)

rule Init_1:
    [ Fr(eph), !Ltk($id_init, ltk) ]
  -->
    [ Init_1($id_init, $id_resp, eph )
      , Out(<$id_init, $id_resp, 'g' ^ eph, sign(…)) ]

rule Init_2:
    [ Init_1( $id_i, $id_r, eph )
      , In(<$id_i, $id_r, Y, signature>)
    ]
  --[ Eq(verify(…,signature),ok), SessionKey($id_i,$id_r, Y ^ eph) ]->
    []

// setup

rule Register_pk:
    [ Fr(ltk) ]
  -->
// Protocol (only initiator)
rule Init_1:
    [ Fr(eph), !Ltk($id_init, ltk) ]
-->[
    Init_1($id_init, $id_resp, eph )
    , Out( <$id_init, $id_resp, 'g' ^ eph, sign(…)> ) ]

rule Init_2:
    [ Init_1( $id_i, $id_r, eph )
      , In(<$id_i, $id_r, Y, signature>)
    ]
    --[ Eq(verify(…,signature),ok), SessionKey($id_i,$id_r, Y ^ eph) ]->
    []

// setup
rule Register_pk:
    [ Fr(ltk) ]
-->[
    !Ltk($A, ltk), !Pk($A, pk(ltk)), Out(pk(ltk)) ]
// Protocol (only initiator)

rule Init_1:
[ Fr(eph), !Ltk($id_init, ltk) ]
-->[
Init_1($id_init, $id_resp, eph),
Out(<$id_init, $id_resp, 'g' ^ eph, sign(…)) ]

rule Init_2:
[ Init_1($id_i, $id_r, eph),
  In(<$id_i, $id_r, Y, signature>) ]
--[ Eq(verify(…,signature), ok), SessionKey($id_i, $id_r, Y ^ eph) ]->

// setup

rule Register_pk:
[ Fr(ltk) ]
-->[
!Ltk($A, ltk), !Pk($A, pk(ltk)), Out(pk(ltk)) ]
// Protocol (only initiator)
rule Init_1:
  [ Fr(eph), !Ltk($id_init, ltk) ]
  -->
  [ Init_1($id_init, $id_resp, eph )
    , Out(<$id_init, $id_resp, 'g' ^ eph, sign(…)) ]

rule Init_2:
  [ Init_1( $id_i, $id_r, eph )
    , In(<$id_i, $id_r, Y, signature>)
  ]
  --[ Eq(verify(…,signature),ok), SessionKey($id_i,$id_r, Y ^ eph) ]->
    []

// setup
rule Register_pk:
  [ Fr(ltk) ]
  -->
  [ !Ltk($A, ltk), !Pk($A, pk(ltk)), Out(pk(ltk)) ]

// Protocol
let Initiator(ltk, id_init, id_resp) =
new eph;
out(<$id_init, $id_resp, 'g' ^ eph, sign(…));
in(…);
let verify(_,signature) = ok then
  event SessionKey(_);

// setup
process
! new ltk; new id;
!( (in(id_resp); Initiator(ltk, id, id_resp))
  | Responder(id,ltk)
  | out(pk(ltk))
)
Message formats

<table>
<thead>
<tr>
<th>Message Formats</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hash(input)</td>
<td>Blake2s(input, 32), returning 32 bytes of output.</td>
</tr>
<tr>
<td>Mac(key, input)</td>
<td>Keyed-Blake2s(key, input, 16), the keyed MAC variant of the BLAKE2s hash function, returning 16 bytes of output.</td>
</tr>
<tr>
<td>Hmac(key, input)</td>
<td>Hmac-Blake2s(key, input, 32), the ordinary BLAKE2s hash function used in an HMAC construction, returning 32 bytes of output.</td>
</tr>
<tr>
<td>Kdf(key, input)</td>
<td>Sets ·0 = Hmac(key, input), ·1 = Hmac(·0, 0x1), ·i = Hmac(·0, ·i ≠ 1 ∈ i), and returns an n-tuple of 32 byte values, (·1,...,·n). This is the HKDF [15] function.</td>
</tr>
<tr>
<td>Timestamp()</td>
<td>Returns the TAI64N timestamp [7] of the current time, which is 12 bytes of output, the first 8 bytes being a big-endian integer of the number of seconds since 1970 TAI and the last 4 bytes being a big-endian integer of the number of nanoseconds from the beginning of that second.</td>
</tr>
</tbody>
</table>

**5.4.1 Protocol Overview**

In the majority of cases, the handshake will complete in 1-RTT, after which transport data follows:

- **Initiator**—i
- **Responder**—r

**Handshake Initiation**

**Handshake Response**

**Transport Data**

If one peer is under load, then a cookie reply message is added to the handshake, to prevent against denial-of-service attacks:

- **Initiator**—i
- **Responder**—r

**Handshake Initiation**

**Handshake Response**

**Transport Data**

**5.4.2 First Message: Initiator to Responder**

The initiator sends this message, `msg`:

- **type**: 0x1 (1 byte)  
- **reserved**: 0 (3 bytes)  
- **sender**: I (4 bytes)  
- **ephemeral**: (32 bytes)  
- **static**: (32 bytes)  
- **timestamp**: (12 bytes)  
- **mac1**: (16 bytes)  
- **mac2**: (16 bytes)

The timestamp field is explained in section 5.1, and `mac1` and `mac2` are explained further in section 5.4.4. I is generated randomly (fl4) when this message is sent, and is used to tie subsequent replies to the session begun by I.
5.4.1 Protocol Overview

In the majority of cases, the handshake will complete in 1-RTT, after which transport data follows:

- **If one peer is under load, then a cookie reply message is added to the handshake, to prevent against denial-of-service attacks:**

5.4.2 First Message: Initiator to Responder

The initiator sends this message, msg:

- **type**: 0x1 (1 byte)
- **reserved**: 0x0 (3 bytes)
- **sender**: I (4 bytes)
- **ephemeral**: 32 bytes
- **static**: 32 bytes
- **timestamp**: 12 bytes
- **mac1**: 16 bytes
- **mac2**: 16 bytes

The timestamp field is explained in section 5.1, and mac1 and mac2 are explained further in section 5.4.4. I, is generated randomly ($\rho$) when this message is sent, and is used to tie subsequent replies to the session begun by
Message formats

5.4.1 Protocol Overview
In the majority of cases, the handshake will complete in 1-RTT, after which transport data follows:

Initiator —→ Responder ←— Initiator —→ Responder

Handshake Initiation
Handshake Response
Transport Data
Transport Data

If one peer is under load, then a cookie reply message is added to the handshake, to prevent against denial-of-service attacks:

Initiator —→ Responder

Handshake Initiation
Cookie Reply
Handshake Initiation
Handshake Response
Transport Data
Transport Data

5.4.2 First Message: Initiator to Responder
The initiator sends this message, msg:

```
type := 0x1 (1 byte)  reserved := 0³ (3 bytes)
sender := I, (4 bytes)
ephemeral := (32 bytes)
static := (32 bytes)
timestamp := (12 bytes)
mac1 := (16 bytes)
mac2 := (16 bytes)
```

The timestamp field is explained in section 5.1, and mac1 and mac2 are explained further in section 5.4.4. I, is generated randomly ($\rho^I$) when this message is sent, and is used to tie subsequent replies to the session begun by
## Message formats

### 5.4.1 Protocol Overview

In the majority of cases, the handshake will complete in 1-RTT, after which transport data follows:

- **Initiator**
  - Handshake Initiation
  - Handshake Response
  - Transport Data

- **Responder**
  - Handshake Initiation
  - Handshake Response
  - Transport Data

If one peer is under load, then a cookie reply message is added to the handshake, to prevent against denial-of-service attacks:

- **Initiator**
  - Handshake Initiation
  - Cookie Reply
  - Handshake Initiation
  - Handshake Response
  - Transport Data

- **Responder**
  - Handshake Initiation
  - Handshake Response
  - Transport Data

### 5.4.2 First Message: Initiator to Responder

The initiator sends this message, `msg`:

- **type**: 0x1 (1 byte)
- **reserved**: 0 (3 bytes)
- **sender**: I (4 bytes)
- **ephemeral**: 32 bytes
- **static**: 32 bytes
- **timestamp**: 12 bytes
- **mac1**: 16 bytes
- **mac2**: 16 bytes

The timestamp field is explained in section 5.1, and `mac1` and `mac2` are explained further in section 5.4.4. I is generated randomly ($\rho$) when this message is sent, and is used to tie subsequent replies to the session begun by

---

**message flow and setup**

---

**message formatting**
5.4.1 Protocol Overview
In the majority of cases, the handshake will complete in 1-RTT, after which transport data follows:

If one peer is under load, then a cookie reply message is added to the handshake, to prevent against denial-of-service attacks:

5.4.2 First Message: Initiator to Responder
The initiator sends this message, msg:

- **type**: 0x1 (1 byte)
- **reserved**: 0 (3 bytes)
- **sender**: I (4 bytes)
- **ephemeral**: (32 bytes)
- **static**: (32 bytes)
- **timestamp**: (12 bytes)
- **mac1**: (16 bytes)
- **mac2**: (16 bytes)

The timestamp field is explained in section 5.1, and mac1 and mac2 are explained further in section 5.4.4. I is generated randomly ($\mu^I$) when this message is sent, and is used to tie subsequent replies to the session begun by
Our project SPECMON

Protocol Implementation (e.g., WireGuard)

| 1) k := rand() |
| 2) x := receive() |
| 3) y := sender(k, x) |
| 4) send(y) |

Protocol Specification (Tamarin MSRs + Format strings)

Fr(x, In(x) -> Out(senc(k, x))

Event Aggregator

Event Stream

Runtime Monitor

- Accept / Abort
- Debug Information
  - Symbolic traces
  - Abstract protocol states

Function call tracing

Known interface OS, UI

Setup information

Env. interface

Crypto Library

Network I/O

Env.
interface
Our project SPECMON

Protocol Implementation (e.g., WireGuard)

Event Aggregator

Protocol Specification (Tamarin MSRs + Format strings)

Runtime Monitor

Event Stream

1 k := rand()
2 x := receive()
3 y := send(x)
4 send(y)

Accept / Abort

Debug Information
- Symbolic traces
- Abstract protocol states

Function call tracing
Known interface (e.g., OS)
Setup information

Env. interface
Crypto Library
Network I/O

Protocol Implementation (e.g., WireGuard)

Protocol Specification (Tamarin MSRs + Format strings)
## Message formats in SPECMON

<table>
<thead>
<tr>
<th>function symbol</th>
<th>monitor</th>
<th>Tamarin</th>
</tr>
</thead>
<tbody>
<tr>
<td>int, byte, string</td>
<td>typing, can be omitted</td>
<td>ignored</td>
</tr>
<tr>
<td>cat</td>
<td>concatenation</td>
<td>pairs / list</td>
</tr>
<tr>
<td>and, or</td>
<td>bitwise ops</td>
<td>pairs</td>
</tr>
<tr>
<td>reverse</td>
<td>BigEndian &lt;-&gt; LittleEndian</td>
<td>ignored</td>
</tr>
<tr>
<td>add</td>
<td>bitwise op</td>
<td>integer addition</td>
</tr>
</tbody>
</table>
Message formats in SPECMON

<table>
<thead>
<tr>
<th>function symbol</th>
<th>monitor</th>
<th>Tamarin</th>
</tr>
</thead>
<tbody>
<tr>
<td>int, byte, string</td>
<td>typing, can be omitted</td>
<td>ignored</td>
</tr>
<tr>
<td>cat</td>
<td>concatenation</td>
<td>pairs / list</td>
</tr>
<tr>
<td>and, or</td>
<td>bitwise ops</td>
<td>pairs</td>
</tr>
<tr>
<td>reverse</td>
<td>BigEndian &lt;-&gt; LittleEndian</td>
<td>ignored</td>
</tr>
<tr>
<td>add</td>
<td>bitwise op</td>
<td>integer addition</td>
</tr>
</tbody>
</table>

5.4.2 First Message: Initiator to Responder

The initiator sends this message, msg:

- type = 0x1 (1 byte)
- reserved = 0 (3 bytes)
- sender = Ii (4 bytes)
- ephemeral (32 bytes)
- static (32 bytes)
- timestamp (12 bytes)
- mac1 (16 bytes)
- mac2 (16 bytes)

The timestamp field is explained in section 5.1, and mac1 and mac2 are explained further in section 5.4.4. Ii is generated randomly (πi) when this message is sent, and is used to tie subsequent replies to the session begun by.
Message formats in SPECMON

<table>
<thead>
<tr>
<th>function symbol</th>
<th>monitor</th>
<th>Tamarin</th>
</tr>
</thead>
<tbody>
<tr>
<td>int, byte, string</td>
<td>typing, can be omitted</td>
<td>ignored</td>
</tr>
<tr>
<td>cat</td>
<td>concatenation</td>
<td>pairs / list</td>
</tr>
<tr>
<td>and, or</td>
<td>bitwise ops</td>
<td>pairs</td>
</tr>
<tr>
<td>reverse</td>
<td>BigEndian &lt;-&gt; LittleEndian</td>
<td>ignored</td>
</tr>
<tr>
<td>add</td>
<td>bitwise op</td>
<td>integer addition</td>
</tr>
</tbody>
</table>

5.4.2 First Message: Initiator to Responder

The initiator sends this message, msg:

```
<table>
<thead>
<tr>
<th>type</th>
<th>reserved</th>
<th>ephemeral</th>
<th>static</th>
<th>mac1 (16 bytes)</th>
<th>mac2 (16 bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1 (1 byte)</td>
<td>0 (3 bytes)</td>
<td>(32 bytes)</td>
<td>(32 bytes)</td>
<td>(16 bytes)</td>
<td>(16 bytes)</td>
</tr>
<tr>
<td>sender</td>
<td>(4 bytes)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

The timestamp field is explained in section 5.1, and mac1 and mac2 are explained further in section 5.4.4. $I_i$ is generated randomly ($\mathcal{p}^8$) when this message is sent, and is used to tie subsequent replies to the session begun by

```
rule Handshake_Init_Next:
  let
    type = '0x01'
    reserved = '0x000000'
    ephemeral = pekI
    msg = cat(byte(type, '1'), byte(reserved, '3'), byte(sender, '4'), byte(ephemeral, '32'), byte(ats, '48'), byte(ats, '28'))
    handshake = cat(.., msg, byte(h(msg), '16'), ..)
  in
    [ Fr(..), L_PrepareHandshake(..) ] -> [ .., Out(handshake) ]
```
Message formats in SPECMON

<table>
<thead>
<tr>
<th>function symbol</th>
<th>monitor</th>
<th>Tamarin</th>
</tr>
</thead>
<tbody>
<tr>
<td>int, byte, string</td>
<td>typing, can be omitted</td>
<td>ignored</td>
</tr>
<tr>
<td>cat</td>
<td>concatenation</td>
<td>pairs / list</td>
</tr>
<tr>
<td>and, or</td>
<td>bitwise ops</td>
<td>pairs</td>
</tr>
<tr>
<td>reverse</td>
<td>BigEndian &lt;-&gt; LittleEndian</td>
<td>ignored</td>
</tr>
<tr>
<td>add</td>
<td>bitwise op</td>
<td>integer addition</td>
</tr>
</tbody>
</table>

5.4.2 First Message: Initiator to Responder
The initiator sends this message, msg:

```plaintext
<table>
<thead>
<tr>
<th>type</th>
<th>0x1 (1 byte)</th>
</tr>
</thead>
<tbody>
<tr>
<td>reserved</td>
<td>0 (3 bytes)</td>
</tr>
<tr>
<td>sender</td>
<td>4 bytes</td>
</tr>
<tr>
<td>ephemeral</td>
<td>32 bytes</td>
</tr>
<tr>
<td>static</td>
<td>12 bytes</td>
</tr>
<tr>
<td>timestamp</td>
<td>12 bytes</td>
</tr>
<tr>
<td>mac1 (16 bytes)</td>
<td>mac2 (16 bytes)</td>
</tr>
</tbody>
</table>
```

The timestamp field is explained in section 5.1, and mac1 and mac2 are explained further in section 5.4.4. $I_i$ is generated randomly ($\rho^i$) when this message is sent, and is used to tie subsequent replies to the session begun by

```
rule Handshake_Init_Next:
  let
    type = '0x01'
    reserved = '0x000000'
    ephemeral = pekI
    msg = cat(byte(type, '1'), byte(reserved, '3'), byte(sender, '4'), byte(ephemeral, '32'), byte(astat, '48'), byte(ats, '28'))
    ..
    handshake = cat(.., msg, byte(h(msg), '16'), ..)
  in
  [ Fr(..), L_PrepareHandshake(..) ] -> [ .., Out(handshake) ]
```
Message formats in SPECMON

<table>
<thead>
<tr>
<th>function symbol</th>
<th>monitor</th>
<th>Tamarin</th>
</tr>
</thead>
<tbody>
<tr>
<td>int, byte, string</td>
<td>typing, can be omitted</td>
<td>ignored</td>
</tr>
<tr>
<td>cat</td>
<td>concatenation</td>
<td>pairs / list</td>
</tr>
<tr>
<td>and, or</td>
<td>bitwise ops</td>
<td>pairs</td>
</tr>
<tr>
<td>reverse</td>
<td>BigEndian &lt;-&gt; LittleEndian</td>
<td>ignored</td>
</tr>
<tr>
<td>add</td>
<td>bitwise op</td>
<td>integer addition</td>
</tr>
</tbody>
</table>

5.4.2 First Message: Initiator to Responder

The initiator sends this message, msg:

```
<table>
<thead>
<tr>
<th>type (1 byte)</th>
<th>reserved (3 bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>sender (4 bytes)</td>
<td>ephemeral (32 bytes)</td>
</tr>
<tr>
<td>static (32 bytes)</td>
<td>timestamp (12 bytes)</td>
</tr>
<tr>
<td>mac1 (16 bytes)</td>
<td>mac2 (16 bytes)</td>
</tr>
</tbody>
</table>
```

The timestamp field is explained in section 5.1, and mac1 and mac2 are explained further in section 5.4.4. $I_i$ is generated randomly ($\mathcal{U}$) when this message is sent, and is used to tie subsequent replies to the session begun by $I_i$.

```
rule Handshake_Init_Next:
    let
type = '0x01'
reserved = '0x000000'
ephemeral = peki
msg = cat(byte(type, '1'), byte(reserved, '3'), byte(sender, '4'), byte(ephemeral, '32'), byte(ats, '48'), byte(ats, '28'))
    handshakes = cat(..., msg, byte(h(msg), '16'), ..)
in
[ Fr(...), L_PrepareHandshake(..) ] -> [ .., Out(handshakes) ]
```
## Message formats in SPECMON

<table>
<thead>
<tr>
<th>function symbol</th>
<th>monitor</th>
<th>Tamarin</th>
</tr>
</thead>
<tbody>
<tr>
<td>int, byte, string</td>
<td>typing, can be omitted</td>
<td>ignored</td>
</tr>
<tr>
<td>cat</td>
<td>concatenation</td>
<td>pairs / list</td>
</tr>
<tr>
<td>and, or</td>
<td>bitwise ops</td>
<td>pairs</td>
</tr>
<tr>
<td>reverse</td>
<td>BigEndian &lt;-&gt; LittleEndian</td>
<td>ignored</td>
</tr>
<tr>
<td>add</td>
<td>bitwise op</td>
<td>integer addition</td>
</tr>
</tbody>
</table>

### 5.4.2 First Message: Initiator to Responder

The initiator sends this message, `msg`:

```
<table>
<thead>
<tr>
<th>type</th>
<th>reserved</th>
<th>sender</th>
<th>ephemeral</th>
<th>static</th>
<th>timestamp</th>
<th>mac1</th>
<th>mac2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1</td>
<td>0x000000</td>
<td>I_i</td>
<td>32 bytes</td>
<td>4 bytes</td>
<td>12 bytes</td>
<td>16 bytes</td>
<td>16 bytes</td>
</tr>
</tbody>
</table>
```

The timestamp field is explained in section 5.1, and `mac1` and `mac2` are explained further in section 5.4.4. `I_i` is generated randomly ($p^8$) when this message is sent, and is used to tie subsequent replies to the session begun by

```
rule Handshake_Init_Next:
  let
    type = '0x01'
    reserved = '0x000000'
    ephemeral = pekI
    msg = cat(byte(type, '1'), byte(reserved, '3'), byte(sender, '4'), byte(ephemeral, '32'), byte(atsat, '48'), byte(ats, '28'))
    handshak = cat(..., msg, byte(h(msg), '16'), ..)
  in
  ..
  [ Fr(..), L_PrepareHandshake(..) ] -> [ .., Out(handshake) ]
```
**Message formats in SPECMON**

### Message formats

**Hash(input)**
- Blake2s(input, 32), returning 32 bytes of output.
- Mac(key, input)
  - Keyed-Blake2s(key, input, 16), the keyed MAC variant of the BLAKE2s hash function, returning 16 bytes of output.
- Hmac(key, input)
  - Hmac-Blake2s(key, input, 32), the ordinary BLAKE2s hash function used in an HMAC construction, returning 32 bytes of output.

**Kdf**
- $K_{df}^n(key, input) = Hmac(key, input)$,
- $K_{df}^1 = Hmac(0, 0x1)$,
- $K_{df}^i = Hmac(0, K_{df}^{i-1})$, and returns an $n$-tuple of 32 byte values, $(K_{df}^1, ..., K_{df}^n)$. This is the HKDF function.

**Timestamp()**
- Returns the TAI64N timestamp of the current time, which is 12 bytes of output, the first 8 bytes being a big-endian integer of the number of seconds since 1970 TAI and the last 4 bytes being a big-endian integer of the number of nanoseconds from the beginning of that second.

### Construction

The UTF-8 string literal "Noise_IKpsk2_25519_ChaChaPoly_BLAKE2s", 37 bytes of output.

The UTF-8 string literal "WireGuard v1 zx2c4 Jason@zx2c4.com", 34 bytes of output.

**Label-Mac1**
- The UTF-8 string literal "mac1----", 8 bytes of output.

**Label-Cookie**
- The UTF-8 string literal "cookie--", 8 bytes of output.

### 5.4.1 Protocol Overview

In the majority of cases, the handshake will complete in 1-RTT, after which transport data follows:

1. **Initiator** — $i$
2. **Responder** — $r$
3. **Handshake Initiation**
4. **Handshake Response**
5. **Transport Data**
6. **Transport Data**

If one peer is under load, then a cookie reply message is added to the handshake, to prevent against denial-of-service attacks:

1. **Initiator** — $i$
2. **Responder** — $r$
3. **Handshake Initiation**
4. **Cookie Reply**
5. **Handshake Initiation**
6. **Handshake Response**
7. **Transport Data**
8. **Transport Data**

### 5.4.2 First Message: Initiator to Responder

The initiator sends this message, $msg$:

- **type** = 0x1 (1 byte)
- **reserved** = 0 (3 bytes)
- **sender** = $i$ (4 bytes)
- **ephemeral** (32 bytes)
- **static** (32 bytes)
- **timestamp** (12 bytes)
- **mac1** (16 bytes)
- **mac2** (16 bytes)

The timestamp field is explained in section 5.1, and mac1 and mac2 are explained further in section 5.4.4. $i$ is generated randomly (pdf) when this message is sent, and is used to tie subsequent replies to the session begun by $i$. The $i$ symbol can be omitted if the type field is 0x1.

```tamarin
let
  type = '0x01'
  reserved = '0x000000'
  ephemeral = pekI
  msg = cat(byte(type, '1'), byte(reserved, '3'), byte(sender, '4'), byte(ephemeral, '32'), byte(ats, '48'), byte(ats, '28'))
  handshake = cat(msg, byte(h(msg), '16'), ..)
  in
  [ Fr(..), L_PrepareHandshake(…) ] -> [ .., Out(handshake) ]
```

### Rules

**Rule Handshake_Init_Next:**

1. **type** = '0x01'
2. **reserved** = '0x000000'
3. **ephemeral** = pekI
4. **msg** = cat(byte(type, '1'), byte(reserved, '3'), byte(sender, '4'), byte(ephemeral, '32'), byte(ats, '48'), byte(ats, '28'))
5. **handshake** = cat(msg, byte(h(msg), '16'), ..)
6. **in**
7. [ Fr(..), L_PrepareHandshake(…) ] -> [ .., Out(handshake) ]
Potential as an input language
Potential as an input language

- MSR + SPECMON-style formats can express:
Potential as an input language

- MSR + SPECMON-style formats can express:
  - `setup` (rules generating starting states for parties)
Potential as an input language

• MSR + SPECMON-style formats can express:
  - setup (rules generating starting states for parties)
  - role-specific flow (rules like on previous slide)
Potential as an input language

• MSR + SPECMON-style formats can express:
  - **setup** (rules generating starting states for parties)
  - **role-specific flow** (rules like on previous slide)
  - **message formats** (part of role-specific rules)
Potential as an input language

• MSR + SPECMON-style formats can express:
  - **setup** (rules generating starting states for parties)
  - **role-specific flow** (rules like on previous slide)
  - **message formats** (part of role-specific rules)
• caveat: facts can be named arbitrarily:
Potential as an input language

• MSR + SPECMON-style formats can express:
  - **setup** (rules generating starting states for parties)
  - **role-specific flow** (rules like on previous slide)
  - **message formats** (part of role-specific rules)

• caveat: facts can be named arbitrarily:
  - standardise fact naming ( $ROLE_$STATE, e.g., Initiator_Start )
• MSR + SPECMON-style formats can express:
  − setup (rules generating starting states for parties)
  − role-specific flow (rules like on previous slide)
  − message formats (part of role-specific rules)
• caveat: facts can be named arbitrarily:
  − standardise fact naming ( $ROLE$_$STATE$, e.g., Initiator_Start )
  − ProVerif: no explicit state (unless "serious" state machines need to be encoded)
SPECMON: secondary uses
SPECMON: secondary uses

- can validate spec against implementation
SPECMON: secondary uses

- can validate spec against implementation
  - why? specification comes before implementation..
SPECMON: secondary uses

- can validate spec against implementation
  - why? specification comes before implementation..
  - Does it, though?
SPECMON: secondary uses

- can validate spec against implementation
  - why? specification comes before implementation..
  - Does it, though?
  - Specs contain errors (rare) or leave room for interpretation (not so rare). This helps refine the spec.
SPECMON: secondary uses

• can validate spec against implementation
  − why? specification comes before implementation..
  − Does it, though?
  − Specs contain errors (rare) or leave room for interpretation (not so rare). This helps refine the spec.

• verification with Tamarin possible
• can validate spec against implementation
  - why? specification comes before implementation.
  - Does it, though?
    - Specs contain errors (rare) or leave room for interpretation (not so rare). This helps refine the spec.
• verification with Tamarin possible
• what about ProVerif?
SPECMON: secondary uses

• can validate spec against implementation
  - why? specification comes before implementation..
  - Does it, though?
    - Specs contain errors (rare) or leave room for interpretation (not so rare). This helps refine the spec.

• verification with Tamarin possible
• what about ProVerif?
  - (work-in-progress: translation Tamarin into ProVerif)
SPECMON: secondary uses

• can validate spec against implementation
  – why? specification comes before implementation..
  – Does it, though?
    – Specs contain errors (rare) or leave room for interpretation (not so rare). This helps refine the spec.

• verification with Tamarin possible

• what about ProVerif?
  – (work-in-progress: translation Tamarin into ProVerif)
  – SAPIC+ translates to Tamarin and ProVerif (and more)
**SPECMON: secondary uses**

- can validate spec against implementation
  - why? specification comes before implementation..
  - Does it, though?
    - Specs contain errors (rare) or leave room for interpretation (not so rare). This helps refine the spec.
- verification with Tamarin possible
- what about ProVerif?
  - (work-in-progress: translation Tamarin into ProVerif)
  - SAPIC+ translates to Tamarin and ProVerif (and more)
  - message formats can be integrated into ProVerif or SAPIC+
While we did not validate models, the monitor is an accessible way to have resulted in overlooked attacks before (e.g., for the KRACK attack in WPA2 [33], [34]). The monitor is an accessible way to help us understand where the model is imprecise and where ad necessary for monitoring (e.g., adding format strings), but they also for this work. Not only do we and corruption scenarios that were considered, which is irrelevant by the latency from event aggregator until the reception by the low (< 34 ms). As shown in the diagram, this duration is dominated and the time the monitor has processed and accepted the event is the latency between the arrival of an event at the event aggregator update a single con-

<table>
<thead>
<tr>
<th>Component</th>
<th>Model [30]</th>
<th>SPECMON model</th>
<th>RFC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message format</td>
<td>Tuples</td>
<td>Format strings</td>
<td></td>
</tr>
<tr>
<td>Long-term keys</td>
<td>Fresh names</td>
<td>Read from files</td>
<td></td>
</tr>
<tr>
<td>Ephemerals</td>
<td>Precomputed</td>
<td>On demand</td>
<td></td>
</tr>
<tr>
<td>DDoS protect.</td>
<td>Ignored</td>
<td>Precomputed</td>
<td></td>
</tr>
<tr>
<td>Counters</td>
<td>Public names</td>
<td>Natural numbers</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Differences compared to [30]

Figure 4: Evaluation results for

Case Study: Wireguard

Table 1: Differences compared to [30]
Case Study: Wireguard

Table 1: Differences compared to [30]

<table>
<thead>
<tr>
<th>Component</th>
<th>Model [30]</th>
<th>SPECMON model</th>
<th>RFC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message format</td>
<td>Tuples</td>
<td>Format strings</td>
<td></td>
</tr>
<tr>
<td>Long-term keys</td>
<td>Fresh names</td>
<td>Read from files</td>
<td></td>
</tr>
<tr>
<td>Ephemerals</td>
<td>Precomputed</td>
<td>On demand</td>
<td></td>
</tr>
<tr>
<td>DDoS protect.</td>
<td>Ignored</td>
<td>Precomputed</td>
<td></td>
</tr>
<tr>
<td>Counters</td>
<td>Public names</td>
<td>Natural numbers</td>
<td></td>
</tr>
</tbody>
</table>

- message formats are necessary
Case Study: Wireguard

Table 1: Differences compared to [30]

<table>
<thead>
<tr>
<th>Component</th>
<th>Model [30]</th>
<th>SPECMON model</th>
<th>RFC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message format</td>
<td>Tuples</td>
<td>Format strings</td>
<td></td>
</tr>
<tr>
<td>Long-term keys</td>
<td>Fresh names</td>
<td>Read from files</td>
<td>✓</td>
</tr>
<tr>
<td>Ephemerals</td>
<td>Precomputed</td>
<td>On demand</td>
<td></td>
</tr>
<tr>
<td>DDoS protect.</td>
<td>Ignored</td>
<td>Precomputed</td>
<td></td>
</tr>
<tr>
<td>Counters</td>
<td>Public names</td>
<td>Natural numbers</td>
<td></td>
</tr>
</tbody>
</table>
While we did not validate models.

The monitor is an accessible way to have resulted in overlooked attacks before (e.g., for the KRACK attack in WPA2 [33], [34]).

In addition to the online setting, we performed an offline setting using pre-recorded traces. Our monitor is able to process the configuration during the complete run. Moreover, the prior model abstracts the counters used in the handshake protocol and the long-term keys are sent, hence we need to include their computation, too. We did not extend the model beyond their computation and did not model DDoS messages—presumably because there is no lemma that covers the computation of values under concurrency.

<table>
<thead>
<tr>
<th>Component</th>
<th>Model [30]</th>
<th>SPECMON model</th>
<th>RFC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message format</td>
<td>Tuples</td>
<td>Format strings</td>
<td></td>
</tr>
<tr>
<td>Long-term keys</td>
<td>Fresh names</td>
<td>Read from files</td>
<td></td>
</tr>
<tr>
<td>Ephemerals</td>
<td>Precomputed</td>
<td>On demand</td>
<td></td>
</tr>
<tr>
<td>DDoS protect.</td>
<td>Ignored</td>
<td>Precomputed</td>
<td></td>
</tr>
<tr>
<td>Counters</td>
<td>Public names</td>
<td>Natural numbers</td>
<td></td>
</tr>
</tbody>
</table>

- **message formats** are necessary
- **long-term keys** are stored persistently, application starts with loading. Need to express this step.
While we did not validate models, security protocols and corruption scenarios that were considered, which is irrelevant for this work. Not only do we express in SAPiC+, which translates into different specification results.

Almost all event implementers mostly in the security properties and that the changes are minor and that they are invalidated, imprecise models and model because it is on the other hand abstractions may leave room for incorrect verification from their reference Tamarin model [30]. We chose the Tamarin model, which is a high-level formal specification language for reasoning about security protocols. We derive a monitor for the WPA2 attack in WPA2 [33], [34]). The monitor is an accessible way to have resulted in overlooked attacks before (e.g., for the KRACK

And corruption scenarios that were considered, which is irrelevant for this work. Not only do we express in SAPiC+, which translates into different specification results.

The processing time of each event is below 400 ms. In addition to the online setting, we performed an offline analysis of faulty implementations.

Table 1: Differences compared to [30]

<table>
<thead>
<tr>
<th>Component</th>
<th>Model [30]</th>
<th>SPECMON model</th>
<th>RFC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message format</td>
<td>Tuples</td>
<td>Format strings</td>
<td>✔️</td>
</tr>
<tr>
<td>Long-term keys</td>
<td>Fresh names</td>
<td>Read from files</td>
<td>×</td>
</tr>
<tr>
<td>Ephemerals</td>
<td>Precomputed</td>
<td>On demand</td>
<td></td>
</tr>
<tr>
<td>DDoS protect.</td>
<td>Ignored</td>
<td>Precomputed</td>
<td></td>
</tr>
<tr>
<td>Counters</td>
<td>Public names</td>
<td>Natural numbers</td>
<td></td>
</tr>
</tbody>
</table>

• message formats are necessary
• long-term keys are stored persistently, application starts with loading. Need to express this step.
While we did not validate models. The monitor is an accessible way to have resulted in overlooked attacks before (e.g., for the KRACK attack in WPA2 [33], [34]). The monitor is an accessible way to have resulted in overlooked attacks before (e.g., for the KRACK attack in WPA2 [33], [34]). The monitor is an accessible way to have resulted in overlooked attacks before (e.g., for the KRACK attack in WPA2 [33], [34]).

### Case Study: Wireguard

#### Table 1: Differences compared to [30]

<table>
<thead>
<tr>
<th>Component</th>
<th>Model [30]</th>
<th>SPECMON model</th>
<th>RFC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message format</td>
<td>Tuples</td>
<td>Format strings</td>
<td>✓</td>
</tr>
<tr>
<td>Long-term keys</td>
<td>Fresh names</td>
<td>Read from files</td>
<td>✗</td>
</tr>
<tr>
<td>Ephemerals</td>
<td>Precomputed</td>
<td>On demand</td>
<td></td>
</tr>
<tr>
<td>DDoS protect.</td>
<td>Ignored</td>
<td>Precomputed</td>
<td></td>
</tr>
<tr>
<td>Counters</td>
<td>Public names</td>
<td>Natural numbers</td>
<td></td>
</tr>
</tbody>
</table>

- **message formats** are necessary
- **long-term keys** are stored persistently, application starts with loading. Need to express this step.
- **ephemerals** may be precomputed to be shared among different threads when load is high
While we did not validate models. attack in WPA2 [33], [34]). The monitor is an accessible way to help us understand where the model is imprecise and where ad hoc abstractions may leave room for incorrect verification. Necessary for monitoring (e.g., adding format strings), but they also required us to further split the rules to re-iterate inaccuracies in the original model [30]. Nevertheless, these constants are precomputed by the implementation even in the case where no DDoS messages are sent, hence we need to include their computation, too. We did not extend the model beyond their computation and did not model this mechanism. Nevertheless, these constants are precomputed by the latency from event aggregator until the reception by the monitor. The processing time of each event is below 400 ms. As shown in the diagram, this duration is dominated by the latency between the arrival of an event at the event aggregator and the time the monitor has processed and accepted the event. Updating a single counter takes less than 34 ms. As shown in the diagram, this duration is dominated by the latency from event aggregator until the reception by the monitor. The processing time of each event is below 400 ms.

### Table 1: Differences compared to [30]

<table>
<thead>
<tr>
<th>Component</th>
<th>Model [30]</th>
<th>SPECMON model</th>
<th>RFC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message format</td>
<td>Tuples</td>
<td>Format strings</td>
<td>✓</td>
</tr>
<tr>
<td>Long-term keys</td>
<td>Fresh names</td>
<td>Read from files</td>
<td>❌</td>
</tr>
<tr>
<td>Ephemerals</td>
<td>Precomputed</td>
<td>On demand</td>
<td>❌</td>
</tr>
<tr>
<td>DDoS protect.</td>
<td>Ignored</td>
<td>Precomputed</td>
<td></td>
</tr>
<tr>
<td>Counters</td>
<td>Public names</td>
<td>Natural numbers</td>
<td></td>
</tr>
</tbody>
</table>

Table 1 lists differences in components between the prior model and the Speck Model. Model and RFC columns are marked as necessary (✓) or not necessary (❌). Message formats are necessary. Long-term keys are stored persistently, application starts with loading. Need to express this step. Ephemerals may be precomputed to be shared among different threads when load is high.
While we did not validate models.

Security Protocols

hoc abstractions may leave room for incorrect veri-
cation from their reference Tamarin model [30]. We chose the Tamarin

Languages, their models di-

ergence to their reference Tamarin model [30]. We chose the Tamarin

Language model that is

tion of faulty implementations.

We derive a monitor for the

wireguard. The processing time of each event is below 400 ms.

This duration is dominated

by the latency from event aggregator until the reception by the

monitor. The processing time of each event is below 400 ms.

Avg. Latency

Table 1: Differences compared to [30]

<table>
<thead>
<tr>
<th>Component</th>
<th>Model [30]</th>
<th>SPECMON model</th>
<th>RFC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message format</td>
<td>Tuples</td>
<td>Format strings</td>
<td>✔️</td>
</tr>
<tr>
<td>Long-term keys</td>
<td>Fresh names</td>
<td>Read from files</td>
<td>✗</td>
</tr>
<tr>
<td>Ephemerals</td>
<td>Precomputed</td>
<td>On demand</td>
<td>✗</td>
</tr>
<tr>
<td>DDoS protect.</td>
<td>Ignored</td>
<td>Precomputed</td>
<td>✔️</td>
</tr>
<tr>
<td>Counters</td>
<td>Public names</td>
<td>Natural numbers</td>
<td></td>
</tr>
</tbody>
</table>

- **message formats** are necessary
- **long-term keys** are stored persistently, application starts with loading. Need to express this step.
- **ephemerals** may be precomputed to be shared among different threads when load is high
- **DDOS protection**: validation forces model to not omit values

Case Study: Wireguard

Table 1: Differences compared to [30]
Case Study: Wireguard

Table 1: Differences compared to [30]

<table>
<thead>
<tr>
<th>Component</th>
<th>Model [30]</th>
<th>SPECMON model</th>
<th>RFC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message format</td>
<td>Tuples</td>
<td>Format strings</td>
<td>✔</td>
</tr>
<tr>
<td>Long-term keys</td>
<td>Fresh names</td>
<td>Read from files</td>
<td>✗</td>
</tr>
<tr>
<td>Ephemerals</td>
<td>Precomputed</td>
<td>On demand</td>
<td>✗</td>
</tr>
<tr>
<td>DDoS protect.</td>
<td>Ignored</td>
<td>Precomputed</td>
<td>✔</td>
</tr>
<tr>
<td>Counters</td>
<td>Public names</td>
<td>Natural numbers</td>
<td>✔</td>
</tr>
</tbody>
</table>

- **message formats** are necessary
- **long-term keys** are stored persistently, application starts with loading. Need to express this step.
- **ephemerals** may be precomputed to be shared among different threads when load is high
- **DDOS protection**: validation forces model to not omit values
### Case Study: Wireguard

![Wireguard Logo](wireguard.png)

**WireGuard**

**Fast. Modern, Secure.**

**VPN Tunnel**

---

#### Table 1: Differences compared to [30]

<table>
<thead>
<tr>
<th>Component</th>
<th>Model [30]</th>
<th>SPECMON model</th>
<th>RFC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message format</td>
<td>Tuples</td>
<td>Format strings</td>
<td>✓</td>
</tr>
<tr>
<td>Long-term keys</td>
<td>Fresh names</td>
<td>Read from files</td>
<td>X</td>
</tr>
<tr>
<td>Ephemerals</td>
<td>Precomputed</td>
<td>On demand</td>
<td>X</td>
</tr>
<tr>
<td>DDoS protect.</td>
<td>Ignored</td>
<td>Precomputed</td>
<td>✓</td>
</tr>
<tr>
<td>Counters</td>
<td>Public names</td>
<td>Natural numbers</td>
<td>✓</td>
</tr>
</tbody>
</table>

---

- **message formats** are necessary
- **long-term keys** are stored persistently, application starts with loading. Need to express this step.
- **ephemerals** may be precomputed to be shared among different threads when load is high
- **DDOS protection**: validation forces model to not omit values
- **Counters**: simplification steps disallowed (good for assurance, but makes ad-hoc "tweaks" difficult)
Case Study: Wireguard

Table 1: Differences compared to [30]

<table>
<thead>
<tr>
<th>Component</th>
<th>Model [30]</th>
<th>SPECMON model</th>
<th>RFC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message format</td>
<td>Tuples</td>
<td>Format strings</td>
<td>✓</td>
</tr>
<tr>
<td>Long-term keys</td>
<td>Fresh names</td>
<td>Read from files</td>
<td>✗</td>
</tr>
<tr>
<td>Ephemerals</td>
<td>Precomputed</td>
<td>On demand</td>
<td>✗</td>
</tr>
<tr>
<td>DDoS protect.</td>
<td>Ignored</td>
<td>Precomputed</td>
<td>✓✓</td>
</tr>
<tr>
<td>Counters</td>
<td>Public names</td>
<td>Natural numbers</td>
<td>✓✓✓</td>
</tr>
</tbody>
</table>

- **message formats** are necessary
- **long-term keys** are stored persistently, application starts with loading. Need to express this step.
- **ephemerals** may be precomputed to be shared among different threads when load is high
- **DDOS protection**: validation forces model to not omit values
- **Counters**: simplification steps disallowed (good for assurance, but makes ad-hoc "tweaks" difficult)
Conclusion

For protocol specifications, message formats are important, we add them to Tamarin and could add them to ProVerif, as well.

As an extra, this permits monitoring, which is an easy way to test the specification for completeness.
Questions

I. Tamarin-style VS ProVerif-style specification?

I. IMHO: should allow both; verification community can provide translations
Questions

I. Tamarin-style VS ProVerif-style specification?

II. How do we integrate FM into the standardisation process?

I. IMHO: should allow both; verification community can provide translations

II. IMHO:
   - formalise what can be formalised
   - make monitoring cheap, so we can validate spec with implementations
I. IMHO: should allow both; verification community can provide translations

II. IMHO:
   - formalise what can be formalised
   - make monitoring cheap, so we can validate spec with implementations

III. IMHO:
   - linter: are types correct? Is an interaction between honest parties possible?
   - monitor: does the spec allow the implementations behaviour?
   - verification: does the spec ensure authentication? etc.

Questions

I. Tamarin-style VS ProVerif-style specification?

II. How do we integrate FM into the standardisation process?

III. How can we use FM for validation?
I. IMHO: should allow both; verification community can provide translations

II. IMHO:
- formalise what can be formalised
- make monitoring cheap, so we can validate spec with implementations

III. IMHO:
- linter: are types correct? Is an interaction between honest parties possible?
- monitor: does the spec allow the implementations behaviour?
- verification: does the spec ensure authentication? etc.

IV. IMHO: at least
- physical constraints of communication,
- timing requirements
- involved mathematical expressions

Questions

I. Tamarin-style VS ProVerif-style specification?

II. How do we integrate FM into the standardisation process?

III. How can we use FM for validation?

IV. Which aspects (of protocols) am I missing?