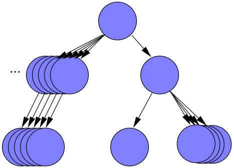
02244 Logic for Security
Security Protocols
Secure Implementation and Typing

Sebastian Mödersheim

February 23, 2026

Lazy Intruder: Summary

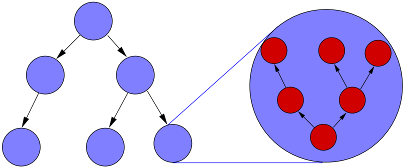
- Without the lazy approach, we would get an infinite search tree because the intruder has often an infinite choice of messages to send.



- We avoid this by using the lazy intruder:

  Layer 1: a symbolic search tree

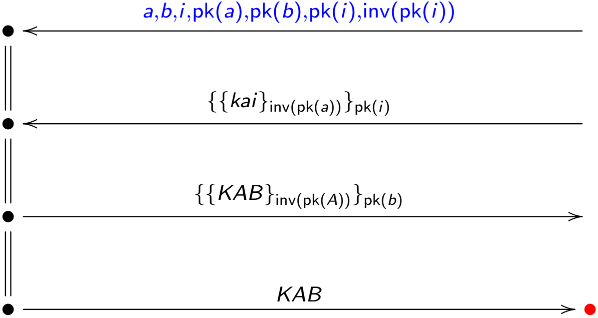
  Layer 2: constraint solving



Lazy Intruder: Summary

- Strand: sequence of incoming and outgoing messages from the point of view of the intruder.
- Can the intruder generate all outgoing messages, given all previous incoming messages?
- Analysis of incoming messages:
  - ★ If encrypted: can the intruder decrypt it?
    - ▶ If so, add the result of the decryption as an incoming message at the earliest point where the decryption key is available.
- Outgoing messages:
  - ★ Axiom: Is there any prior incoming message that unifies?
    - ▶ If so, apply the unifier and remove the outgoing message.
  - ★ Compose: Is the message composed with a public function?
    - ▶ If so replace the message with its subterms.
  - ★ There may be several possibilities and all must be followed!
- Solved if all outgoing messages are variables
  - ★ The intruder can always send something. Be lazy!

Last Week's Challenge

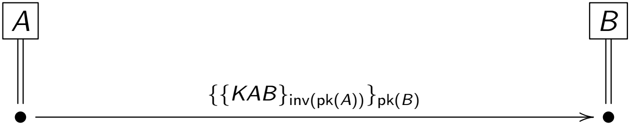


Exercise: show that this constraint has both

- a solution where $A = i$ (i.e., a normal execution)
- a solution where $A = a$ (i.e., an attack)

Last Week's Challenge

Last week's exercise was based on the simple protocol:

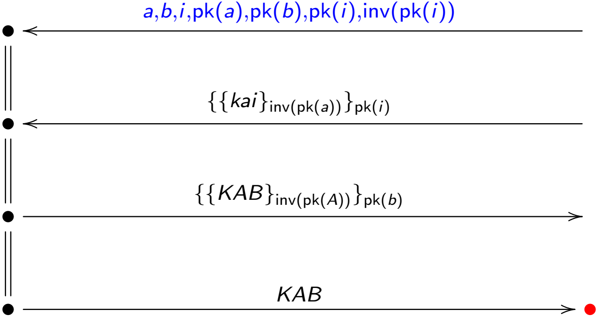


where KAB is a fresh session key that A and B can use thereafter. [1]

[1] Note that if you try this in OFMC you get a special attack ...

- unless $A = B$ is excluded
- because OFMC uses a special algebraic property that $\{\{M\}_{\mathsf{inv}(K)}\}_K = M$

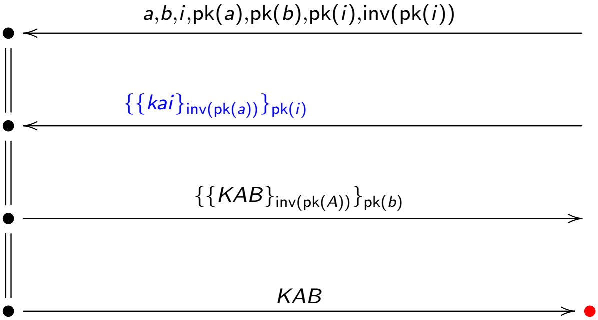
Last Week's Challenge



Exercise: show that this constraint has both

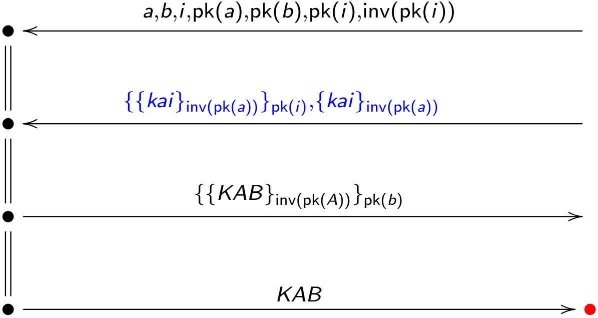
- a solution where $A = i$ (i.e., a normal execution)
- a solution where $A = a$ (i.e., an attack)

Last Week's Challenge



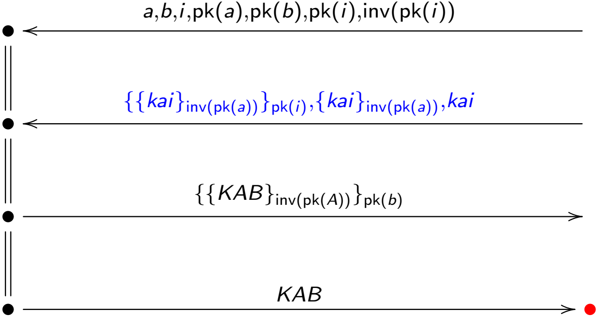
- The incoming message is encrypted with pk(i) and the intruder knows inv(pk(i)).

Last Week's Challenge



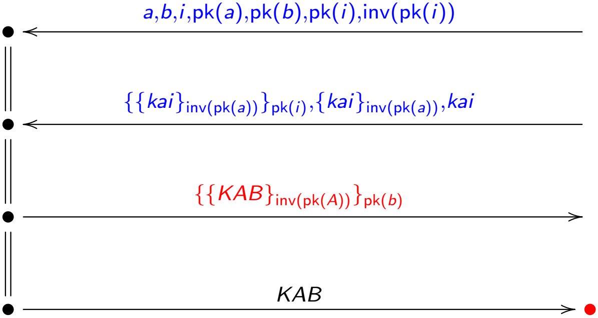
- The incoming message is encrypted with $\mathrm{pk}(i)$ and the intruder knows $\mathrm{inv}(\mathrm{pk}(i))$.
  - ★ Decryption yields $\{kai\}_{\mathrm{inv}(\mathrm{pk}(a))}$ which is a signature

Last Week's Challenge



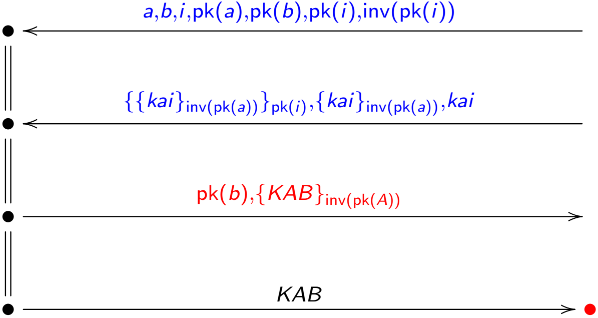
- The incoming message is encrypted with $\mathrm{pk}(i)$ and the intruder knows $\mathrm{inv}(\mathrm{pk}(i))$.
  - ★ Decryption yields $\{kai\}_{\mathrm{inv}(\mathrm{pk}(a))}$ which is a signature
    - ► So by further analysis, the intruder also learns kai

Last Week's Challenge

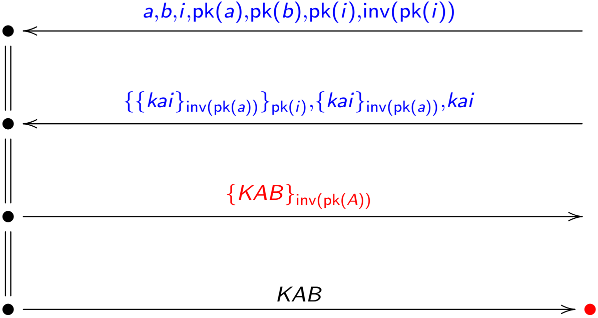


- The outgoing message is encrypted with $\mathsf{pk}(b)$ and the intruder has no such message in his knowledge
  - ★ handle by composition: generate subterms $\mathsf{pk}(b), \{KAB\}_{\mathsf{inv}(\mathsf{pk}(A))}$

Last Week's Challenge

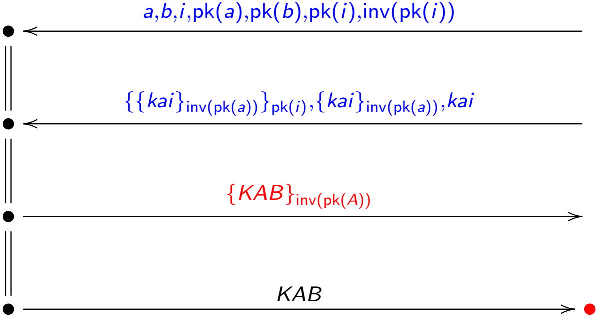


- $\mathrm{pk}(b)$ is easy with axiom.



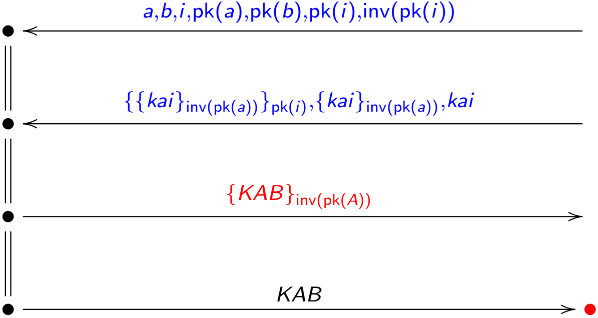
- What about $\{KAB\}_{\mathrm{inv}(\mathrm{pk}(A))}$?

Last Week's Challenge



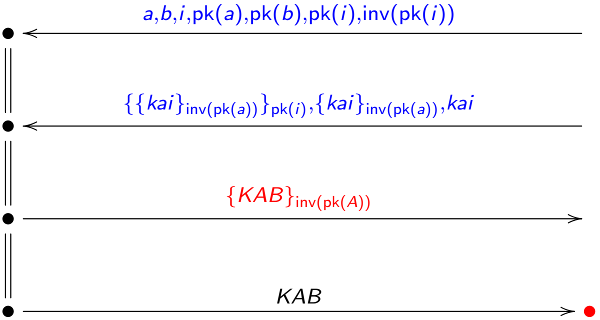
- What about $\{KAB\}_{\mathrm{inv}(\mathrm{pk}(A))}$?
  1. Compose: construct the subterms
  2. Axiom: unify with any term in the knowledge that fits

Last Week's Challenge



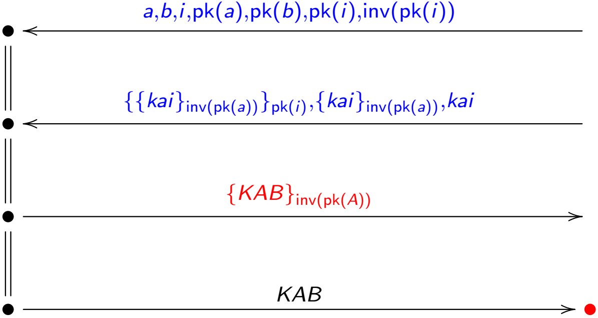
- What about $\{KAB\}_{\mathsf{inv}(\mathsf{pk}(A))}$?
  1. Compose: construct the subterms: $\mathsf{inv}(\mathsf{pk}(A)), KAB$
  2. Axiom: unify with any term in the knowledge that fits

Last Week's Challenge

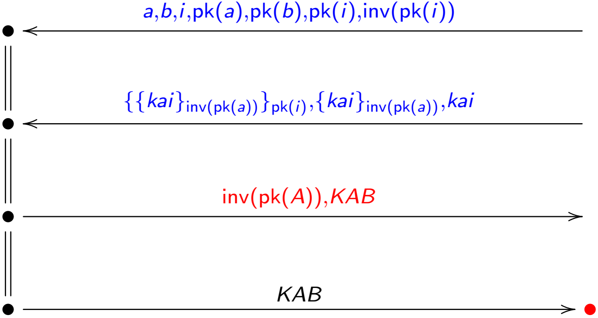


- What about $\{KAB\}_{\mathsf{inv}(\mathsf{pk}(A))}$?
  1. Compose: construct the subterms: $\mathsf{inv}(\mathsf{pk}(A)), KAB$
  2. Axiom: unify with any term in the knowledge that fits
     - only choice: $\{kai\}_{\mathsf{inv}(\mathsf{pk}(a))}$

Last Week's Challenge

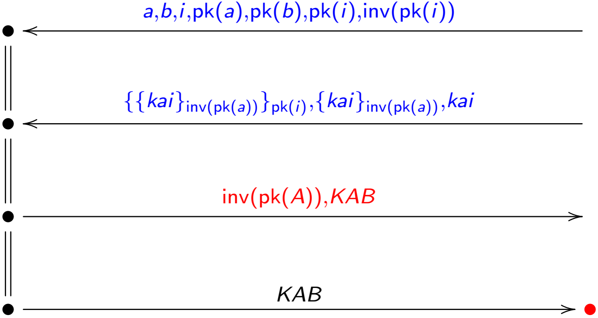


- What about $\{KAB\}_{\mathsf{inv}(\mathsf{pk}(A))}$?
  1. Compose: construct the subterms: $\mathsf{inv}(\mathsf{pk}(A)), KAB$
  2. Axiom: unify with any term in the knowledge that fits
     - only choice: $\{kai\}_{\mathsf{inv}(\mathsf{pk}(a))}$ thus $A = a, KAB = kai$.



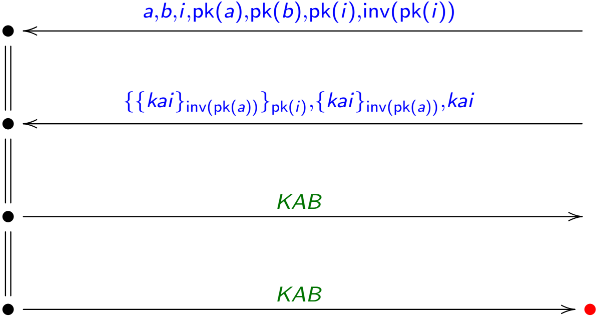
- Using (1) compose for $\{KAB\}_{\mathsf{inv}(\mathsf{pk}(A))}$:
  - i signs some message KAB with some private key $\mathsf{inv}(\mathsf{pk}(A))$

Last Week's Challenge



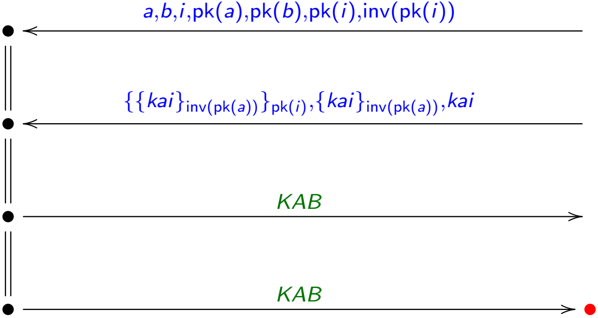
- Using (1) compose for $\{KAB\}_{\mathrm{inv}(\mathrm{pk}(A))}$:
  - i signs some message KAB with some private key $\mathrm{inv}(\mathrm{pk}(A))$
  - i can only has only his own private key: $\mathrm{inv}(\mathrm{pk}(i))$ thus $A = i$.

Last Week's Challenge



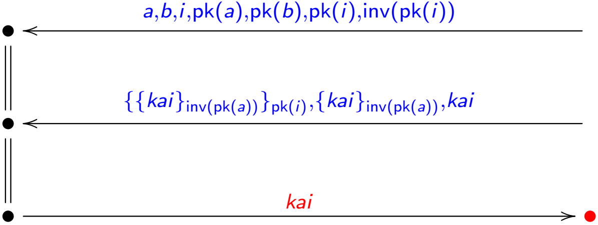
- Using (1) compose for $\{KAB\}_{\mathrm{inv}(\mathrm{pk}(A))}$:
  - ★ i signs some message KAB with some private key $\mathrm{inv}(\mathrm{pk}(A))$
  - ★ i can only has only his own private key: $\mathrm{inv}(\mathrm{pk}(i))$ thus $A = i$.
  - ★ KAB remains lazy: the intruder can use whatever he wants.

Last Week's Challenge



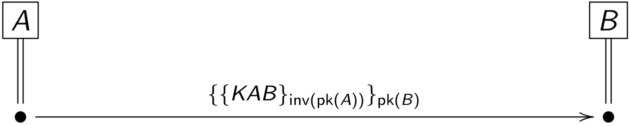
- Using (1) compose for $\{KAB\}_{\text{inv}(\text{pk}(A))}$:
  - ★ i signs some message KAB with some private key inv(pk(A))
  - ★ i can only has only his own private key: inv(pk(i)) thus $A = i$.
  - ★ KAB remains lazy: the intruder can use whatever he wants.
- This actually is the normal protocol execution with the intruder playing role A.

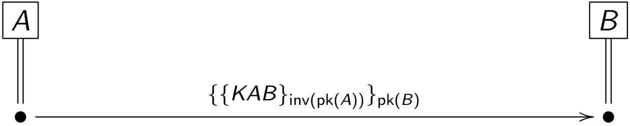
Last Week's Challenge



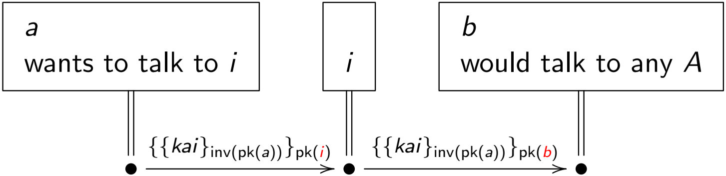
- Using (2) Axiom case ($A = a, KAB = kai$)
  - ★ It remains to generate kai – with Axiom
- This is actually an attack to the protocol.

A Common Mistake...



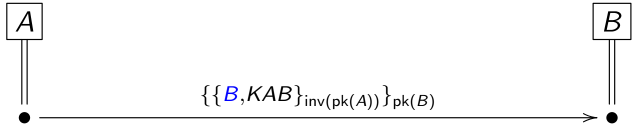


Attack:

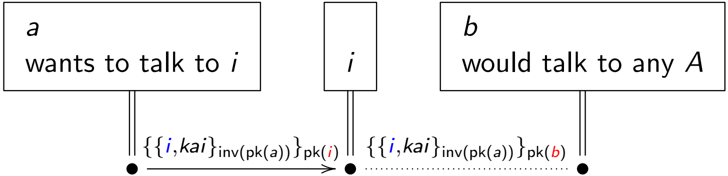


- a thinks: *kai* is a secure session key with *i* ✓
- b thinks: *kai* is a secure session key with *a* ✗
  - ★ the intruder knows *kai* and *a* might have never heard of *b*.

Include the name of the intended recipient in the signature:



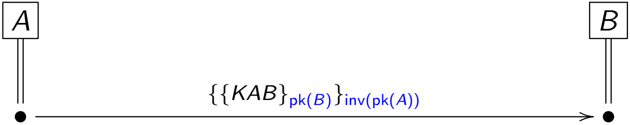
The attack does not work anymore:



- Always be clear what the messages mean!

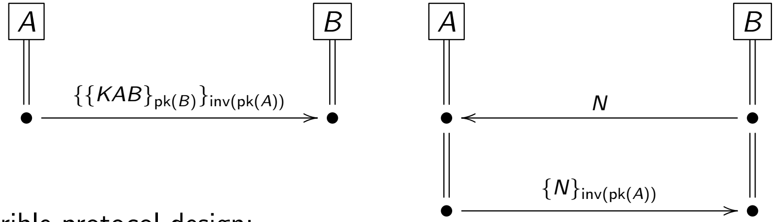
An Alternative Solution

What about first encrypting and then signing?



- Indeed, this also prevents the attack.
- This violates, however, a common recommendation:
  - ★ Do not design protocols where users must sign encrypted data.
  - ★ Why not?

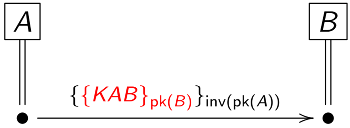
Mind the Environment

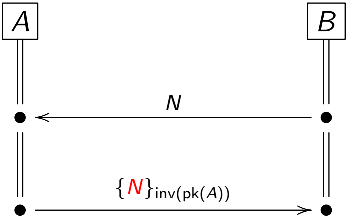


Terrible protocol design:

- while each of these protocols is secure in isolation

Mind the Environment

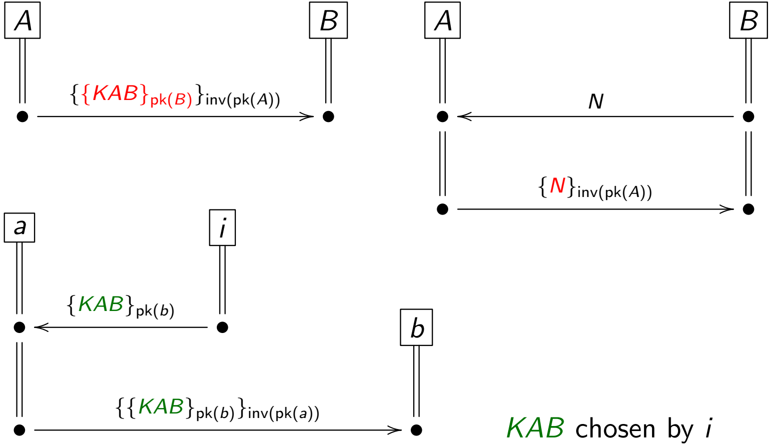




Terrible protocol design:

- while each of these protocols is secure in isolation
- together they break because the signed messages don't say what they mean...

Mind the Environment



KAB chosen by *i*

Attack/Confusion: *a* runs right protocol, *b* runs left protocol.

This is called a type flaw attack.

Classics: Otway-Rees [1987]

see **OFMC** `examples/6.3-Sym-Key-TTP/`

```
A->B: M,A,B,{|NA,M,A,B|}sk(A,s)
B->s: M,A,B,{|NA,M,A,B|}sk(A,s),
            {|NB,M,A,B|}sk(B,s)
s->B: M,A,B,{|NA,KAB|}sk(A,s),
            {|NB,KAB|}sk(B,s)
B->A: M,A,B,{|NA,KAB|}sk(A,s)
```

- Actually secure in default typed mode.
- Use option `--untyped` to get an attack (or remove type declaration for KAB):

```
ATTACK TRACE:
(x401,1) -> i: M(1),x401,x25,{|NA(1),M(1),x401,x25|}_(sk(x401,s))
i -> (x401,1): M(1),x401,x25,{|NA(1),M(1),x401,x25|}_(sk(x401,s))
```

The Problem

$$M, A, B, \{\!|NA, M, A, B|\!\}_{\mathsf{sk}(A,s)}$$
$$\ldots$$
$$M, A, B, \{\!|NA, \ KAB \ |\!\}_{\mathsf{sk}(A,s)}$$

- Given that M, A, B has the same length as KAB
- i can replay the former message in place of the latter

The Problem

$$M, A, B, \{\!|NA, M, A, B|\!\}_{\mathsf{sk}(A,s)}$$
$$\dots$$
$$M, A, B, \{\!|NA, \;KAB\;|\!\}_{\mathsf{sk}(A,s)}$$

- Given that M, A, B has the same length as KAB
- i can replay the former message in place of the latter
  - ★ the recipient will accept M, A, B as the new session key.
  - ★ M, A, B is known to the intruder.

The Problem

$$M, A, B, \{\!|NA, M, A, B|\!\}_{\mathsf{sk}(A,s)}$$
$$\dots$$
$$M, A, B, \{\!|NA, \;\; KAB \;\; |\!\}_{\mathsf{sk}(A,s)}$$

- Given that M, A, B has the same length as KAB
- i can replay the former message in place of the latter
  - ★ the recipient will accept M, A, B as the new session key.
  - ★ M, A, B is known to the intruder.
- Designers must make message formats sufficiently different whenever they mean something different!
- Actually, implementations will not just use string concatenation to structure messages.

Structuring Messages

Real-world Example: TLS 1.3 Handshake

```
enum { client_hello(1),
       server_hello(2),
       new_session_ticket(4),
       ...
} HandshakeType;

struct {
  HandshakeType msg_type;    /* handshake type */
  uint24 length;             /* remaining bytes in message */
  select (Handshake.msg_type) {
    case client_hello:          ClientHello;
    case server_hello:          ServerHello;
    ...
  };
} Handshake;

uint16 ProtocolVersion;
opaque Random[32];
uint8 CipherSuite[2];    /* Cryptographic suite selector */

struct {
  ProtocolVersion legacy_version = 0x0303;    /* TLS v1.2 */
  Random random;
  opaque legacy_session_id<0..32>;
  CipherSuite cipher_suites<2..2^16-2>;
  opaque legacy_compression_methods<1..2^8-1>;
  Extension extensions<8..2^16-1>;
} ClientHello;
```

Structuring Messages

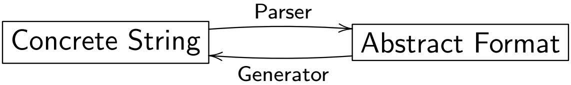
Concrete syntax of message formats, e.g.:

- Record data types (like TLS)
- XML
- JSON
- ...

Abstract syntax of message formats:

- A new function symbol for each message format, e.g. $\mathit{client\_hello}(\mathit{random}, \mathit{cipher\_suites}, \mathit{extensions})$
- The arguments are simply messages (can be random numbers, agent names, encrypted messages,...)
- The function symbol represents abstractly that there is some concrete way to structure the data.

Concrete and Abstract syntax connected by a parser and a generator:



Crypto API

For concrete implementations, we assume a crypto-library:

- `String scrypt(String key, String msg)` implements a symmetric encryption (like AES)
- `String dscrypt(String key, String cipher)` implements the corresponding decryption algorithm will fail if `cipher` is not the result of an encryption with `key`.
- Similar functions for other cryptographic primitives.

We expect that $\mathtt{dscrypt}(\mathtt{k}, \mathtt{scrypt}(\mathtt{k}, \mathtt{m})) = \mathtt{m}$.

Cryptographic soundness results:

- Roughly: by cryptanalysis the intruder cannot achieve anything that he could not achieve by calls of the crypto-API.
- Can be shown under some restrictions and hardness assumptions e.g. [Abadi & Rogaway] [Backes et al.]

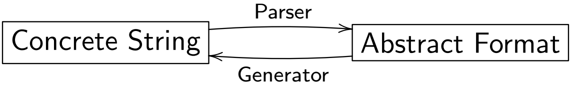
Non-Crypto API

Similarly, we assume a non-crypto-library:
For every format $f(t_1, \ldots, t_n)$:

- A corresponding data-type `F` that has fields for $t_1, \ldots, t_n$
- `F parseF(String s)` that tries to parse the given string for this format and return a corresponding data structure.
- `String generate(F form)` that generates a string from the given data structure.

We expect that

- `parseF(generate(form))` $=$ `form`
  for every object `form` of datatype `F`
- `generate(parseF(s))` $=$ `s`
  for every string `s` where `parseF(s)` does not fail.



Soundness

$$\begin{tikzcd} \text{Concrete String} \arrow[r, bend left, "\text{Parser}"] & \text{Abstract Format} \arrow[l, bend left, "\text{Generator}"] \end{tikzcd}$$

Desirable properties:

- Unambiguous: For a concrete string there is at most one way to parse it for a format.
- Disjointness: No string can be parsed for more than one format.

A soundness result for the non-crypto API [M.&Katsoris]:

- Similar to the result for soundness of the crypto API
- Roughly: by string manipulation the intruder cannot achieve anything that he could not achieve by calls of the non-crypto-API.
- Requires that formats are unambiguous and pairwise disjoint, and soundness of crypto.

Formats for Otway Rees

Consider again the Otway-Rees protocol – without the cleartext messages for simplicity:

```
A->B: {|NA,M,A,B|}sk(A,s)
B->s: {|NA,M,A,B|}sk(A,s),
      {|NB,M,A,B|}sk(B,s)
s->B: {|NA,KAB|}sk(A,s),
      {|NB,KAB|}sk(B,s)
B->A: {|NA,KAB|}sk(A,s)
```

Uses two different message formats:

- `NA, M, A, B` of type `number,number,agent,agent`.
- `NA, KAB` of type `nonce,symkey`.

We could define two data-formats for this:

- `f1(N, M, A, B)` with four arguments
- `f2(N, K)` with two arguments

Formats for Otway Rees

```
A->B: {|f1(NA,M,A,B)|}sk(A,s)
B->s: {|f1(NA,M,A,B)|}sk(A,s),
      {|f1(NB,M,A,B)|}sk(B,s)
s->B: {|f2(NA,KAB)|}sk(A,s),
      {|f2(NB,KAB)|}sk(B,s)
B->A: {|f2(NA,KAB)|}sk(A,s)
```

Notes:

- The intruder can construct and deconstruct `f1` and `f2` like concatenations.

Formats in OFMC

```
Types: Format f1,f2;
       ...
Knowledge: A: A,B,sk(A,s);
           B: B,A,sk(B,s);
           s: A,B,sk(A,s),sk(B,s)
Actions:
A->B: M,A,B,{|f1(NA,M,A,B)|}sk(A,s)
B->s: M,A,B,{|f1(NA,M,A,B)|}sk(A,s),{|f1(NB,M,A,B)|}sk(B,s)
s->B: M,{|f2(NA,KAB)|}sk(A,s),{|f2(NB,KAB)|}sk(B,s)
B->A: M,{|f2(NA,KAB)|}sk(A,s)
Goals:...
```

- Formats are automatically in the knowledge of every agent, thus also the intruder can apply them.
- Formats are transparent: if the intruder knows `f1(NB,M,A,B)` then also `NB,M,A,B`.

Formats for Otway Rees

```
A->B: {|f1(NA,M,A,B)|}sk(A,s)
B->s: {|f1(NA,M,A,B)|}sk(A,s),
      {|f1(NB,M,A,B)|}sk(B,s)
s->B: {|f2(NA,KAB)|}sk(A,s),
      {|f2(NB,KAB)|}sk(B,s)
B->A: {|f2(NA,KAB)|}sk(A,s)
```

Does using formats prevent all type-flaw attacks on Otway-Rees?

Formats for Otway Rees

```
A->B:  {|f1(NA,M,A,B)|}sk(A,s)
B->s:  {|f1(NA,M,A,B)|}sk(A,s),
       {|f1(NB,M,A,B)|}sk(B,s)
s->B:  {|f2(NA,KAB)|}sk(A,s),
       {|f2(NB,KAB)|}sk(B,s)
B->A:  {|f2(NA,KAB)|}sk(A,s)
```

Does using formats prevent all type-flaw attacks on Otway-Rees?

- The intruder can still construct and send messages like
  `{| f1(a,b,i,b) |}sk(i,s)`
- This message is called ill-typed because it contains agents where numbers are expected
- It would actually still be accepted by the server.

Formats for Otway Rees

```
A->B: {|f1(NA,M,A,B)|}sk(A,s)
B->s: {|f1(NA,M,A,B)|}sk(A,s),
      {|f1(NB,M,A,B)|}sk(B,s)
s->B: {|f2(NA,KAB)|}sk(A,s),
      {|f2(NB,KAB)|}sk(B,s)
B->A: {|f2(NA,KAB)|}sk(A,s)
```

Does using formats prevent all type-flaw attacks on Otway-Rees?

- The intruder can still construct and send messages like
  `{| f1(a,b,i,b) |}sk(i,s)`
- This message is called ill-typed because it contains agents where numbers are expected
- It would actually still be accepted by the server.
- Idea: the intruder cannot really exploit this, because nobody would accidentally read this as an `f2` message for instance.

Message Patterns

- We now give a result for protocols that are resistant to type flaws – a notion we need to define.
- For that, we first define what sub-message patterns are.

Definition (Sub-Message-Patterns)

The sub-message patterns $SMP(P)$ of a protocol P are the least set

- that contains all the protocol messages
- for every message $f(t_1, \ldots, t_n) \in SMP(P)$ also the sub-messages $t_1, \ldots, t_n$ are in $SMP(P)$.
- for every message of the form $\{m\}_k \in SMP(P)$ also $\mathrm{inv}(k) \in SMP(P)$.

For simplicity, every pair m_1, m_2 can directly be considered as two messages m_1 and m_2.
Finally, rename all variables such that every two distinct messages $s, t \in SMP(P)$ have no variables in common.

Example: Otway-Rees

Messages of the protocol:

```
M,A,B,{|f1(NA,M,A,B)|}sk(A,s),
M,A,B,{|f1(NA,M,A,B)|}sk(A,s),
{|f1(NB,M,A,B)|}sk(B,s),
M,{|f2(NA,KAB)|}sk(A,s),{|f2(NB,KAB)|}sk(B,s),
M,{|f2(NA,KAB)|}sk(A,s)
```

Subterms:

```
f1(NA,M,A,B)
f2(NA,KAB)
sk(A,s),NA,M,A,B
```

Example: Otway-Rees

Renaming (and removing duplicates)
$SMP(Otway - Rees) = \{$

```
M_1,A_1,B_1,
{|f1(NA_2,M_2,A_2,B_2)|}sk(A_2,s),
{|f1(NB_3,M_3,A_3,B_3)|}sk(B_3,s),
{|f2(NA_4,KAB_4)|}sk(A_4,s),
{|f2(NB_5,KAB_5)|}sk(B_5,s),
f1(NA_6,M_6,A_6,B_6)
f2(NA_7,KAB_7)
sk(A_8,s),
NA_9
```

$\}$.

Type-Flaw Resistance

Type-Flaw Resistance

A protocol is called type-flaw resistant if the following holds:

- Take any two elements s and t of the message patterns that are not variables
- If s and t can be unified then s and t have the same type.

Example: Otway-Rees

We have to check only messages that are not variables themselves:

```
1. {|f1(NA_2,M_2,A_2,B_2)|}sk(A_2,s),
2. {|f1(NB_3,M_3,A_3,B_3)|}sk(B_3,s),
3. {|f2(NA_4,KAB_4)|}sk(A_4,s),
4. {|f2(NB_5,KAB_5)|}sk(B_5,s),
5. f1(NA_6,M_6,A_6,B_6)
6. f2(NA_7,KAB_7)
7. sk(A_8,s),
```

Example: Otway-Rees

We have to check only messages that are not variables themselves:

```
1. {|f1(NA_2,M_2,A_2,B_2)|}sk(A_2,s),
2. {|f1(NB_3,M_3,A_3,B_3)|}sk(B_3,s),
3. {|f2(NA_4,KAB_4)|}sk(A_4,s),
4. {|f2(NB_5,KAB_5)|}sk(B_5,s),
5. f1(NA_6,M_6,A_6,B_6)
6. f2(NA_7,KAB_7)
7. sk(A_8,s),
```

The only pairs of unifiable messages are:

- 1. and 2. – are of the same type
- 3. and 4. – are of the same type

These are all type-correct. Thus type-flaw resistant!

Counter-Example: Original Otway-Rees

The original protocol without formats:

```
1. {|NA_2,M_2,A_2,B_2|}sk(A_2,s),
2. {|NB_3,M_3,A_3,B_3)|}sk(B_3,s),
3. {|NA_4,KAB_4|}sk(A_4,s),
4. {|NB_5,KAB_5|}sk(B_5,s),
5. sk(A_7,s),
```

For instance, 1. and 3. have a unifier

- `NA_2=NA_4, (M_2,A_2,B_2)=KAB_4, A_2=B_3`
- This violates our notion of type-flaw resistance, so the following theorem about type-flaw resistant protocols does not apply.
- Note that the entire concatenation `M_2,A_2,B_2` is here a single message that can be unified with `KAB`.
  - ★ This may or may not work in a real implementation...

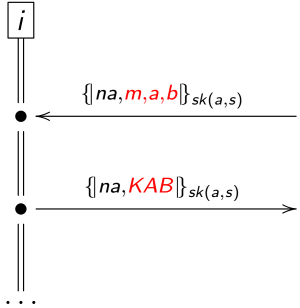
A Typing Result

Theorem

Given an attack against a type-flaw resistant protocol. Then there is a well-typed attack against the protocol, i.e., where the intruder sends no ill-typed messages. [Hess & M.], extending [Arapinis & Duflot]

- As a consequence, it is sound to restrict the intruder model to well-typed messages for type-flaw resistant protocols.
- This often removes a lot of "garbage" from the analysis.
- This comes at a low price: clear messages are good engineering practice anyway!

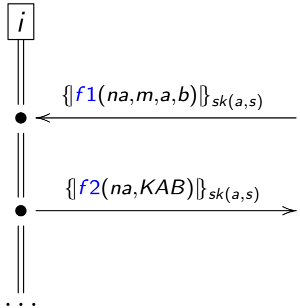
When the lazy intruder analyzes a protocol that is not type flaw-resistant, the following can happen:



and the intruder solves this by an Axiom, leading to the ill-typed unifier $KAB = (m, a, b)$.

Proof Idea

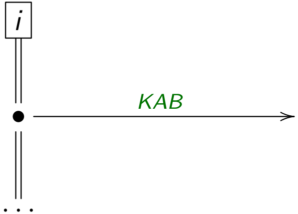
When the lazy intruder analyzes a protocol that is type flaw-resistant



the unification is not possible when the terms in question have different type – and are not variables.

Proof Idea

If a term to generate is a variable, e.g.:



we are lazy and there is always something well-typed i can use.

Thus, on type-flaw resistant protocols

- the lazy intruder never performs an ill-typed substitution
- for all remaining variables there is well-typed choice.
- and thus, if there is an attack, then there is a well-typed one.

Summary

For secure implementations:

- Use a well-established crypto library
  - ★ Do not cook up your own stuff (unless you are a cryptographer)
  - ★ Make sure you understand the requirements and guarantees of the library and its functions, and what they achieve
- Mind the things that are not crypto:
  - ★ Define precise formats
  - ★ Check they are unambiguous and pairwise disjoint
  - ★ Write parsers and generators that do not suffer from buffer overflows and the like
- Ensure that all subterms of different types are distinguishable
  - ★ Use the formats for that
  - ★ Do use crypto on raw data like $\{N\}_{\mathsf{inv}(k)}$.
- Verify your abstract design with a tool like OFMC to find logical flaws – in the typed model.

Challenge/Free Exercise

Consider the following protocol:

```
A->B: A,NA
B->s: A,B,NA,NB,{|A,NA,NB|}sk(B,s)
s->A: {|B,KAB,NA,NB|}sk(A,s),{|A,KAB|}sk(B,s)
A->B: {|A,KAB|}sk(B,s),{|NB|}KAB
```

- Can you find a type-flaw attack against this protocol?
  Hint: ignore A and s and consider just one honest b in role B.
- Can you suggest formats for this protocol so that it becomes type-flaw resistant?

Relevant Research Papers

- Martín Abadi and Phillip Rogaway. *Reconciling Two Views of Cryptography*. J. Cryptology 20(3), 2007.
- Myrto Arapinis and Marie Duflot. *Bounding Messages for Free in Security Protocols.* FSTTCS 2007.
- Michael Backes, Markus Dürmuth and Ralf Küsters. *On Simulatability Soundness and Mapping Soundness of Symbolic Cryptography*. FSTTCS 2007.
- Véronique Cortier, Bogdan Warinschi. *A composable computational soundness notion.* CCS 2011.
- Andreas Hess and Sebastian Mödersheim. *Formalizing and Proving a Typing Result for Security Protocols in Isabelle/HOL.* CSF 2017.
- Sebastian Mödersheim and Georgios Katsoris. *A Sound Abstraction of the Parsing Problem.* CSF 2014.
- Dave Otway and Owen Rees. *Efficient and timely mutual authentication.* ACM SIGOPS Op. Sys. Rev. 21 (1), 1987.