Riccardo Bresciani

The ZRTP Protocol
Security Considerations

Research Report LSV-07-20

May 2007

Laboratoire Spécification et Vérification

Ecole Normale Supérieure de Cachan
61, avenue du Président Wilson
94235 Cachan Cedex France

The ZRTP Protocol
Security Considerations

Riccardo Bresciani[†]

May 22, 2007

Abstract ZRTP is draft of key agreement protocol by Phil Zimmermann, which relies on a Diffie-Hellman exchange to generate SRTP session parameters, providing confidentiality and protecting against *Man-in-the-Middle* attacks even without a public key infrastructure or endpoint certificates. This is an analysis of the protocol performed with AVISPA and ProVerif, which tests security properties of ZRTP; in order to perform the analysis, the protocol has been modeled in HLPSL (for AVISPA) and in the applied π-calculus (for Proverif). An improvement to gather some extra resistance against *Man-in-the-Middle* attacks is also proposed.

Contents

1 Overview of the ZRTP Protocol

ZRTP is a protocol[1] for media path Diffie-Hellman exchange to agree on a session key and parameters for establishing SRTP sessions[2]. [1, 25].

This protocol does not rely on a public key infrastructure or on certification authorities, in fact ephimeral Diffie-Hellman keys are generated on each session establishment: this allows to bypass the complexity of creating and mantaining a complex third trusted party.
These keys will contribute to the generation of the session key and parameters for SRTP sessions, along with previously shared secrets: this gives protection against *Man-in-the-Middle attacks*, assuming the attacker was not present in the first session between the two endpoints.

[†]LSV, ENS Cachan, France & Scuola Superiore Sant'Anna, Italy — bresciani@sssup.it. This work has been done during my stage at LSV, under the supervision of Steve Kremer (LSV, ENS Cachan & CNRS & INRIA, France — kremer@lsv.ens-cachan.fr).

[1] Actually, at the time of this writing, ZRTP is still a *draft of* protocol.

[2] SRTP has not been projected to perform this task, so it has to rely on external key management protocols. [5, 28]

To ensure that the attacker is indeed not present in the first session (when no shared secrets exist), the Short Authentication String method is used: the two endpoint compare a value by reading it aloud. In case the two values match, then no *Man-in-the-Middle attack* has been performed.

1.1 Call flow

In the call flow of the protocol, two roles can be identified: *initiator* and *responder*[3].

The parties exchange ZRTP messages, which are enclosed in a frame (see figure 2) that makes ZRTP packets clearly distinguishable from RTP ones, thus maintaining backward compatibility with clients which do not support ZRTP.

The CRC included in the frame is more reliable than the one built-in in UDP, thus reducing the chances of mistaking a transmission error for an attack.

Each message starts with a preamble containing information about the message length (including preamble length, 1 word).

1.2 RTP Session Establishment

The key agreement is performed via an RTP session, which is established through a signaling protocol such as SIP [6, 29].

The signaling protocol passes the signaling secret (`sigs`), which is the hash of the concatenation of Call-ID (`call-id`), *to* tag (`to-tag`) and *from* tag (`from-tag`):

$$\texttt{sigs} = \mathcal{H}(\texttt{call-id}|\texttt{to-tag}|\texttt{from-tag})$$

It is a secret because `call-id` is random-generated, but only if the SIP session is secure (usually data is transmitted through a TLS channel).

The signaling also provides the SRTP secret (`srtps`), which is the hash of SRTP master key (`SRTP-mkey` and salt (`SRTP-msalt`):

$$\texttt{srtps} = \mathcal{H}(\texttt{SRTP-mkey}|\texttt{SRTP-msalt})$$

1.3 Key Agreement — Diffie-Hellman Mode

The key agreement algorithm can be divided into 4 steps:

1. discovery;
2. hash committment;
3. Diffie-Hellman exchange;
4. switch to SRTP and confirmation.

These steps, shown in figure 1, are discussed in the following subsections.

Discovery

During the discovery phase *Alice* and *Bob* exchange their ZRTP identifiers[4] (`ZID`). Besides, they gather information about each other's capabilities, in terms of supported ZRTP versions (`zrtpv`), hash functions[5] (`hash`), ciphers[6] (`cipher`), authorization tag lengths[7] (`at`), key agreement types[8] (`keya`), and SAS algorithms[9] (`sas`).

[3] In this writing the role of the initiator will always be played by *Bob*, the one of the responder by *Alice* and *Marvin* will be an active attacker.

[4] Each ZRTP instance has a unique 96 bit random identifier, generated once at installation time.

[5] At the time of this writing the only supported hash function is SHA-256 (`S256`) [3, 30]

[6] The cipher to be used is AES, with 128 bit (`AES1`) or 256 bit (`AES2`) keys [9, 32, 5].

[7] The authentication tag relies on HMAC-SHA1 [4, 31] and can be 32 bit (`HS32`) or 80 bit (`HS80`) long [5].

[8] This chooses the Diffie-Hellman mode, which can use 3072 bit (`DH3k`) or 4096 bit (`DH4k`) [7, 8], or the Preshared mode (`Prsh`) which uses only the shared secrets, see section 1.4.

[9] Possible SAS schemes use base 32 (`B32`) or base 256 (`B256`) encoding.

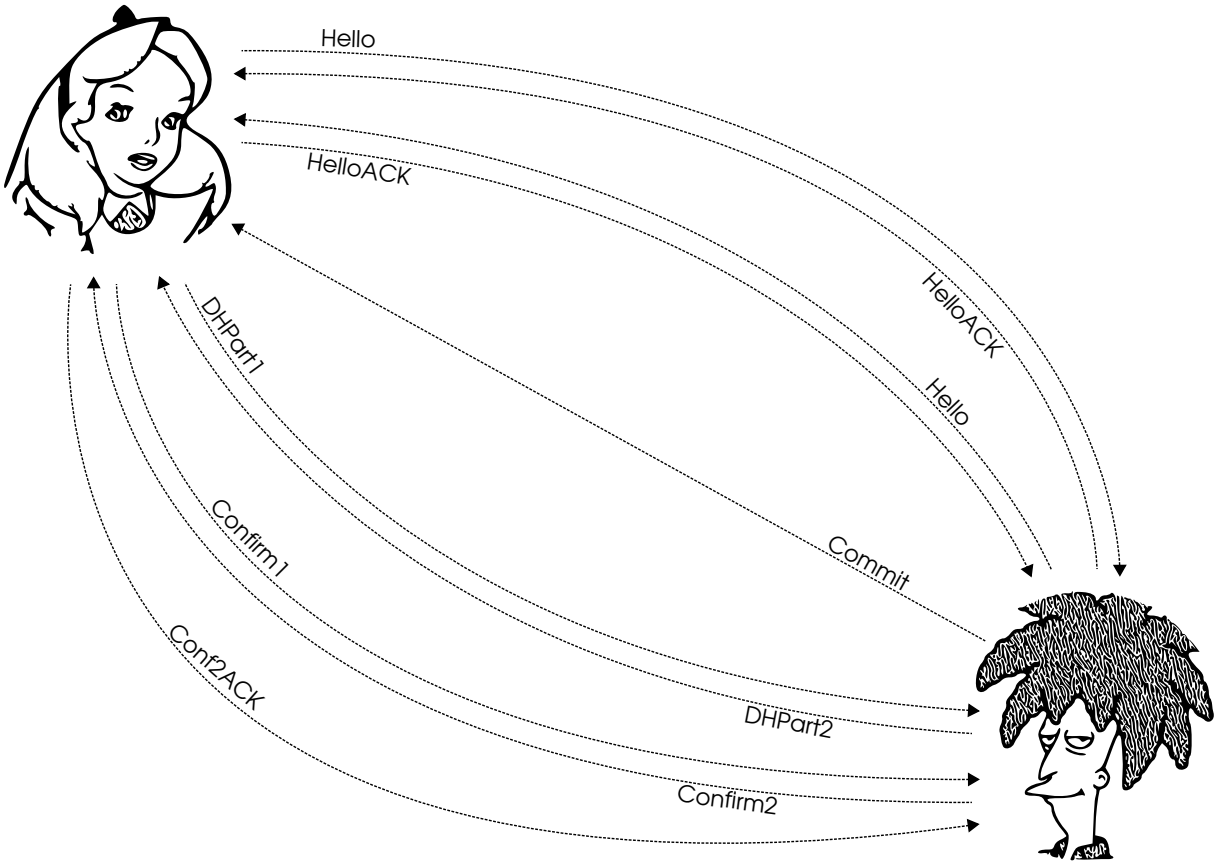

Figure 1: Key agreement algorithm

The messages exchanged during this phase are called `Hello` messages (variable length), to which each party replies with a `HelloACK` message[10] (3 words, see figure 2).

The `ZID` is used to retrieve information on retained shared secrets (`rs1`, `rs2`). Additional other secrets may exist and they are referred to as `os`.
If a secret of a given type does not exist a random value is generated, so an eavesdropper can't get to know the number of secrets shared between the two parties: the random value will be discarded when the `DHPart1` and `DHPart2` messages are exchanged[11].

Hash Commitment

After the discovery phase, *Bob* performs the hash commitment, by choosing — among the ones supported by both endpoints — which hash function, cipher, authorization tag length, key agreement type and SAS algorithm should be used.
His choice is sent via the `Commit` message (19 words, see figure 2).

Bob generates a fresh Diffie-Hellman key pair (secret value `svB` and public value `pvB`, `svB` being twice as long as the AES key length):

$$\texttt{pvB} = g^{\texttt{svB}} \bmod p$$

where g and p are determined by the key agreement type value.

[10] *Bob* could skip sending his `HelloACK` message and reply with a `Commit` message.

[11] An user should be careful when accepting a session where there is no retained shared secret match (for example the first session): *Marvin* could successfully perform an attack by causing mismatches in all the secrets, thus being able to establish an SRTP session with both parties. In this case the solution is the SAS: it is a short string generated from the previously exchanged messages — therefore it is not secret — that has to be compared from the two endpoints, usually by having each user to read aloud his string. It is necessary when all secrets mismatch to ensure that no *Man-in-the-Middle attack* has been performed.

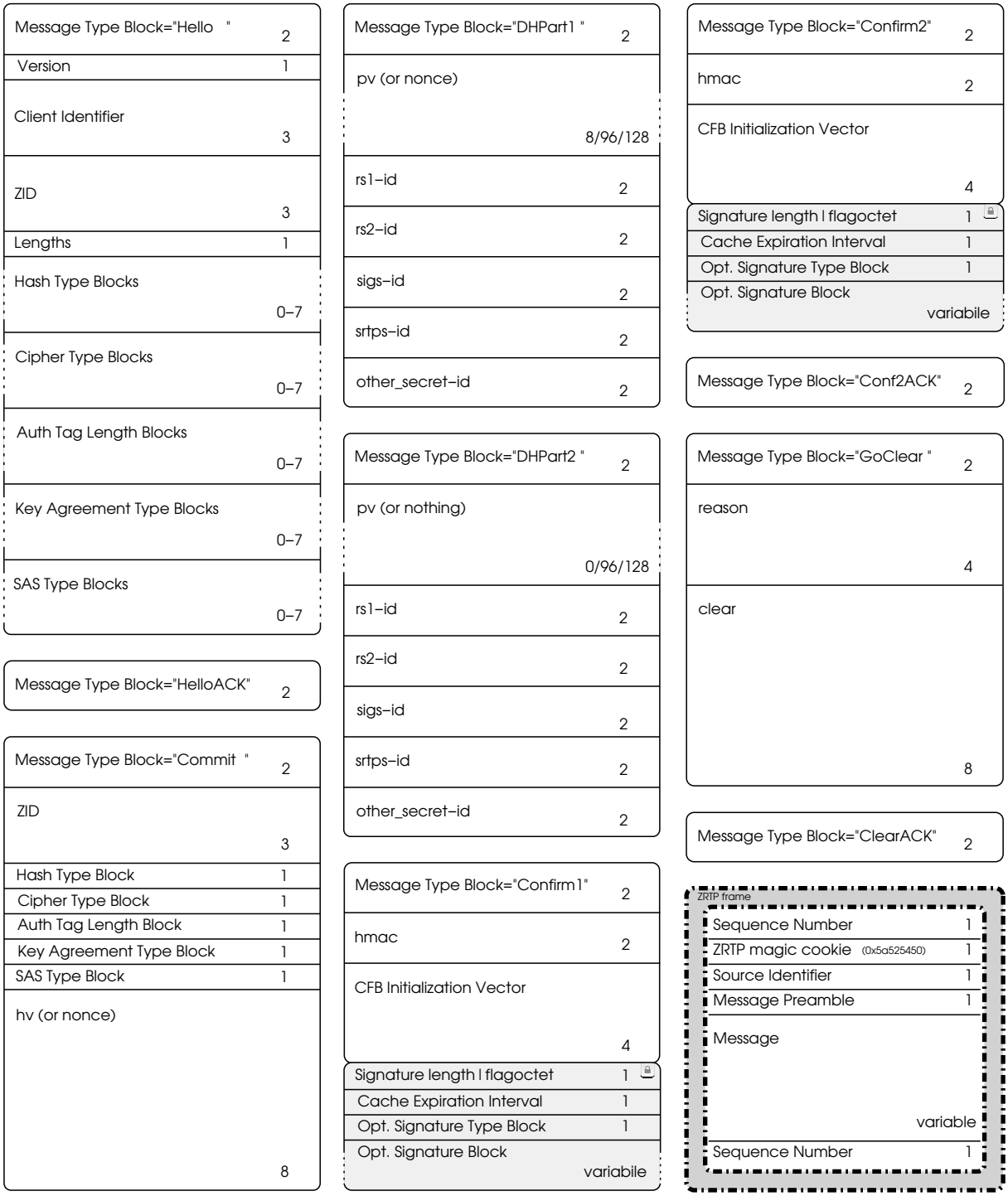

Figure 2: ZRTP messages.

Bob calculates `hvB` as the hash of `pvB` concatenated with the information of *Alice*'s `Hello` message[12]:

$$\texttt{hvB} = \mathcal{H}(\texttt{pvB}|\textit{Alice}\text{'s }\texttt{Hello})$$

Because of the simmetry of the discovery phase, it can happen that *Alice* tries to act as the initiator, therefore sending her own `Commit` message upon receipt of *Bob*'s `HelloACK` message: *Alice*'s message is discarded if $\texttt{hvA} < \texttt{hvB}$, otherwise the protocol lets her be the initiator.

Diffie-Hellman Exchange

By means of the Diffie-Hellman exchange, *Alice* and *Bob* aim to generate a new shared secret `s0`, from which later will derive the new retained shared secret `rs0`.

When *Alice* gets the `Commit` message, she generates her own fresh Diffie-Hellman key pair (`svA` and `pvA`).

Alice gathers the retained secrets shared with *Bob* (`rs1`, `rs2`): she will be able to generate a new secret `s0` by hashing the concatenation of the Diffie-Hellman shared secret (`dhss`, which she will be able to compute upon receipt of the `DHPart2` message from *Bob*), the shared secrets (the retained ones, `sigs`, `srtps` and `os`) and the message hash (`mh`, hash of the concatenation of her `Hello` message, `Commit`, `DHPart1` and `DHPart2`):

$$\texttt{mh} = \mathcal{H}(\textit{Alice}\text{'s }\texttt{Hello}|\texttt{Commit}|\texttt{DHPart1}|\texttt{DHPart2})$$
$$\texttt{s0} = \mathcal{H}(\texttt{dhss}|\texttt{s1}|\texttt{s2}|\texttt{s3}|\texttt{s4}|\texttt{s5}|\texttt{mh})$$

Alice and *Bob* could have their retained shared secrets sorted in a different order (or even have different secrets or none at all) and that would compromise the calculation of `s0`: to avoid this *Alice* sorts hers following *Bob*'s order, assigning to `sX` the correct secret (in case of mismatch `sX` will be assigned a null value).
To achieve this goal *Alice* calculates a sequence of HMACs truncated to 64 bit ($\mathcal{H}_M(k, s)$, where k is the key and s is the string on which the key is applied) using the retained shared secrets as keys:

$$\begin{aligned}
\texttt{rs1-idA} &= \mathcal{H}_M(\texttt{rs1}, \texttt{"Responder"}) \\
\texttt{rs2-idA} &= \mathcal{H}_M(\texttt{rs2}, \texttt{"Responder"}) \\
\texttt{sigs-idA} &= \mathcal{H}_M(\texttt{sigs}, \texttt{"Responder"}) \\
\texttt{srtps-idA} &= \mathcal{H}_M(\texttt{srtps}, \texttt{"Responder"}) \\
\texttt{os-idA} &= \mathcal{H}_M(\texttt{os}, \texttt{"Responder"})
\end{aligned}$$

Alice sends these values to *Bob* in the `DHPart1` message (109/141 words depending on `keya`, see figure 2): first he checks that $\texttt{pvA} \neq 1$ and $\texttt{pvA} \neq (p-1)$, case in which the exchange is terminated[13].
After this check, *Bob* gathers the secret associated with *Alice*: he will calculate `s0` in the same way *Alice* does:

$$\texttt{mh} = \mathcal{H}(\textit{Alice}\text{'s }\texttt{Hello}|\texttt{Commit}|\texttt{DHPart1}|\texttt{DHPart2})$$
$$\texttt{s0} = \mathcal{H}(\texttt{dhss}|\texttt{s1}|\texttt{s2}|\texttt{s3}|\texttt{s4}|\texttt{s5}|\texttt{mh})$$

Like *Alice*, he calculates the HMACs as following:

$$\begin{aligned}
\texttt{rs1-idB} &= \mathcal{H}_M(\texttt{rs1}, \texttt{"Initiator"}) \\
\texttt{rs2-idB} &= \mathcal{H}_M(\texttt{rs2}, \texttt{"Initiator"}) \\
\texttt{sigs-idB} &= \mathcal{H}_M(\texttt{sigs}, \texttt{"Initiator"}) \\
\texttt{srtps-idB} &= \mathcal{H}_M(\texttt{srtps}, \texttt{"Initiator"}) \\
\texttt{os-idB} &= \mathcal{H}_M(\texttt{os}, \texttt{"Initiator"})
\end{aligned}$$

and afterwards he calculates the expected HMACs from *Alice* and compares them with the ones in the `DHPart1` message: the matching secrets are kept, the others are replaced by a null value.

[12] This is a precaution against *bid-down attacks*, which aim to make the authentication procedure rely on weaker mechanisms even when stronger ones are available.
[13] *Marvin* could inject a false `DHPart1` message to weaken the final Diffie-Hellman result.

He then assigns to each sX the corresponding value, thus defining the sorting of the secrets[14]:

$$\begin{aligned} \texttt{s1} &= \texttt{rs1} \\ \texttt{s2} &= \texttt{rs2} \\ \texttt{s3} &= \texttt{sigs} \\ \texttt{s4} &= \texttt{srtps} \\ \texttt{s5} &= \texttt{os} \end{aligned}$$

Bob calculates the Diffie-Hellman result (dhr) and dhss by hashing dhr:

$$\begin{aligned} \texttt{dhr} &= \texttt{pvA}^{\texttt{svB}} \bmod p \\ &= (g^{\texttt{svA}} \bmod p)^{\texttt{svB}} \bmod p \\ &= g^{\texttt{svA}\cdot\texttt{svB}} \bmod p \\ \texttt{dhss} &= \mathcal{H}(\texttt{dhr}) \end{aligned}$$

After this, *Bob* is actually able to compute s0.

He then sends *Alice* the DHPart2 message (109/141 words depending on keya, see figure 2): upon receipt she performs the usual check on pvB (pvB $\neq 1$, pvB $\neq (p-1)$) and then calculates the hash of pvB concatenated with her Hello message: if it is different a *Man-in-the-Middle attack* is taking place and the exchange is aborted.
After these checks *Alice* calculates dhr and finally calculates dhss and s0, after sorting her shared secrets in the same order as *Bob*:

$$\begin{aligned} \texttt{dhr} &= \texttt{pvB}^{\texttt{svA}} \bmod p \\ &= (g^{\texttt{svB}} \bmod p)^{\texttt{svA}} \bmod p \\ &= g^{\texttt{svA}\cdot\texttt{svB}} \bmod p \\ \texttt{dhss} &= \mathcal{H}(\texttt{dhr}) \end{aligned}$$

A further remark: since a Diffie-Hellman exchange affects the state of the retained shared secret cache, it is possible for only one exchange to occur at a time. In case multiple exchanges are needed — this is the case of multiple media streams[15] to be established in parallel — the subsequent one can run only after the Conf2ACK message relative to the preceding exchange has been received.

Switch to SRTP and Confirmation

Alice and *Bob* can now use s0 to generate SRTP master keys (SRTP-mkey) and salts (SRTP-msalt) — separate in each direction for each media stream — using an HMAC function and truncating the obtained values to the length required by the chosen SRTP algorithm[16]:

$$\begin{aligned} \texttt{SRTP-mkeyA} &= \mathcal{H}_M(\texttt{s0}, \texttt{"Responder SRTP master key"}) \\ \texttt{SRTP-msaltA} &= \mathcal{H}_M(\texttt{s0}, \texttt{"Responder SRTP master salt"}) \\ \texttt{SRTP-mkeyB} &= \mathcal{H}_M(\texttt{s0}, \texttt{"Initiator SRTP master key"}) \\ \texttt{SRTP-msaltB} &= \mathcal{H}_M(\texttt{s0}, \texttt{"Initiator SRTP master salt"}) \end{aligned}$$

Next they compute their HMAC keys[17] (hmackey) and the new retained secret rs0:

$$\begin{aligned} \texttt{hmackeyA} &= \mathcal{H}_M(\texttt{s0}, \texttt{"Responder HMAC key"}) \\ \texttt{hmackeyB} &= \mathcal{H}_M(\texttt{s0}, \texttt{"Initiator HMAC key"}) \\ \texttt{rs0} &= \mathcal{H}_M(\texttt{s0}, \texttt{"retained secret"}) \end{aligned}$$

After this, *Alice* and *Bob* generate the ZRTP keys (ZRTP-key), which will be destroyed only at the end of the call signaling session: this will allow ZRTP Preshared mode to generate new SRTP key-salt pairs

[14] As the non-matching secrets have been replaced by a null value, they have no effect on the cancatenation.
[15] Except when ZRTP runs in Preshared mode, see section 1.4.
[16] The default settings of ZRTP use SRTP with no MKI, 32 bit authentication using HMAC-SHA1, AES-CM 128 or 256 bit key length, 112 bit session salt key length, 2^{48} key derivation rate, and SRTP prefix length 0.
[17] These keys are used only by ZRTP, not by SRTP.

for new concurrent media streams between *Alice* and *Bob*, within the limit of the call signaling session[18].

$$\texttt{ZRTP-keyA} = \mathcal{H}_M(\texttt{s0}, \texttt{"Responder ZRTP Key"})$$
$$\texttt{ZRTP-keyB} = \mathcal{H}_M(\texttt{s0}, \texttt{"Initiator ZRTP Key"})$$

They also compute the SAS value (`sasvalue`) as the 64 least significative bits of `mh`.

At this point *Alice* and *Bob* can send the confirmation messages `Confirm1` and `Confirm2` (variable length, see figure 2), which are exchanged essentially for two reasons:

1. they confirm that the whole key agreement procedure was successful and encryption is working, and they enable automatic detection of *Man-in-the-Middle attacks*;
2. they allow the SRTP-encrypted transmission of the SAS Verified flag (`V`), so that no passive observer can learn wheter *Alice* and *Bob* have the good habit to verify the SAS.

`ConfirmX` messages contain a ciphered part composed by the cache expiration interval for `rs0`, an optional signature and an 8 bit unsigned integer (`flagoctet`), which contains the *Disclosure* flag (`D`), the *Allow clear* flag (`A`), and the *SAS Verified flag* (`V`):

$$\texttt{flagoctet} = \texttt{V} \cdot 2^2 + \texttt{A} \cdot 2^1 + \texttt{D} \cdot 2^0$$

The encrypted part of `ConfirmX` is ciphered via CFB algorithm [10, 33], using `ZRTP-key` as key: its initialization vector is sent in the message, along with an HMAC (`hmac`) covering the encrypted part:

$$\texttt{hmac} = \mathcal{H}_M(\texttt{hmackey}, \text{Encrypted part of } \texttt{ConfirmX})$$

The `Conf2ACK` message (3 words, see figure 2) is a confirmation sent by *Alice* upon receipt of `Confirm2` message.

After the confirmation procedure, both parties discard the `rs2` secret and replace it by `rs1` secret and `rs1` by `rs0`.

1.4 Key Agreement — Preshared Mode

The Preshared mode is slightly different from the Diffie-Hellman one: it is a light-weight approach allowing not to exchange Diffie-Hellman keys (and therefore not to generate Diffie-Hellman key pairs, a computationally expensive task).
In this mode, security relies on the shared secrets (in case *Alice* and *Bob* have no shared secrets the exchange is aborted), and should be used only after SAS has been performed.
The difference from Diffie-Hellman mode is that an 8 word nonce (`nonceB` must be different from all the values freshly generated since the last Diffie-Hellman exchange) is sent in the `Commit` message in place of `hvB`. *Alice* replies with another nonce (`nonceA`) in place of `pvA` in the `DHPart1` message.

The secret `s0` is therefore computed by the hash of the secrets and `mh`[19], omitting `dhss` which of course does not exist:

$$\texttt{s0} = \mathcal{H}(\texttt{s1}|\texttt{s2}|\texttt{s3}|\texttt{s4}|\texttt{s5}|\texttt{mh})$$

1.5 Session Termination

An SRTP session or a ZRTP exchange ends by means of the `GoClear` message (15 words, see figure 2).

In case of switching from SRTP to RTP, ZRTP stops relying on the SRTP authentication tag, sending an HMAC computed with `hmackey` instead:

$$\texttt{clear} = \mathcal{H}_M(\texttt{hmackey}, \texttt{"GoClear"})$$

Having separate `hmackeys` ensures that `GoClear` messages cannot be cached by *Marvin* and reflected back to the endpoint.

[18] In case of separate calls (in SIP terms, separate SIP dialogs), each call has its own `ZRTP-keys`.
[19] It implicitly includes the nonces.

The session anyway remains in secure mode until receipt of the `ClearACK` message, when both parties can start sending RTP packets. Both endpoints delete the cryptographic context, only `ZRTP-key`s remain till the end of the signaling session.

In case of a ZRTP exchange termination, the `reason` string contains the reason why that event occured; if the event occurs before the key agreement was successfully performed, `clear` is a bunch of zeros, because there exists no `hmackey`.

2 AVISPA Model

AVISPA[20] is a cryptographic protocol verifier, developed by:

- Artificial Intelligence Laboratory, DIST, University of Genova, Italy;
- Eidgenössische Technische Hochschule Zürich (ETHZ), Information Security Group, Department of Computer Science, Zürich, Switzerland;
- CASSIS, joint INRIA team from LORIA Nancy and LIFC Besançon, France;
- Siemens Aktiengesellschaft, Munich, Germany.

The tool processes input files in IF format [16] or in HLPSL format [15].
The latter is a higher level format (in fact it has to be translated into IF format before the protocol can actually be analyzed) and therefore is the one used here.

The AVISPA analysis relies on four different back-ends:

1. On-the-fly Model-Checker (OFMC);
2. CL-based Attack Searcher (CL-AtSe);
3. SAT-based Model-Checker (SATMC);
4. Tree Automata-based Protocol Analyser (TA4SP).

Only the first two back-ends have been used here, as the others do not support exponentiation.

OFMC is a tool performing protocol verification through the exploration of the transition system described in the protocol specification on a bounded number of session. [17]

CL-AtSe translates the protocol specification into constraints and runs it over a finite number of iterations, after reducing it by means of simplification heuristics and redundancy elimination techniques. [18]

2.1 The Model

The protocol has been modeled in the case when no secret mismatch is detected (the constant `secretsab`, unknown to the intruder, is passed to both *Alice* and *Bob*).
Moreover, *Bob* does not choose the hash function (and therefore the HMAC) depending on *Alice*'s `Hello` message, as strings and secrets are joined into cumulative dummy constants[21]: the hash function is instead fixed and publicly known.

The requested goals are *secrecy*[22] of the encrypted texts exchanged in the confirmation messages and secrecy of `S0` (anyway this is redundant as the encrypted texts can be secret only if `S0` is also secret).
Moreover *Alice authenticates*[23] *Bob* on `S0` and viceversa (this is also redundant as it can be inferred from the fact that only *Alice* and *Bob* know the shared secrects).

[20] Automated Validation of Internet Security Protocols and Applications.
[21] It would make no sense giving them a meaning, as they will have to be processed by the program using ZRTP that runs on the endpoints' machines.
[22] Secrecy is accomplished when an attacker cannot get to know a secret value.
[23] Authentication is accomplished when an attacker cannot pretend to be the authenticated user.

HLPSL Code

```
role alice(
    A,B : agent,                              % Agents
    SECRETS : text,                           % A's Secrets
    STRINGA : text,                           % A's Capabilities & IDs
    H,HM : hash_func,                         % Chosen Hash and HMAC functions
    SEND,RECV : channel(dy))                  % Channels

 played_by A def=

 local State : nat,                           % State
       STRINGB : text,                        % B's Capabilities & IDs
       SECRETSIDA,SECRETSIDB : message,       % Secret IDs
       S0 : message,                          % New secret
       COMMITSTRING : text,                   % Commit message
       HVB,MH : message,                      % Stuff
       ZRTPKEYA,ZRTPKEYB: message,            % ZRTP keys
       CIPHBLOCKA,CIPHBLOCKB : message,       % CFB
       PVA,PVB : message,                     % Public keys
       SVA : text                             % A's Secret key

 init State := 0

 transition

 1. State = 0 /\ RECV(start) =|>
  State' := 2 /\ SEND(hello.STRINGA)

 2. State = 2 /\ RECV(helloack.hello.STRINGB') =|>
  State' := 4 /\ SEND(helloack)

 3. State = 4 /\ RECV(commit.COMMITSTRING'.HVB') =|>
  State' := 6 /\ SECRETSIDA' := HM(SECRETS.responder)
              /\ SVA':=new()
              /\ PVA' := exp(g,SVA')
              /\ SEND(dhpart1.PVA'.SECRETSIDA')

 4. State = 6 /\ RECV(dhpart2.PVB'.SECRETSIDB')
              /\ SECRETSIDB' = HM(SECRETS.initiator)
              /\ HVB = H(PVB'.hello.STRINGA) =|>
  State' := 8 /\ MH' := H(hello.STRINGA.commit.COMMITSTRING.dhpart1.PVA.
                 SECRETSIDA.dhpart2.PVB'.SECRETSIDB')
              /\ S0' := H(H(exp(PVB,SVA)).SECRETS.MH')
              /\ secret(S0',s0,{A,B})
              /\ witness(A,B,bob_alice_s0,S0')
              /\ ZRTPKEYA' := HM(S0'.rzrtp)
              /\ CIPHBLOCKA' := new()
              /\ SEND(confirm1.HM(CIPHBLOCKA').cfbivA.{CIPHBLOCKA'}_ZRTPKEYA')
              /\ secret(CIPHBLOCKA',ciphA,{A,B})

 5. State = 8 /\ RECV(confirm2.HM(CIPHBLOCKB').cfbivB.{CIPHBLOCKB'}_ZRTPKEYB')
              /\ ZRTPKEYB' = HM(S0.izrtp) =|>
 State' := 10 /\ SEND(conf2ack)
              /\ request(A,B,alice_bob_s0,S0)
```

```
end role

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

role bob(
    A,B : agent,                              % Agents
    SECRETS : text,                           % A's Secrets
    STRINGB : text,                           % A's Capabilities & IDs
    H,HM : hash_func,                         % Chosen Hash and HMAC functions
    COMMITSTRING : text,                      % Commit message
    SEND,RECV : channel(dy))                  % Channels

 played_by B def=

 local State : nat,                           % State
       STRINGA : text,                        % B's Capabilities & IDs
       SECRETSIDA,SECRETSIDB : message,       % Secret IDs
       S0 : message,                          % New secret
       HVB,MH : message,                      % Stuff
       ZRTPKEYA,ZRTPKEYB: message,            % ZRTP keys
       CIPHBLOCKA,CIPHBLOCKB : message,       % CFB
       PVA,PVB : message,                     % Public keys
       SVB : text                             % B's Secret key

 init State := 1

 transition

 1. State = 1 /\ RECV(hello.STRINGA') =|>
  State' := 3 /\ SEND(helloack.hello.STRINGB)

 2. State = 3 /\ RECV(helloack) =|>
  State' := 5 /\ SVB':=new()
              /\ PVB' := exp(g,SVB')
              /\ HVB' := H(PVB'.hello.STRINGA)
              /\ SEND(commit.COMMITSTRING.HVB')

 3. State = 5 /\ RECV(dhpart1.PVA'.SECRETSIDA')
              /\ SECRETSIDA' = HM(SECRETS.responder) =|>
  State' := 7 /\ SECRETSIDB' := HM(SECRETS.initiator)
              /\ SEND(dhpart2.PVB.SECRETSIDB')

 4. State = 7 /\ RECV(confirm1.HM(CIPHBLOCKA').cfbivA.{CIPHBLOCKA'}_ZRTPKEYA')
              /\ MH' = H(hello.STRINGA.commit.COMMITSTRING.dhpart1.PVA.
                         SECRETSIDA.dhpart2.PVB.SECRETSIDB')
              /\ S0' = H(H(exp(PVA,SVB)).SECRETS.MH')
              /\ ZRTPKEYA' = HM(S0'.rzrtp) =|>
  State' := 9 /\ CIPHBLOCKB' := new()
              /\ ZRTPKEYB' := HM(S0'.izrtp)
              /\ SEND(confirm2.HM(CIPHBLOCKB').cfbivB.{CIPHBLOCKB'}_ZRTPKEYB')
              /\ secret(CIPHBLOCKB',ciphB,{A,B})
              /\ request(B,A,bob_alice_s0,S0)
              /\ secret(S0',s0,{A,B})
              /\ witness(B,A,bob_alice_s0,S0')

 5. State = 9 /\ RECV (conf2ack) =|>
```

```
 State' := 11

end role

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

role session(
 A,B : agent,
 SECRETSA,SECRETSB : text,
 STRINGA,STRINGB : text)

 def=

 local SA,SB,RA,RB : channel(dy)

 const hello,helloack,commit,dhpart1,dhpart2,confirm1,confirm2,conf2ack,
       responder,initiator,rzrtp,izrtp,commitstring,cfbivA,cfbivB: text,
       g : nat,
       h,hm : hash_func

 composition

 alice(A,B,SECRETSA,STRINGA,h,hm,SA,RA)
 /\ bob(A,B,SECRETSB,STRINGB,h,hm,commitstring,SB,RB)

end role

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
role environment()

 def=

 const a,b : agent,
 secretsab,secretsai,secretsib : text,
 stringa,stringb,stringi : text,
 s0,ciphA,ciphB,alice_bob_s0,bob_alice_s0 : protocol_id

 intruder_knowledge = {a,b,h,hm,secretsai,secretsib,stringa,stringb,stringi,
                       hello,helloack,commit,dhpart1,dhpart2,confirm1,
                       confirm2,conf2ack,responder,initiator,rzrtp,izrtp,
                       commitstring,cfbivA,cfbivB,g}

 composition

    session(a,b,secretsab,secretsab,stringa,stringb)
 /\ session(a,i,secretsai,secretsai,stringa,stringi)
 /\ session(i,b,secretsib,secretsib,stringi,stringb)

 end role

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
goal

 secrecy_of s0
 secrecy_of ciphA
 secrecy_of ciphB
```

```
 authentication_on alice_bob_s0
 authentication_on bob_alice_s0

end goal

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
environment()
```

2.2 Analysis Output

Here is the output of the analysis, which has been performed with OFMC and CL-AtSe back-ends.

On-the-fly Model-Checker Output

```
% OFMC
% Version of 2006/02/13
SUMMARY
  SAFE
DETAILS
  BOUNDED_NUMBER_OF_SESSIONS
PROTOCOL
  /avispa-1.1/testsuite/results/zrtp.if
GOAL
  as_specified
BACKEND
  OFMC
COMMENTS
STATISTICS
  parseTime: 0.00s
  searchTime: 144.82s
  visitedNodes: 17301 nodes
  depth: 16 plies
```

CL-based Attack Searcher Output

```
SUMMARY
  SAFE

DETAILS
  BOUNDED_NUMBER_OF_SESSIONS
  TYPED_MODEL

PROTOCOL
  /avispa-1.1/testsuite/results/zrtp.if

GOAL
  As Specified

BACKEND
  CL-AtSe

STATISTICS

  Analysed   : 20549 states
  Reachable  : 8015 states
  Translation: 0.08 seconds
```

```
Computation: 21.74 seconds
```

2.3 Results and Considerations

None of the two back-ends succeded in finding an attack on the protocol.

By slightly modifying the HLPSL code, `secretsab` can be added to the intruder knowledge: this happens, for example, when all the secrets mismatch and all of them are discarded.
Obviously, in this case the back-ends find that a *Man-in-the-Middle attack* is possible: that is why the SAS is needed whenever the endpoints cannot rely on the shared secrets.

3 ProVerif Model

ProVerif is a cryptographic protocol verifier, developed by Bruno Blanchet[24]. [19]

The tool processes input files formatted as a sequence of Horn clauses or as a process in the applied π-calculus[25], which will be translated into Horn clauses before being run: the latter is the format used here, as described in [21], [22] and [23].

3.1 The Model

The protocol has been modeled almost in the same way as for AVISPA (see section 2.1): no secret mismatch is detected (a new name `secrets`, unknown to the attacker, is used by each couple of *Alice* and *Bob* processes), *Bob* does not choose the hash function depending on *Alice*'s `Hello` message (it is instead fixed and publicly known) and strings and secrets are joined into cumulative public variables.
A different channel is used for each message, so that the information of the *Message Type Block* is contained in the channel used.

The queries involve the *secrecy* of the encrypted blocks exchanged in the confirmation messages (and therefore the secrecy of the key material used to encrypt them).

For completeness, also *strong secrecy*[26] is queried for these encrypted blocks.
To prove strong secrecy, these blocks are given a set of possible values each: if these blocks contain the optional signature suggested in the latest ZRTP protocol draft as an alternative to SAS, they are unlikely to assume the same values, so the intersection of the possible values is an empty set; if the optional signature is not used, this intersection may not be empty as these blocks contain only two parameters (plus the signature length, which is zero) that are not unlikely to be the same.

Applied π-calculus Process

$Alice =$		PROCESS FOR ALICE
	$\overline{helloa}\langle \texttt{STRINGA} \rangle$.	*send* `Hello` *message*
	$helloacka(\texttt{HELLOACKBA})$.	*wait for Bob's* `HelloACK` *message*
	$hellob(\texttt{STRINGB})$.	*wait for Bob's* `Hello` *message*
	$\overline{helloackb}\langle \texttt{HELLOACKA} \rangle$.	*send* `HelloACK` *message*
	$commit((\texttt{COMMITSTRINGBA}, \texttt{HVBA}))$.	*wait for* `Commit` *message*
	$\nu \texttt{SVA}$.	*generate secret value*
	let $\{\texttt{PVA} = g^{\texttt{SVA}}\}$ *in*	*compute public value*
	let $\{\texttt{SECRETSIDA} = \mathcal{H}_M(\texttt{secrets}, \texttt{responder})\}$ *in*	*compute her IDs of the secrets*
	$\overline{dhpart1}\langle (\texttt{PVA}, \texttt{SECRETSIDA}) \rangle$.	*send* `DHPart1` *message*

[24] Researcher in Informatics, CNRS (Département d'Informatique) — École Normale Supérieure Paris.
[25] The applied π-calculus is an extension of the original π-calculus by Robin Milner [20], which has been developed by Martín Abadi and Cédric Fournet. [21]
[26] Strong secrecy is accomplished when two processes holding a secret cannot be distinguished if the value of the secret changes (the processes are *observationally equivalent*) [24], i.e. an attacker knowing a possible value of secret data cannot tell if that data is the one actually sent.

$dhpart2((\mathtt{PVBA}, \mathtt{SECRETSIDBA}))$. — *wait for DHPart2 message*
if $\mathtt{SECRETSIDBA} = \mathcal{H}_M(\mathtt{secrets}, \mathtt{initiator})$ *then* — *check Bob's IDs of the secrets*
if $\mathtt{HVBA} = \mathcal{H}(\mathtt{PVBA}, \mathtt{hello}, \mathtt{STRINGA})$ *then* — *check HVB*
let $\{\mathtt{MHA} = \mathcal{H}(\mathtt{STRINGA}, \mathtt{COMMITSTRINGBA}, \mathtt{PVA}, \mathtt{SECRETSIDA}, \mathtt{PVBA}, \mathtt{SECRETSIDBA})\}$ *in* — *compute the message hash*
let $\{\mathtt{SOA} = \mathcal{H}(\mathcal{H}(\mathtt{PVBA}^{\mathtt{SVA}}), \mathtt{secrets}, \mathtt{MHA})\}$ *in* — *compute S0*
let $\{\mathtt{ZRTPKEYA} = \mathcal{H}_M(\mathtt{SOA}, \mathtt{rzrtp})\}$ *in* — *compute her ZRTP key*
$\overline{confirm1}\langle(\mathcal{H}_M(\mathtt{SOA}, \mathtt{CIPHBLOCKA}), \mathtt{cfbiva}, \mathcal{E}_{\mathtt{ZRTPKEYA}}(\mathtt{CIPHBLOCKA}))\rangle$. — *send Confirm1 message*
$confirm2((\mathtt{HMACCIPHBLOCKBA}, \mathtt{CFBIVBA}, \mathtt{ENCRYPTEDCIPHBLOCKBA}))$. — *wait for Confirm2 message*
let $\{\mathtt{ZRTPKEYBA} = \mathcal{H}_M(\mathtt{SOA}, \mathtt{izrtp})\}$ *in* — *compute Bob's ZRTP key*
let $\{\mathtt{CIPHBLOCKBA} = \mathcal{D}_{\mathtt{ZRTPKEYBA}}(\mathtt{ENCRYPTEDCIPHBLOCKBA})\}$ *in* — *decrypt Bob's secret block*
if $\mathtt{HMACCIPHBLOCKBA} = \mathcal{H}_M(\mathtt{SOA}, \mathtt{CIPHBLOCKBA})$ *then* — *check integrity of Bob's secret block*
$\overline{conf2ack}\langle\mathtt{CONF2ACKA}\rangle$. — *send Conf2ACK message*

Bob = — Process for Bob
$helloa(\mathtt{STRINGAB})$. — *wait for Alice's Hello message*
$\overline{helloacka}\langle\mathtt{HELLOACKB}\rangle$. — *send HelloACK message*
$\overline{hellob}\langle\mathtt{HELLOB}\rangle$. — *send Hello message*
$helloackb(\mathtt{HELLOACKAB})$. — *wait for Alice's HelloACK message*
$\nu\mathtt{SVB}$. — *generate secret value*
let $\{\mathtt{PVB} = g^{\mathtt{SVB}}\}$ *in* — *compute public value*
let $\{\mathtt{HVB} = \mathcal{H}(\mathtt{PVB}, \mathtt{STRINGAB})\}$ *in* — *compute HVB*
$\overline{commit}\langle(\mathtt{COMMITSTRINGB}, \mathtt{HVB})\rangle$. — *send Commit message*
$dhpart1((\mathtt{PVAB}, \mathtt{SECRETSIDAB}))$. — *wait for DHPart1 message*
if $\mathtt{SECRETSIDAB} = \mathcal{H}_M(\mathtt{secrets}, \mathtt{responder})$ *then* — *check Alice's IDs of the secrets*
let $\{\mathtt{SECRETSIDB} = \mathcal{H}_M(\mathtt{secrets}, \mathtt{initiator})\}$ *in* — *compute his IDs of the secrets*
$\overline{dhpart2}\langle(\mathtt{PVB}, \mathtt{SECRETSIDB})\rangle$. — *send DHPart2 message*
$confirm1((\mathtt{HMACCIPHBLOCKAB}, \mathtt{CFBIVAB}, \mathtt{ENCRYPTEDCIPHBLOCKAB}))$. — *wait for Confirm1 message*
let $\{\mathtt{MHB} = \mathcal{H}(\mathtt{STRINGAB}, \mathtt{COMMITSTRINGB}, \mathtt{PVAB}, \mathtt{SECRETSIDAB}, \mathtt{PVB}, \mathtt{SECRETSIDB})\}$ *in* — *compute the message hash*
let $\{\mathtt{SOB} = \mathcal{H}(\mathcal{H}(\mathtt{PVAB}^{\mathtt{SVB}}), \mathtt{secrets}, \mathtt{MHB})\}$ *in* — *compute S0*
let $\{\mathtt{ZRTPKEYAB} = \mathcal{H}_M(\mathtt{SOB}, \mathtt{rzrtp})\}$ *in* — *compute Alice's ZRTP key*
let $\{\mathtt{CIPHBLOCKAB} = \mathcal{D}_{\mathtt{ZRTPKEYAB}}(\mathtt{ENCRYPTEDCIPHBLOCKAB})\}$ *in* — *decrypt Alice's secret block*
if $\mathtt{HMACCIPHBLOCKAB} = \mathcal{H}_M(\mathtt{SOB}, \mathtt{CIPHBLOCKAB})\}$ *then* — *check integrity of Alice's secret block*
let $\{\mathtt{ZRTPKEYB} = \mathcal{H}_M(\mathtt{SOB}, \mathtt{izrtp})\}$ *in* — *compute his ZRTP key*
$\overline{confirm2}\langle(\mathcal{H}_M(\mathtt{SOB}, \mathtt{CIPHBLOCKB}), \mathtt{cfbivb}, \mathcal{E}_{\mathtt{ZRTPKEYB}}(\mathtt{CIPHBLOCKB}))\rangle$. — *send Confirm2 message*
$conf2ack(\mathtt{CONF2ACKAB})$. — *wait for Conf2ACK message*

ProVerif Code (Applied π-calculus Format)

```
(************************ D E C L A R A T I O N S ***********************)

(* Diffie-Hellman exponentials *)
data g/0.
fun exp/2.
equation exp(exp(g,y),z)=exp(exp(g,z),y).

(* Hash and HMAC *)
fun hash/1.
fun hmac/2.
```

```
(* Encryption and Decryption *)
fun encrypt/2.
reduc decrypt(encrypt(x,y),y) = x.

(* Channels *)
free helloa,helloacka,hellob,helloackb,commit,dhpart1,dhpart2,confirm1,
     confirm2,conf2ack.

(* Constants *)
data izrtp/0.
data rzrtp/0.
data cfbiva/0.
data cfbivb/0.
data initiator/0.
data responder/0.
data STRINGA/0.
data STRINGB/0.
data COMMITSTRINGB/0.
data HELLOACKA/0.
data HELLOACKB/0.
data CONF2ACKA/0.

(* Encrypted blocks for Confirm messages *)
private free CIPHBLOCKA,CIPHBLOCKB.

(* Pool of possible values for the encrypted blocks *)
free CBA1,CBA2.
free CBB1,CBB2.


(***************************** Q U E R I E S *****************************)

(***** Secrecy *****)
query attacker: CIPHBLOCKA; attacker: CIPHBLOCKB.


(***** Strong secrecy *****)

(* CIPHBLOCKX contains optional signature: no common values *)
noninterf CIPHBLOCKA among(CBA1,CBA2), CIPHBLOCKB among(CBB1,CBB2).

(* CIPHBLOCKX contains no optional signature: at least 1 common value *)
noninterf CIPHBLOCKA among(CBA1,CBA2), CIPHBLOCKB among(CBB1,CBA2).


(***************************** P R O C E S S *****************************)

let Alice=
        out(helloa,STRINGA);
        in(helloacka,HELLOACKBA);
        in(hellob,STRINGBA);
        out(helloackb,HELLOACKA);
        in(commit,(COMMITSTRINGBA,HVBA));
        new SVA;
        let PVA=exp(g,SVA) in
```

```
        let SECRETSIDA=hmac(secrets,responder) in
        out(dhpart1,(PVA,SECRETSIDA));
        in(dhpart2,(PVBA,SECRETSIDBA));
        if SECRETSIDBA=hmac(secrets,initiator) then
        (
          if HVBA=hash((PVBA,STRINGA)) then
          (
            let MHA=hash((STRINGA,COMMITSTRINGBA,PVA,SECRETSIDA,PVBA,
                          SECRETSIDBA)) in
            let SOA=hash((hash(exp(PVBA,SVA)),secrets,MHA)) in
            let ZRTPKEYA=hmac(SOA,rzrtp) in
            out(confirm1,(hmac(SOA,CIPHBLOCKA),cfbiva,encrypt(CIPHBLOCKA,
                   ZRTPKEYA)));
            in(confirm2,(HMACCIPHBLOCKBA,CFBIVBA,ENCRYPTEDCIPHBLOCKBA));
            let ZRTPKEYBA=hmac(SOA,izrtp) in
            let CIPHBLOCKBA=decrypt(ENCRYPTEDCIPHBLOCKBA,ZRTPKEYBA) in
            if hmac(SOA,CIPHBLOCKBA)=HMACCIPHBLOCKBA then
            (
              out(conf2ack,CONF2ACKA)
            )
          )
        ).

let Bob=
        in(helloa,STRINGAB);
        out(helloacka,HELLOACKB);
        out(hellob,STRINGB);
        in(helloackb,HELLOACKAB);
        new SVB;
        let PVB=exp(g,SVB) in
        let HVB=hash((PVB,STRINGAB)) in
        out(commit,(COMMITSTRINGB,HVB));
        in(dhpart1,(PVAB,SECRETSIDAB));
        if SECRETSIDAB=hmac(secrets,responder) then
        (
           let SECRETSIDB=hmac(secrets,initiator) in
          out(dhpart2,(PVB,SECRETSIDB));
          in(confirm1,(HMACCIPHBLOCKAB,CFBIVAB,ENCRYPTEDCIPHBLOCKAB));
          let MHB=hash((STRINGAB,COMMITSTRINGB,PVAB,SECRETSIDAB,PVB,
                        SECRETSIDB)) in
          let SOB=hash((hash(exp(PVAB,SVB)),secrets,MHB)) in
          let ZRTPKEYAB=hmac(SOB,rzrtp) in
          let CIPHBLOCKAB=decrypt(ENCRYPTEDCIPHBLOCKAB,ZRTPKEYAB) in
          if hmac(SOB,CIPHBLOCKAB)=HMACCIPHBLOCKAB then
          (
            let ZRTPKEYB=hmac(SOB,izrtp) in
            out(confirm2,(hmac(SOB,CIPHBLOCKB),cfbivb,encrypt(CIPHBLOCKB,
                   ZRTPKEYB)));
            in(conf2ack,CONF2ACKAB)
          )
        ).

(* Main *)
process
        !(new secrets;             (* generate the shared secrets *)
        ((!Alice)|(!Bob)))
```

3.2 Analysis Output

Here is the output of the analysis[27], splitted for convenience in a subsection for each query (the *Linear part* is only in the first subsection).

Secrecy

```
Linear part:
exp(exp(g(),y_5),z_6) = exp(exp(g(),z_6),y_5)
Completing equations...
Completed equations:
exp(exp(g(),y_5),z_6) = exp(exp(g(),z_6),y_5)
Convergent part:
Completing equations...
Completed equations:
Completed destructors:
decrypt(encrypt(x_51,y_52),y_52) => x_51
-- Secrecy & events.
Starting rules:

   [...]

Completing...
200 rules inserted. The rule base contains 118 rules. 8 rules in the queue.
Starting query not attacker:CIPHBLOCKA[]
RESULT not attacker:CIPHBLOCKA[] is true.
Starting query not attacker:CIPHBLOCKB[]
RESULT not attacker:CIPHBLOCKB[] is true.
```

Strong Secrecy - Secret Blocks Containing Signature

```
-- Non-interference on CIPHBLOCKA CIPHBLOCKB
Starting rules:

   [...]

Completing...
200 rules inserted. The rule base contains 141 rules. 40 rules in the queue.
400 rules inserted. The rule base contains 280 rules. 45 rules in the queue.
RESULT Equivalence proof succeeded (bad not derivable).
```

Strong Secrecy - Secret Blocks Not Containing Signature

```
-- Non-interference on CIPHBLOCKA CIPHBLOCKB
Starting rules:

   [...]

Completing...
200 rules inserted. The rule base contains 141 rules. 40 rules in the queue.
400 rules inserted. The rule base contains 280 rules. 45 rules in the queue.
goal reachable: bad:

   [...]

RESULT Equivalence proof failed (bad derivable).
```

[27] Shortened by removing all the generated rules.

3.3 Results and Considerations

The queries confirmed that the secrecy of the encrypted data exchanged during the protocol run is preserved.

In addition the queries confirmed that strong secrecy is accomplished when signatures are exchanged in the secret blocks[28], so even an attacker having a pool of possible valid blocks cannot tell which block is the one actually sent.

Viceversa, if the endpoints do not exchange signatures in the `ConfirmX` messages, the ciphered blocks are only made of *Signature Length* (null value), *Cache Expiration Interval* and `flagoctet` so there is a high probability for the endpoints to send the same block[29].
In this case the proof fails: the attacker in fact can see when the two blocks are identical[30], though he still cannot understand which block among the possible ones appearing in both lists — if more than one — is the one actually sent.

4 Conclusion

Nor AVISPA, neither ProVerif have succeeded in finding possible attacks on the protocol model, which has proven to be robust and not vulnerable to attacks throughout its run, under the hypothesis that the endpoints share the secrets needed to generate the key.

In case this hypothesis is removed, the protocol is still secure under the assumption that SAS provides the means to detect a *Man-in-the-Middle*.
This assumption holds if the attacker has no voice-forgery capabilities[31]: a *Man-in-the-Middle* could attack the protocol if he were able to generate a collision on `sasvalue` (the first 64 bits of the message hash `mh`), but this is not possible as neither party can deterministically influence `sasvalue`. In fact the initiator chooses his public value and sends a "hint" of it in the `Commit` message and therefore he cannot change it upon receipt of the responder's public value — being able to do so would imply being able to choose a different public value which would result in the same"hint", thus being able to find a collision in the hash function and this is assumed to be computationally infeasible.

For the same reason, security against *bid-down attacks* is provided by including the initiator's `Hello` message in the "hint", which is sent upon the hash commitment, and again when generating the secret `S0`.

4.1 Analysis with AVISPA and with ProVerif

Both tools use the Dolev-Yao model for channels, in which the intruder can do whatever he pleases with the exchanged messages, including altering them, preventing them to reach their destination or replaying them whenever he wants: a way to see this is to think of a net modeled as a star, with the intruder in the central node.

In both tools each single ZRTP protocol run has been modeled as two concurrent processes that synchronize by means of message exchange: the arrival of an awaited message lets a process get to the following state; in both cases the underlying hypotheses are that there is no shared secret mismatch and that the hash function is chosen *a priori*.

Here are the differences between performing the analysis with AVISPA and Proverif:

- endpoint authentication and key secrecy have been proven only in AVISPA, as these are redundant proves (authentication is given because only the two endpoints can have the secrets; if the encrypted data is secret also key secrecy must be accomplished);

[28] This is the first `noninterf` declaration, when the list of possible values for `CIPHBLOCKA` is entirely different from the one of possible values for `CIPHBLOCKB`.
[29] This is the second `noninterf` declaration, when the lists of possible values for `CIPHBLOCKA` and for `CIPHBLOCKB` have one — at least one, generally speaking — value in common.
[30] He just has to check whether `HMACCIPHBLOCKA` equals `HMACCIPHBLOCKB`, as in this case they would be the HMAC of the same value under the same key.
[31] This case will be discussed in section 4.2.

- in AVISPA the endpoints communicate through the same channel and the message type is identified by a string, in ProVerif each message is sent on a different channel;
- ProVerif allows analysing the protocol on an unbounded number of session, as the instantiation of the endpoints has been made with the (infinite) replication operator "!", in AVISPA the used back-ends are limited to a bounded number of sessions.

One thing about these tools — more from the user's point of view rather than from a technical one — is that on one hand one has to say that AVISPA is much better documented than ProVerif and therefore easier to learn (no wonder, AVISPA has been developed by more people), on the other hand it might be easier to approach ProVerif if one already knows the applied π-calculus, as there is no need to learn a different protocol specification language.

4.2 SAS: the Achilles' heel of ZRTP

If the assumption that SAS provides the means to detect a *Man-in-the-Middle* is removed — and this is the case of a voice-forgery capable attacker, which in not unreasonable though it requires him to be really skilled, powerful and resourceful — some possible attacks have been found through an analysis in the Murphi model checker[32] carried out by Andrew Schwartz and Jeremy Robin on a pre-realease version of ZRTP [11]: the following table 4.2 contains the attacks that can be performed, depending on the attacker's capabilities.

	VF	BC		6M	CR	HCCR	
Attacker can convert voice to his own in real time	○	○	○	○	●	●	●
Initiator forgets voice between sessions	○	○	●	●	○	○	○
Responder forgets voice between sessions	○	●	○	●	○	○	○
Attacker can forge Initiator's voice	●	○	●	○	○	○	●
Attacker can forge Responder's voice	●	●	○	○	○	●	○
Initiator does not know Responder's voice in advance	○	○	●	●	●	○	●
Responder does not know Initiator's voice in advance	○	●	○	●	●	●	○

Table 2: Possible attacks on ZRTP. [11]

It is worth spending a few words to present the attacks found:

Voice Forgery Attack can be performed when *Marvin* is able to imitate the voices of *Alice* and *Bob*: at the SAS phase he authenticates with each party using different SAS strings which are read with the other party's voice;

Bill Clinton Attack is based on the assumption that a celebrity will not remember the voice of a random person such as *Alice*, while *Alice* will remember the celebrity's voice: *Marvin* can therefore execute SAS with the celebrity with his own voice to create a false shared secret and when *Alice* will later try to contact the celebrity, *Marvin* will authenticate to *Alice* by imitating the celebrity's voice. This schema can work for each random person without knowing in advance who this random person will be, *Marvin* only needs the celebrity to forget voices;

6 Month Attack is based on the assumption that *Alice* and *Bob* will not remember for a long time their voices, as they are unfamiliar and they do not happen to talk very often: in this case *Marvin* can establish false shared secrets with each party — as made with the celebrity in the *Bill Clinton attack* — so they will not execute SAS on the following sessions and *Marvin* will be able to listen to the traffic and simply relay it;

Court Reporter Attack can be performed if *Alice* and *Bob* do not know each other's voice: *Marvin* establishes a session with each one of them and, after performing SAS with his own voice, he has to relay the traffic in both directions in his own voice. It is easier to perform this attack with two *court reporters* and two *talkers*, because delay margins are slightly less restrictive;

[32] The Murphi model checker (`http://verify.stanford.edu/dill/murphi.html`) is a finite-state machine verification tool developed by the Stanford University and the University of Utah.

Hybrid Clinton-Court Reporter Attack is a mix of *Court Reporter* and *Bill Clinton attacks* and can be performed when *Marvin* can forge *Alice*'s voice and *Alice* does not know *Bob*'s voice, but *Bob* knows *Alice*'s: *Marvin* do not need to be a *court reporter* if he is able to imitate *Alice*'s voice.

4.3 A trusted server for SAS

The Achilles' heel of ZRTP protocol is SAS authentication, as no protocol can provide protection against an active attacker to both initiator and responder.

A possible solution to this problem could be the one proposed in the latest ZRTP protocol draft, that consists in sending a signature in the encrypted part of the `ConfirmX` messages: this is a solution that anyway requires some kind of underlying Public-Key Infrastructure.

Another possible solution, that still does not require a Public-Key Infrastructure, could be accepting to rely on a third trusted party only when no secret shared between the endpoints is available.
This solution could be implemented through a protocol which allows only to verify that the trusted party is not being emulated by the attacker (this is quite an easy requirement to accomplish, as the trusted server would be publicly known).

Here is the possible scheme:

1. *Alice* sends *Trent* (the trusted server) the SAS value she has computed (for additional security she could send the entire message hash `mh`) along with her and *Bob*'s `ZID`s;
2. *Bob* sends *Trent* the SAS value he has computed (or `mh`) along with his and *Alice*'s `ZID`s;
3. *Trent* verifies that the same task was not performed in the preceding 60 seconds (*Marvin* should perform this procedure with one endpoint at a time, but this would introduce a great delay which would prevent him from passing undetected) and that the values associated to the (`ZIDA`,`ZIDB`) pair match;
4. *Trent* returns the result of the comparison to both *Alice* and *Bob*.

In this way one of the following cases happens:

- *Alice* and *Bob* manage to talk;
- *Alice* talks to *Marvin* pretending to be *Bob*, and *Bob* gets an error from *Trent*;
- *Bob* talks to *Marvin* pretending to be *Alice*, and *Alice* gets an error from *Trent*;
- both *Alice* and *Bob* get an error from *Trent*.

The actual messages exchanged from each client to *Trent* could be a simple concatenation of string:

$$\texttt{authreqA} = \texttt{authreqB} = (\texttt{ZIDA}|\texttt{ZIDB}|\texttt{sasvalue})$$

After *Trent* has received the authentication request messages, he only has to make an almost unexpensive operation such as comparison, after checking that the same couple of `ZID`s has not requested authentication in the preceding minute.
Only if the check is positive and the comparison succeeds[33], *Trent* sends back a signature (for example the hash of `authreqX` encrypted with his private key `svT`), otherwise he sends a bunch of zeros (or a random value) to notify the authentication failure:

$$\texttt{authresult} = \{\mathcal{H}(\texttt{authreqX})\}_{\texttt{svT}}$$

Upon receipt of `authresult` *Alice* and *Bob* decrypt it with *Trent*'s public key `pvT` and check the hash correctness, thus being able to validate or reject the key exchange.

[33] This makes the server resistant to *Denial of Service Attacks*, both memory-exhausting and computational.
If the same couple or the same IP(s) tries to authenticate an excessive number of times — even at minute intervals — this could be an attack, and the server may decide to blacklist the couple or the IP(s).

In the authentication request messages one coud also send the hash of the concatenation of `ZID`s and `sasvalue` along with a string indentifing the `ZID` pair[34] in order to transmit less data, thus leaving to *Trent* only the operation of signing the hash and sending the signature: this does not compromise the authentication in principle, although this may result in an increase in the probability of a collision, especially if *Trent* has many clients to authenticate.

This way of authenticating, though requiring a trusted server, has the following benefits:

- it does not require the user to interact with the protocol;
- a *Man-in-the-Middle* has no chances to pass undetected;
- the protocol is immune to the *Voice Forgery attack*;
- the protocol is immune to the *Court Reporter attack*;
- the protocol is immune to the *Hybrid Clinton-Court Reporter attack*;
- it still does not need a public key infrastucture;
- the trusted server is rather simple, as it does not need to keep big databases or perform expensive computations;
- from a more practical point of view, the server load is unlikely to be too heavy, as it is limited to a few packets to transmit and receive, a hashing operation and an encryption for each couple.

To ensure immunity — or at least to be more resistant — to *6 Month* and *Bill Clinton attacks*, the cache expiration interval should be lowered or set to `0`, though in this last case this implies that *Trent* will be required to authenticate the endpoints each time they want to connect.
As the user is not required to interact with the protocol, the task of authenticating the endpoints can be performed more often, without annoying the users.

Bibliography

[1] Phil Zimmermann, *ZRTP: Extension to RTP for Diffie-Hellman Key Agreement for SRTP*, —— , 2007 — `http://www.ietf.org/internet-drafts/draft-zimmermann-avt-zrtp-03.txt`

[2] *The Zfone Project* website — `http://zfoneproject.com/`

[3] *Secure Hash Standard*, Federal Information Processing Standards Publication 180-2, 2002 — `http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf`

[4] H. Krawczyk, M. Bellare, R. Canetti, *HMAC: Keyed-Hashing for Message Authentication*, —— , 1997 — `http://www.ietf.org/rfc/rfc2104.txt`

[5] M. Baugher, D. McGrew, M. Naslund, E. Carrara, K. Norrman, *The Secure Real-time Transport Protocol (SRTP)*, —— , 2004 — `http://www.ietf.org/rfc/rfc3711.txt`

[6] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, E. Schooler, *SIP: Session Initiation Protocol*, —— , 2002 — `http://www.ietf.org/rfc/rfc3261.txt`

[7] D. Harkins, D. Carrel, *The Internet Key Exchange (IKE)*, —— , 1998 — `http://www.ietf.org/rfc/rfc2409.txt`

[8] T. Kivinen, M. Kojo, *More Modular Exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE)*, —— , 2003 — `http://www.ietf.org/rfc/rfc3526.txt`

[9] *Advanced Encryption Standard*, Federal Information Processing Standards Publication 197, 2001 — `http://www.csrc.nist.gov/publications/fips/fips197/fips-197.pdf`

[34] This can be the first bytes of each `ZID`, the hash of their concatenation or any string derived from the `ZID`s that is long enough to have a low probability to collide with the one of another couple of endpoints.

[10] Morris Dworkin, *Recommendation for Block Cipher: Methods and Techniques*, NIST Special Publication 800-38A 2001 Edition, —— , 2001 — http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf

[11] A. Schwartz, J. Robin, *Analysis of ZRTP*, —— , 2006 — http://www.stanford.edu/class/cs259/projects/project05/

[12] W. Aiello, S.M. Bellovin, M. Blaze, R. Canetti, J. Ioannidis, A.D. Keromytis, O. Reingold, *Just Fast Keying: Key Agreement In A Hostile Internet*, ACM Transactions on Information and System Security (TISSEC), 2004 — http://www.wisdom.weizmann.ac.il/~reingold/publications/jfk-tissec.pdf

AVISPA

[13] *AVISPA v1.1 User Manual*, —— , 2006 — http://www.avispa-project.org/package/user-manual.pdf

[14] *HLPSL Tutorial*, —— , 2006 — http://www.avispa-project.org/package/tutorial.pdf

[15] *The High Level Protocol Specification Language*, —— , 2003 — http://www.avispa-project.org/delivs/2.1/d2-1.pdf

[16] *The Intermediate Format*, —— , 2003 — http://www.avispa-project.org/delivs/2.3/d2-3.pdf

[17] David Basin, Sebastian Mödersheim, Luca Viganò, *OFMC: A symbolic model checker for security protocols*, Springer-Verlag, 2004 — http://www.avispa-project.org/papers/ofmc-jis05.pdf

[18] Mathieu Turuani, *The CL-Atse Protocol Analyser*, Springer, 2006 — http://hal.inria.fr/docs/00/10/35/73/PDF/RTA06_16_Turuani.pdf

ProVerif and π-calculus

[19] Bruno Blanchet, *ProVerif, Automatic Cryptographic Protocol Verifier — User Manual*, —— , 2007 — http://www.di.ens.fr/~blanchet/crypto/proverif-manual.ps.gz

[20] Robin Milner, *Communicating and Mobile Systems: the π-calculus*, Cambridge University Press, 1999

[21] Martín Abadi, Cédric Fournet, *Mobile Values, New Names, and Secure Communication*, Proceedings of the 28th ACM Symposium on Principles of Programming Languages, 2001 — http://www.soe.ucsc.edu/~abadi/Papers/popl01-abadi-fournet.ps

[22] Martín Abadi, Bruno Blanchet, *Analyzing Security Protocols with Secrecy Types and Logic Programs*, Journal of the ACM 52(1), 2005 — http://www.di.ens.fr/~blanchet/publications/AbadiBlanchetJACM7037.pdf

[23] Martín Abadi, Bruno Blanchet, Cédric Fournet, *Just Fast Keying in the Pi Calculus*, ACM Transactions on Information and System Security (TISSEC), 2007 — http://www.di.ens.fr/~blanchet/publications/AbadiBlanchetFournetTISSEC07.pdf

[24] Bruno Blanchet, *Automatic Proof of Strong Secrecy for Security Protocols*, —— , 2004 — http://www.di.ens.fr/~blanchet/publications/BlanchetMPII04.ps.gz

Wikipedia

[25] *ZRTP** — http://en.wikipedia.org/wiki/ZRTP

[26] *Zfone* — http://en.wikipedia.org/wiki/Zfone

[27] *Real-time Transport Protocol* — http://en.wikipedia.org/wiki/Real-time_Transport_Protocol

[28] *Secure Real-time Transport Protocol* — http://en.wikipedia.org/wiki/Secure_Real-time_Transport_Protocol

[29] *Session Initiation Protocol* — http://en.wikipedia.org/wiki/Session_Initiation_Protocol

[30] *SHA hash functions* — http://en.wikipedia.org/wiki/SHA_hash_functions

[31] *HMAC* — http://en.wikipedia.org/wiki/Hmac

[32] *Advanced Encryption Standard* — http://en.wikipedia.org/wiki/Advanced_Encryption_Standard

[33] *Block cipher modes of operation* — http://en.wikipedia.org/wiki/Block_cipher_modes_of_operation

*This article has been modified and expanded by the author.