The ZRTP Protocol
Analysis on the Diffie-Hellman Mode

Riccardo Bresciani

Computer Science Department Technical Report **TCD-CS-2009-13**
Foundations and Methods Research Group **June 12, 2009**

The ZRTP Protocol

Analysis on the Diffie-Hellman Mode

Technical Report TCD-CS-2009-13 [†]

Riccardo Bresciani [*]

June 12, 2009

Abstract ZRTP is a key agreement protocol by Philip Zimmermann, Alan Johnston and Jon Callas, 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 ProVerif, which tests security properties of ZRTP; in order to perform the analysis, the protocol has been modeled in the applied π-calculus.

Contents

1 Introduction

Communications over the internet can take the advantage of using the *Real-time Transport Protocol* (RTP): this protocol has been conceived to effectively transport media streams, such as voice or video, and for this reason it gives great importance to real-time issues.

The issue of security is not a concern of RTP, as it depends just on the data carried in RTP packets. *Secure RTP* (SRTP) is a profile of RTP that deals with security issues such as encryption and message protection in general.

[†]The present work has emanated from research conducted with the financial support of Science Foundation Ireland.
This work has been done using exclusively free software. Images in the present work include modifications of media from the Tango Icon Library (`http://tango.freedesktop.org/Tango_Desktop_Project`).
The author wishes to thank *Phil Zimmermann*, *Andrew Butterfield*, *Bruno Blanchet* and *Steve Kremer* for their helpful comments and suggestions.

[*]Foundations and Methods Group, Trinity College Dublin — `bresciar@cs.tcd.ie`

SRTP needs some key material to provide this, and this key material must be known to the endpoints that want to communicate. A way to accomplish this is to negotiate a fresh key, from which all the necessary key material will be derived, and parameters to establish an SRTP session: a key agreement protocol that can be used is ZRTP [1, 2].

The mechanism underlying this protocol is a Diffie-Hellman exchange. Ephemeral Diffie-Hellamn keys are generated each time a session is established: this allows us to avoid the complexity of creating and maintaining a *Public Key Infrastructure* (PKI).
When having to deal with Diffie-Hellman exchanges, the problem that immediately arises is how to secure the exchange against *Man-in-the-Middle attacks*[1]: ZRTP comes up with different solutions to solve this issue.
The first mechanism uses cached secrets that are established on the very first session between a pair of endpoints and that evolve in time. This can work only if there is no attacker taking part in the first session: to ensure this, the *Short Authentication String* (SAS) method is used — the two endpoints compare a value by reading it aloud and in the case the two values match[2] it is verified that no *Man-in-the-Middle attack* has been performed.

The protocol is strengthened against *denial of service attacks* by means of a sequence of *keyed-Hash Message Authentication Codes* (HMAC) that allow each agent to disregard false messages, injected by an intruder, upon receipt of the next message. This is achieved through the *hash images* `H0`, `H1`, `H2` and `H3` sent throughout the run of the protocol (see figure 2): this mechanism is explained in subsection 2.2.

At the time of this writing the ZRTP protocol is in its pre-RFC state, *i.e.* the protocol draft is frozen and it is being reviewed before becoming a RFC.

2 Protocol Overview

ZRTP has three possible working modes:

Diffie-Hellman mode is based on a Diffie-Hellman exchange: all SRTP keys are computed from the secret value computed by each party;

Multistream mode is usable only if there is already an active SRTP session between the endpoints: new SRTP keys for a new stream can be derived from a the preceding Diffie-Hellman exchange, avoiding the expensive computations of a new one;

Preshared mode does not rely on a Diffie-Hellman exchange, but on previously cached secrets. This is secure as far as the secret cache is not corrupted. Indeed, even in this case, it mantains the perfect forward secrecy of the protocol, as keying material is deleted as soon as each session is terminated.

The present work addresses the Diffie-Hellman mode only, as it is the setting where a *Man-in-the-Middle attack* can be performed[3].

2.1 Call flow

The agents taking part in the protocol play two different roles: one is the *initiator*, who requests the Diffie-Hellman exchange to start, and the other is the *responder*, who accomplishes the initiator's request. In case both endpoints try to act as initiators[4], the protocol has arbitration rules that assign the appropriate role to each agent.

The agents 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 not supporting ZRTP. The *User Datagram Protocol* (UDP) is used to exchange these messages.

[1]A *Man-in-the-Middle attack* takes place when an active attacker is able to relay all the traffic among the agents and he can take advantage of this situation to alter the exchanged messages, in such a way that allows him to break into the communication undetected.

[2]In some *Private Branch eXchange* (PBX) environments there is the scenario when the PBX acts as a trusted *Man-in-the-Middle*: to handle this case ZRTP offers a SAS relaying feature, which will not be discussed in the present work.

[3]If no *Man-in-the-Middle attack* has succeeded in this session, all subsequent sessions in Multistream or Preshared mode that rely on this one are safe, under the assumption that integrity of the secret cache is preserved.

[4]This may happen because of the symmetry of the discovery phase, see subsection 2.2.

The ZRTP frame features a *Cyclic Redundancy Check* (CRC), which is more reliable than the one built-in in UDP (this lowers the probability 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).

2.2 Key Agreement in Diffie-Hellman Mode

The key agreement algorithm can be divided into 4 steps:

1. discovery;
2. hash committment;
3. Diffie-Hellman exchange and key derivation;
4. confirmation.

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

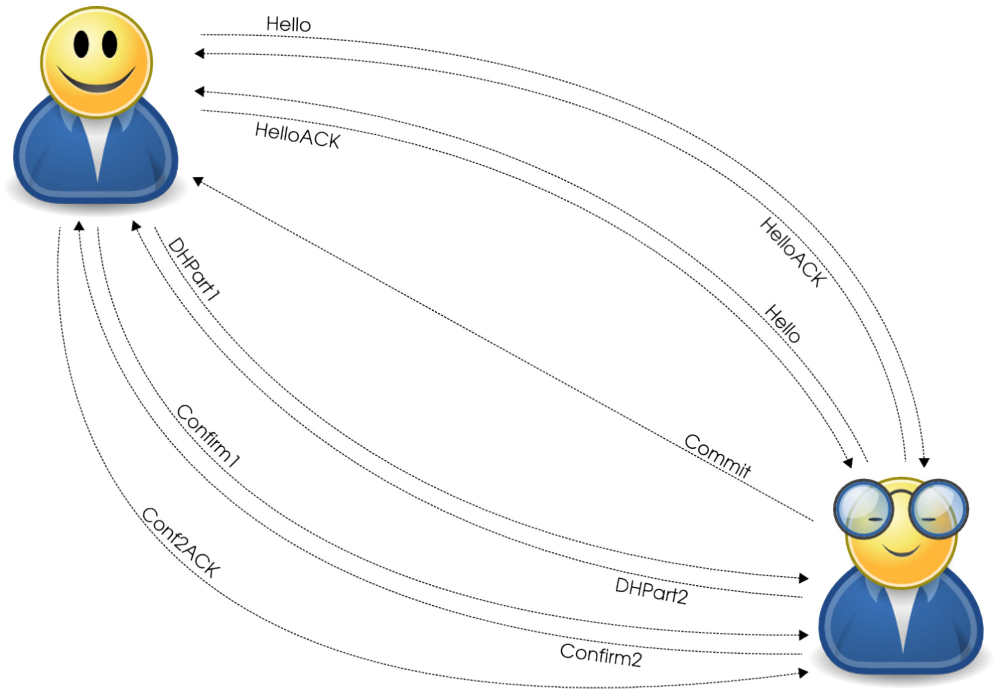

Figure 1: Key agreement call flow

Discovery

During the discovery phase the initiator and the responder exchange their ZRTP identifiers[5]. Besides, they gather information about each other's capabilities, in terms of supported ZRTP versions, hash functions[6], ciphers[7], authorization tag lengths[8], key agreement types[9], and SAS algorithms[10].

The messages exchanged during this phase are called `Hello` messages (variable length). An acknowledgement is sent upon receipt of a `Hello` message: this is the `HelloACK` message[11] (see figure 2).

[5]Each ZRTP instance has a unique 96 bit random identifier, generated only once when the client is set up for the first time.
[6]At the time of this writing the only supported hash function is SHA-256 (`S256`) [3]
[7]The cipher to be used is AES, with 128 bit (`AES1`) or 256 bit (`AES3`) keys [10, 5].
[8]The authentication tag relies on HMAC-SHA1 [4] and can be 32 bit (`HS32`) or 80 bit (`HS80`) long [5].
[9]This chooses the Diffie-Hellman mode, which can use 3072 bit (`DH3k`) or — for slower processors — 2048 bit (`DH2k`) [7, 8]. Elliptic curve Diffie-Hellman exchange is also supported, using curves P-256 (`EC25`), P-384 (`EC38`) or P-521 (`EC52`) [9]. This block may select Preshared mode (`Prsh`) or Multistream mode (`Mult`).
[10]Possible SAS schemes use base 32 (`B32`) or base 256 (`B256`) encoding.
[11]The initiator may skip sending his `HelloACK` message and reply immediately with a `Commit` message.

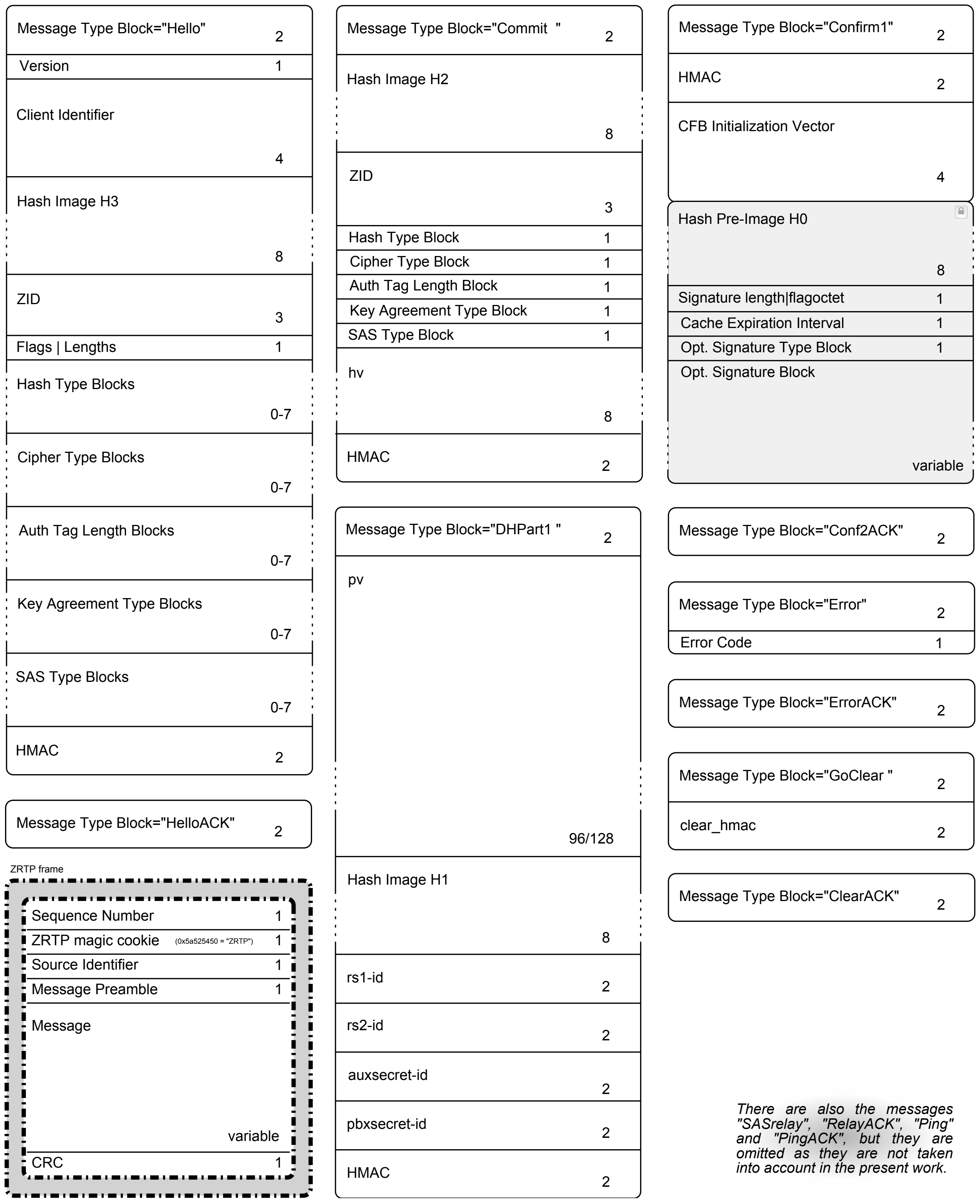
Message Type Block="Hello" 2
Version 1
Client Identifier 4
Hash Image H3 8
ZID 3
Flags | Lengths 1
Hash Type Blocks 0-7
Cipher Type Blocks 0-7
Auth Tag Length Blocks 0-7
Key Agreement Type Blocks 0-7
SAS Type Blocks 0-7
HMAC 2
Message Type Block="HelloACK" 2
ZRTP frame
Sequence Number 1
ZRTP magic cookie (0x5a525450 = "ZRTP") 1
Source Identifier 1
Message Preamble 1
Message variable
CRC 1
Message Type Block="Commit " 2
Hash Image H2 8
ZID 3
Hash Type Block 1
Cipher Type Block 1
Auth Tag Length Block 1
Key Agreement Type Block 1
SAS Type Block 1
hv 8
HMAC 2
Message Type Block="DHPart1 " 2
pv 96/128
Hash Image H1 8
rs1-id 2
rs2-id 2
auxsecret-id 2
pbxsecret-id 2
HMAC 2
Message Type Block="Confirm1" 2
HMAC 2
CFB Initialization Vector 4
Hash Pre-Image H0 8
Signature length|flagoctet 1
Cache Expiration Interval 1
Opt. Signature Type Block 1
Opt. Signature Block variable
Message Type Block="Conf2ACK" 2
Message Type Block="Error" 2
Error Code 1
Message Type Block="ErrorACK" 2
Message Type Block="GoClear " 2
clear_hmac 2
Message Type Block="ClearACK" 2
There are also the messages "SASrelay", "RelayACK", "Ping" and "PingACK", but they are omitted as they are not taken into account in the present work.

Figure 2: ZRTP messages

Hash Commitment

After the discovery phase, the initiator chooses which hash function, cipher, authorization tag length, key agreement type and SAS algorithm should be used, basing his choice on the information brought in the `Hello` messages: this is sent to the responder via the `Commit` message (see figure 2).

The initiator then generates a fresh Diffie-Hellman key pair (secret value `svI` and public value `pvI`, `svI` being twice as long as the AES key length):

$$\texttt{pvI} = g^{\texttt{svI}} \bmod p$$

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

The initiator calculates `hvI` as the hash of his `DHPart2` message (yet to be sent), concatenated with the information of the responder's `Hello` message[12]:

$$\texttt{hvI} = \mathcal{H}(\text{Initiator's } \texttt{DHPart2} | \text{Responder's } \texttt{Hello})$$

The hash commitment prevents the initiator from choosing his keys depending on the responder's choice: being able to do such a thing would imply being able to find a collision in the hashing function, which is assumed to be computationally infeasable.
As the SAS will be function of the exchanged messages, this also implies that neither party is able to influence deterministically the SAS value.

Diffie-Hellman Exchange and Key Derivation

The endpoints aim at generating a new shared secret `s0`, from which later they will derive the new retained shared secret `rs0`, by means of the Diffie-Hellman exchange.

When the responder gets the `Commit` message, he generates his own fresh Diffie-Hellman key pair (`svR` and `pvR`).

$$\texttt{pvR} = g^{\texttt{svR}} \bmod p$$

The responder then looks up in the cache and gets the retained secrets shared with the initiator[13] (`rs1`, `rs2`).
A random value is generated if a secret of a given type does not exist, so an eavesdropper can't get to know the number of secrets shared between the two parties: the random value will generate a mismatch and therefore can be discarded afterwards[14].

Even if the retained shared secrets match, it is also necessary that the order in which these secrets are sorted is the same. This is accomplished by having the responder sorting his secrets as the initiator does: each secret `sX` that will take part in the computation of the new secret `s0` is either a null value (in case of mismatch), either the `X`-th secret, according to the initiator's ordering.
To achieve this goal the responder calculates a sequence of HMACs ($\mathcal{H}_M(k, s)$, where k is the key and s is the string on which the key is applied) using the retained shared secrets as keys:

$$\begin{aligned}
\texttt{rs1-idR} &= \mathcal{H}_M(\texttt{rs1}, \texttt{"Responder"}) \\
\texttt{rs2-idR} &= \mathcal{H}_M(\texttt{rs2}, \texttt{"Responder"}) \\
\texttt{auxsecret-idR} &= \mathcal{H}_M(\texttt{auxsecret}, \texttt{"Responder"}) \\
\texttt{pbxsecret-idR} &= \mathcal{H}_M(\texttt{pbxsecret}, \texttt{"Responder"})
\end{aligned}$$

[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]The `ZID` is used to retrieve information on retained shared secrets (`rs1`, `rs2`). Additional secrets may exist, such as `auxsecret` and `pbxsecret` — these secrets depend on the particular environment where ZRTP is running.

[14]An user should be careful when accepting a session where there is no retained shared secret match (for example the first session): an intruder could successfully mount 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.

The responder sends these values to the initiator in the DHPart1 message (see figure 2). When the initiator gets the message, he checks that $\texttt{pvR} \neq 1$ and $\texttt{pvR} \neq (p-1)$: in this case the exchange is terminated[15].

The initiator performs the same operation:

$$\begin{aligned}\texttt{rs1-idI} &= \mathcal{H}_M(\texttt{rs1}, \texttt{"Initiator"})\\ \texttt{rs2-idI} &= \mathcal{H}_M(\texttt{rs2}, \texttt{"Initiator"})\\ \texttt{auxsecret-idI} &= \mathcal{H}_M(\texttt{auxsecret}, \texttt{"Initiator"})\\ \texttt{pbxsecret-idI} &= \mathcal{H}_M(\texttt{pbxsecret}, \texttt{"Initiator"})\end{aligned}$$

The initiator can now send the DHPart2 message (see figure 2). Upon receipt the responder will perform the usual check on pvB ($\texttt{pvB} \neq 1$, $\texttt{pvB} \neq (p-1)$) and then will check for consistency of the hash commitment hvi: if the check fails, a *Man-in-the-Middle attack* is probably taking place and the exchange is aborted.

Afterwards the initiator calculates which HMACs he should expect from the responder: after comparing them with the ones in the DHPart1 message, he keeps the matching secrets and disregards the others, replacing them by a null value.

The assignment of each shared secret to sX defines the order:

$$\begin{aligned}\texttt{s1} &= \texttt{rs1}\\ \texttt{s2} &= \texttt{auxsecret}\\ \texttt{s3} &= \texttt{pbxsecret}\end{aligned}$$

If there is a mismatch in rs1, then $\texttt{s1} = \texttt{rs2}$; if there is a mismatch also for rs2, that secret is discarded.

After the DHPart2 message has been exchanged, both agents can compute the Diffie-Hellman result (dhr):

$$\begin{aligned}\texttt{dhr} &= \texttt{pvI}^{\texttt{svR}} \bmod p\\ &= (g^{\texttt{svI}} \bmod p)^{\texttt{svR}} \bmod p\\ &= g^{\texttt{svR}\cdot\texttt{svI}} \bmod p\\ &= (g^{\texttt{svR}} \bmod p)^{\texttt{svI}} \bmod p\\ &= \texttt{pvR}^{\texttt{svI}} \bmod p\end{aligned}$$

Now the agents are able to generate a new secret s0 by hashing the concatenation of the Diffie-Hellman result (dhr), the message hash (mh, hash of the concatenation of the responder's Hello message, Commit, DHPart1 and DHPart2) and the shared secrets (s1, s2 and s3)[16]:

$$\begin{aligned}\texttt{mh} =& \mathcal{H}(\text{Responder's } \texttt{Hello}|\texttt{Commit}|\texttt{DHPart1}|\texttt{DHPart2})\\ \texttt{s0} =& \mathcal{H}(\texttt{dhr}|\texttt{mh}|\texttt{s1}|\texttt{s2}|\texttt{s3})\end{aligned}$$

It is now clear why non-matching secrets have been disregarded and replaced by a null value: they have no effect on the concatenation.

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[17] to be established in parallel — the subsequent one can run only after the Conf2ACK message relative to the preceding exchange has been received.

Confirmation

The endpoints can now use s0 to generate a ZRTP session key[18] and SRTP master keys (SRTP-mkey) and salts (SRTP-msalt) — separate in each direction for each media stream — using the key derivation function

[15] An attacker could inject a false DHPart1 message to weaken the final Diffie-Hellman result.

[16] Actually some other parameters are also concatened in this hash, following the requirements listed in NIST SP800-56A: these extra parameters are omitted for the sake of simplicity. The actual s0 should be $\texttt{s0} = \mathcal{H}(\texttt{counter=1}|\texttt{dhr}|\texttt{"ZRTP-HMAC-KDF"}|\texttt{ZIDR}|\texttt{ZIDI}|\texttt{mh}|len(\texttt{s1})|\texttt{s1}|len(\texttt{s2})|\texttt{s2}|len(\texttt{s3})|\texttt{s3})$.

[17] Except when ZRTP runs in Preshared or Multistream mode.

[18] This will be used to generate not only the sasvalue, but also new keys in case of a subsequent exchange in Multistream mode.

$\mathcal{K}$, which is an HMAC function taking the length (where the obtained values should be truncated) and the `KDFContext` (defined as the concatenation of `ZIDI`, `ZIDR` and `mh`) as extra parameters: this provides keys of the length required by the chosen SRTP algorithm[19].
For the sake of simplicity these parameters will always be omitted[20]:

$$\begin{aligned}
\texttt{SRTP-mkeyR} &= \mathcal{K}(\texttt{s0}, \texttt{"Responder SRTP master key"})\\
\texttt{SRTP-msaltR} &= \mathcal{K}(\texttt{s0}, \texttt{"Responder SRTP master salt"})\\
\texttt{SRTP-mkeyI} &= \mathcal{K}(\texttt{s0}, \texttt{"Initiator SRTP master key"})\\
\texttt{SRTP-msaltI} &= \mathcal{K}(\texttt{s0}, \texttt{"Initiator SRTP master salt"})
\end{aligned}$$

Each session has a unique identifier, computed using the function $\mathcal{K}$ and passing a different context to it: instead of the usual `KDFContext`, the value is only the concatenation of `ZIDI` and `ZIDR`[21]. For this reason $\mathcal{K}'$ is used in the expression of `ZRTPSess` instead of $\mathcal{K}$:

$$\texttt{ZRTPSess} = \mathcal{K}'(\texttt{s0}, \texttt{"ZRTP Session Key"})$$

Next they compute their HMAC keys[22] (`hmackey`) and the new retained secret `rs0`:

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

After this, the agents 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 for new concurrent media streams between the same endpoints, within the limit of the call signaling session[23].

$$\begin{aligned}
\texttt{ZRTP-keyR} &= \mathcal{K}(\texttt{s0}, \texttt{"Responder ZRTP Key"})\\
\texttt{ZRTP-keyI} &= \mathcal{K}(\texttt{s0}, \texttt{"Initiator ZRTP Key"})
\end{aligned}$$

After this `s0` is deleted[24], all other key material will be deleted as soon as it is no longer used, in any case no later than the end of the session.

They also compute the SAS value (`sasvalue`):

$$\begin{aligned}
\texttt{sashash} &= \mathcal{K}(\texttt{ZRTPSess}, \texttt{"SAS"})\\
\texttt{sasvalue} &= \text{[Rightmost 32 bits of]}\ \texttt{sashash}
\end{aligned}$$

Finally the agents can send the confirmation messages `Confirm1` and `Confirm2` (see figure 2), which are exchanged essentially for three 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 CFB-encrypted transmission of the SAS Verified flag (`V`), so that no passive observer can learn whether the agents have the good habit of verifying the SAS;
3. they allow the CFB-encrypted transmission of the hash image `H0` (see subsection 2.2).

[19] 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.

[20] As the pourpose of the present work is a formal proof of security, computational aspects are not taken into account: the functions $\mathcal{K}$ and $\mathcal{H}_M$ are substantially the same thing, as the length parameter is irrelevant from a non-computational point of view and the value of `KDFContext` is publicly known.

[21] As `ZRTPSess` is needed to create new keying material in the case of a subsequent run of the protocol in Multistream mode, it must not depend on `mh`. This function is substantially as secure as $\mathcal{K}$ with the usual `KDFContext`, as the concatenation of `ZIDI` and `ZIDR` is also a publicly known value.

[22] These keys are used only by ZRTP, not by SRTP.

[23] In case of separate calls, each call has its own `ZRTP-keys`.

[24] In the protocol draft the authors put great emphasis on the importance of deleting `s0` as soon as it is no longer needed: this prevents the possibility of recreating the keys, and this is important in case something may go wrong in that session, for example an attacker somehow gaining access to that value.

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), the *SAS Verified flag* (V) and the *PBX enrollment* flag (E):

$$\texttt{flagoctet} = \texttt{E} \cdot 2^3 + \texttt{V} \cdot 2^2 + \texttt{A} \cdot 2^1 + \texttt{D} \cdot 2^0$$

The encrypted part of ConfirmX is ciphered via the CFB algorithm [11], 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 (see figure 2) is a confirmation sent by the responder 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.
It must be noted that if one endpoint fails to update the secret cache, there still could be a secret[25] to rely on for a subsequent key agreement.

Hash images

When the key agreement procedure starts, both agents generate an 8-word nonce (H0R and H0I, respectively). From this they will compute a sequence of *hash images*:

$$\begin{aligned} \texttt{H1R} &= \mathcal{H}(\texttt{H0R}) & \qquad \texttt{H1I} &= \mathcal{H}(\texttt{H0I}) \\ \texttt{H2R} &= \mathcal{H}(\texttt{H1R}) & \qquad \texttt{H2I} &= \mathcal{H}(\texttt{H1I}) \\ \texttt{H3R} &= \mathcal{H}(\texttt{H2R}) & \qquad \texttt{H3I} &= \mathcal{H}(\texttt{H2I}) \end{aligned}$$

The Hello, Commit, DHPart1 and DHPart2 messages contain HMACs, which are keyed with these values. This will allow each agent to detect injected false messages upon receipt of the next message:

- the responder's Hello message contains H3R;
- the HMAC of the responder's Hello message is keyed with H2R;
- the initiator's Hello message contains H3I;
- the HMAC of the initiator's Hello message is keyed with H2I;
- the Commit message contains H2I: the responder can check the HMAC of the initiator's Hello message;
- the HMAC of the Commit message is keyed with H1I;
- the DHPart1 message contains H1R: the initiator can derive H3R and check the HMAC of the responder's Hello message;
- the HMAC of the DHPart1 message is keyed with H0R;
- the DHPart2 message contains H1I: the responder can check the HMAC of the Commit message;
- the HMAC of the DHPart2 message is keyed with H0I;
- the encrypted part of Confirm1 message contains H0R: the intiator can check the HMAC of the DHPart1 message;
- the encrypted part of Confirm2 message contains H0I: the responder can check the HMAC of the DHPart1 message.

Moreover each agent checks that the hash images are coherent among themselves.

[25]The non-updated rs1 will match the updated rs2, in case the rs1s were matching in the current key agreement.

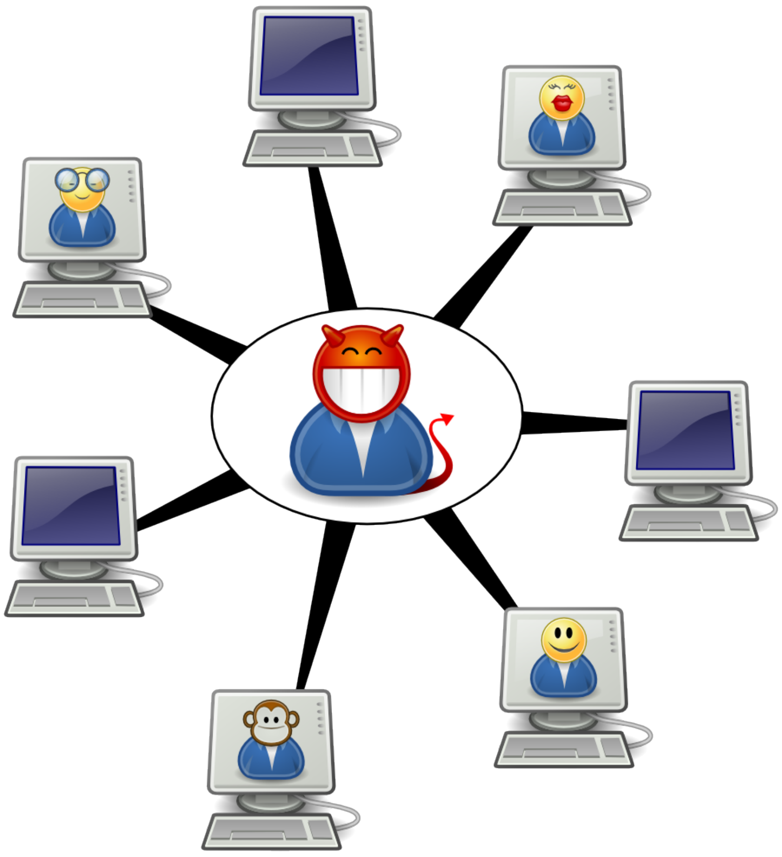

Figure 3: The Dolev-Yao model

2.3 Session Termination

An SRTP session or a ZRTP exchange ends by means of the `GoClear` message (see figure 2) or of an `Error` message (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:

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

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

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-keys` remain till the end of the signaling session.

3 ProVerif Model

3.1 ProVerif and the Dolev-Yao Model

The Dolev-Yao model [13], schematised in figure 3, assumes that:

- the net is under the intruder's control: messages can be intercepted and altered. New messages can be injected to the net;

- the cryptographic primitives are perfect;
- the protocol admits any number or participants and any number of parallel sessions;
- the protocol messages can be of any size.

This is a formal model that can be effectively captured by automatic protocol verifiers and it is much easier to be implemented than computational models (for this reason proofs in the computational model have been done by hand till very recent times): most automatic proofs on protocols have been done in this model.

ProVerif is a cryptographic protocol verifier written by Bruno Blanchet (ENS, Paris) that implements the Dolev-Yao model, providing also support for equational reasoning [14, 15, 16].

By modeling a cryptographic protocol as a sequence of Horn clauses or as a process in the applied π-calculus (the process will be automatically translated into a sequence of Horn clauses — see the following subsection for the syntax of this process calculus), it is possible to test it in this setting.

ProVerif can be used to verify trace and equivalence properties of protocols. Among the trace properties we are interested in testing secrecy properties, *i.e.* whether a Dolev-Yao attacker is able to derive a term from the messages exchanged among the agents: for example this can be used to prove the correctness of a key agreement protocol, by proving that a term, encrypted under the negotiated key and sent on a public channel, is not derivable by an attacker.

ProVerif verify secrecy properties by applying resolution algorithms to the system of Horn clauses that describe the protocol. These clauses are made of predicates that state either that the attacker knows a certain term or that a certain message can be send.
Initially these predicates are combined in a set of Horn clauses that describe the initial knowledge pool of the attacker and the rules that can be used to infer other predicates. The resolution process consist in starting from this rules and verify that an attacker cannot derive any term that we require to be secret.

The applied π-calculus

Martín Abadi and Cédric Fournet have built the applied π-calculus [18] on top of Milner's π-calculus [17]: the main thing is that names are replaced by terms (the atomic values of the π-calculus are not enough to deal efficiently with the complexity of a cryptographic protocol). By using equational theories, it is possible to take fully advantage of this calculus to test security properties of communication protocols.

The syntax of the applied π-calculus is:

M,N:==		Terms
	x,y,z	Variables
	a,b,c,k,s	Names
	$f(M_1,\ldots,M_n)$	Constructor application
P,Q:==		Processes
	$\overline{M}\langle N\rangle.P$	Output
	$M(x).P$	Input
	$let\ x = g(M_1,\ldots,M_n)\ in\ P\ else\ Q$	Destructor application
	$if\ M = N\ then\ P\ else\ Q$	Conditional
	0	Nil process
	$P\|Q$	Parallel composition
	$!P$	Replication
	$(\nu a).P$	Restriction

3.2 The Model

The protocol has been modeled in the following way:

- there is no mismatch in the secrets: the key agreement procedure can rely on this for key generation. This is ideally the typical run of the protocol, when SAS has been verified in the very first session between the agents and the secret cache has been correctly updated in each subsequent session;

- publicly known dummy constants have been used for what does not concern security;
- no negotiation is done during the discovery phase, thus the hash function is predefined and publicly known, as well as the encryption algorithms, ZRTP version and so on.

The goal of ProVerif is to verify that the attacker cannot get to know the secret values exchanged in the confirmation messages: this means that the key agreement procedure cannot be compromised by an attacker in a way that allows him to read the communication between the endpoints, as these values are encrypted under the key on which the endpoints have agreed during the protocol run.

Applied π-calculus Process

$\mathit{Initiator} =$

$\nu\mathtt{SVI}.$	*generate secret value*
$\nu\mathtt{HOI}.$	*generate hash image* `HOI`
$\mathit{let}\ \{\mathtt{H1I} = \mathcal{H}(\mathtt{HOI})\}\ \mathit{in}$	*compute hash image* `H1I`
$\mathit{let}\ \{\mathtt{H2I} = \mathcal{H}(\mathtt{H1I})\}\ \mathit{in}$	*compute hash image* `H2I`
$\mathit{let}\ \{\mathtt{H3I} = \mathcal{H}(\mathtt{H2I})\}\ \mathit{in}$	*compute hash image* `H3I`
$\mathit{hellor}((\mathtt{HELLOSTRINGRI,H3RI,HMACHELLORI})).$	*wait for* `Hello` *message*
$\overline{\mathit{helloackr}}\langle\mathtt{helloack}\rangle.$	*send* `HelloACK` *message*
$\overline{\mathit{helloi}}\langle(\mathtt{hellostringi}, \mathtt{H3I}, \mathcal{H}_M((\mathtt{hellostringi}, \mathtt{H3I}), \mathtt{H2I}))\rangle.$	*send* `Hello` *message*
$\mathit{helloacki}(\mathtt{HELLOACKRI}).$	*wait for* `HelloACK` *message*
$\mathit{let}\ \{\mathtt{PVI} = g^{\mathtt{SVI}}\}\ \mathit{in}$	*compute public value*
$\mathit{let}\ \{\mathtt{SECRETSIDI} = \mathcal{H}_M(\mathtt{initiator}, \mathtt{secrets})\}\ \mathit{in}$	*compute IDs of the secrets*
$\mathit{let}\ \{\mathtt{HVI} = \mathcal{H}((\mathtt{PVI}, \mathtt{SECRETSIDI}, \mathtt{H1I}, \mathtt{HELLOSTRINGRI}, \mathtt{H3RI}))\}\ \mathit{in}$	*compute hash commitment*
$\overline{\mathit{commit}}\langle(\mathtt{commitstring}, \mathtt{HVI}, \mathtt{H2I},$ $\mathcal{H}_M((\mathtt{commitstring}, \mathtt{HVI}, \mathtt{H2I}), \mathtt{H1I}))\rangle.$	*send* `Commit` *message*
$\mathit{dhpart1}((\mathtt{PVRI}, \mathtt{SECRETSIDRI}, \mathtt{H1RI}, \mathtt{HMACDHPART1RI})).$	*wait for* `DHPart1` *message*
$\mathit{if}\ \mathtt{H3RI} = \mathcal{H}(\mathcal{H}(\mathtt{H1RI}))\ \mathit{then}$	*check* `H3R`
$\mathit{if}\ \mathtt{HMACHELLORI} = \mathcal{H}_M((\mathtt{HELLOSTRINGRI}, \mathtt{H3RI}), \mathcal{H}(\mathtt{H1RI}))\ \mathit{then}$	*check* `Hello` *HMAC*
$\mathit{if}\ \mathtt{SECRETSIDRI} = \mathcal{H}_M(\mathtt{responder}, \mathtt{secrets})\ \mathit{then}$	*check responder's IDs of the secrets*
$\overline{\mathit{dhpart2}}\langle(\mathtt{PVI}, \mathtt{SECRETSIDI}, \mathtt{H1I}, \mathcal{H}_M((\mathtt{PVI}, \mathtt{SECRETSIDI}, \mathtt{H1I}), \mathtt{HOI}))\rangle.$	*send* `DHPart2` *message*
$\mathit{confirm1}((\mathtt{CONFIRMRI}, \mathtt{HMACSECRI}, \mathtt{ENCHORI}, \mathtt{ENCSECRI})).$	*wait for* `Confirm1` *message*
$\mathit{let}\ \{\mathtt{MHI} = \mathcal{H}((\mathtt{HELLOSTRINGRI}, \mathtt{H3RI}, \mathtt{commitstring}, \mathtt{HVI}, \mathtt{H2I},$ $\mathtt{PVRI}, \mathtt{SECRETSIDRI}, \mathtt{H1RI}, \mathtt{PVI}, \mathtt{SECRETSIDI}, \mathtt{H1I}))\}\ \mathit{in}$	*compute the message hash*
$\mathit{let}\ \{\mathtt{SOI} = \mathcal{H}((\mathtt{PVRI}^{\mathtt{SVI}}, \mathtt{secrets}, \mathtt{MHI}))\}\ \mathit{in}$	*compute* `SO`
$\mathit{let}\ \{\mathtt{ZRTPKEYI} = \mathcal{K}(\mathtt{SOI}, \mathtt{zrtpi})\}\ \mathit{in}$	*compute ZRTP key*
$\mathit{let}\ \{\mathtt{ZRTPKEYRI} = \mathcal{K}(\mathtt{SOI}, \mathtt{zrtpr})\}\ \mathit{in}$	*compute responder's ZRTP key*
$\mathit{let}\ \{\mathtt{HORI} = \mathcal{D}_{\mathtt{ZRTPKEYRI}}(\mathtt{ENCHORI})\}\ \mathit{in}$	*decrypt* `HOR`
$\mathit{let}\ \{\mathtt{SECRI} = \mathcal{D}_{\mathtt{ZRTPKEYRI}}(\mathtt{ENCSECRI})\}\ \mathit{in}$	*decrypt responder's secret block*
$\mathit{if}\ \mathtt{H1RI} = \mathcal{H}(\mathtt{HORI})\ \mathit{then}$	*check* `H1R`
$\mathit{if}\ \mathtt{HMACDHPART1RI} = \mathcal{H}_M((\mathtt{PVRI}, \mathtt{SECRETSIDRI}, \mathtt{H1RI}), \mathtt{HORI})\ \mathit{then}$	*check* `Confirm2` *HMAC*
$\mathit{if}\ \mathtt{HMACSECRI} = \mathcal{H}_M((\mathtt{HORI}, \mathtt{SECRI}), \mathtt{ZRTPKEYRI})\ \mathit{then}$	*check integrity of encrypted part*
$\overline{\mathit{confirm2}}\langle(\mathtt{confirmi}, \mathcal{H}_M((\mathtt{HOI}, \mathtt{ZRTPKEYI}), \mathtt{ZRTPKEYI}),$ $\mathcal{E}_{\mathtt{ZRTPKEYI}}(\mathtt{HOI}), \mathcal{E}_{\mathtt{ZRTPKEYI}}(\mathtt{SECI}))\rangle.$	*send* `Confirm2` *message*
$\mathit{conf2ack}(\mathtt{CONF}).$	*send* `Conf2ACK` *message*

```
$Responder =$
    $\nu\mathtt{SVR}.$                                                        generate secret value
    $\nu\mathtt{H0R}.$                                                        generate hash image H0R
    $let\ \{\mathtt{H1R} = \mathcal{H}(\mathtt{H0R})\}\ in$                  compute hash image H1R
    $let\ \{\mathtt{H2R} = \mathcal{H}(\mathtt{H1R})\}\ in$                  compute hash image H2R
    $let\ \{\mathtt{H3R} = \mathcal{H}(\mathtt{H2R})\}\ in$                  compute hash image H3R
    $\overline{hellor}\langle(\mathtt{hellostringr}, \mathtt{H3R}, \mathcal{H}_M((\mathtt{hellostringr}, \mathtt{H3R}), \mathtt{H2R}))\rangle.$    send Hello message
    $helloackr(\mathtt{HELLOACKIR}).$                                         wait for HelloACK message
    $helloi((\mathtt{HELLOSTRINGIR,H2IR,HMACHELLOIR})).$                       wait for Hello message
    $\overline{helloacki}\langle\mathtt{helloack}\rangle.$                    send HelloACK message
    $commit((\mathtt{COMMITSTRINGIR}, \mathtt{HVIR}, \mathtt{H2IR}, \mathtt{HMACCOMMITIR})).$    wait for Commit message
    $if\ \mathtt{H3IR} = \mathcal{H}(\mathtt{H2IR})\ then$                   check H3I
    $if\ \mathtt{HMACHELLOIR} = \mathcal{H}_M((\mathtt{HELLOSTRINGIR}, \mathtt{H3IR}), \mathtt{H2IR})\ then$    check Hello HMAC
    $let\ \{\mathtt{PVR} = g^{\mathtt{SVR}}\}\ in$                           compute public value
    $let\ \{\mathtt{SECRETSIDR} = \mathcal{H}_M(\mathtt{responder}, \mathtt{secrets})\}\ in$    compute IDs of the secrets
    $\overline{dhpart1}\langle(\mathtt{PVR}, \mathtt{SECRETSIDR}, \mathtt{H1R}, \mathcal{H}_M((\mathtt{PVR}, \mathtt{SECRETSIDR}, \mathtt{H1R}), \mathtt{H0R}))\rangle.$    send DHPart1 message
    $dhpart2((\mathtt{PVIR}, \mathtt{SECRETSIDIR}, \mathtt{H1IR}, \mathtt{HMACDHPART2IR})).$    wait for DHPart2 message
    $if\ \mathtt{H2IR} = \mathcal{H}(\mathtt{H1IR})\ then$                   check H2I
    $if\ \mathtt{HMACCOMMITIR} = \mathcal{H}_M((\mathtt{COMMITSTRINGIR}, \mathtt{HVIR}, \mathtt{H2IR}), \mathtt{H1IR})\ then$    check Commit HMAC
    $if\ \mathtt{SECRETSIDIR} = \mathcal{H}_M(\mathtt{initiator}, \mathtt{secrets})\ then$    check initiator's IDs of the secrets
    $if\ \mathtt{HVIR} = \mathcal{H}((\mathtt{PVIR}, \mathtt{SECRETSIDIR}, \mathtt{hellostringr}, \mathtt{H3R}))\ then$    check HVI
    $let\ \{\mathtt{MHR} = \mathcal{H}((\mathtt{hellostringr}, \mathtt{H3R}, \mathtt{COMMITSTRINGIR}, \mathtt{HVIR}, \mathtt{H2IR},$    compute the message hash
                  $\mathtt{PVR}, \mathtt{SECRETSIDR}, \mathtt{H1R}, \mathtt{PVIR}, \mathtt{SECRETSIDIR}, \mathtt{H1IR}))\}\ in$
    $let\ \{\mathtt{S0R} = \mathcal{H}((\mathtt{PVIR}^{\mathtt{SVR}}, \mathtt{secrets}, \mathtt{MHR}))\}\ in$    compute S0
    $let\ \{\mathtt{ZRTPKEYR} = \mathcal{K}(\mathtt{S0R}, \mathtt{zrtpr})\}\ in$    compute ZRTP key
    $\overline{confirm1}\langle(\mathtt{confirmr}, \mathcal{H}_M((\mathtt{H0R}, \mathtt{ZRTPKEYR}), \mathtt{ZRTPKEYR}),$    send Confirm1 message
                  $\mathcal{E}_{\mathtt{ZRTPKEYR}}(\mathtt{H0R}), \mathcal{E}_{\mathtt{ZRTPKEYR}}(\mathtt{SECR}))\rangle.$
    $confirm2((\mathtt{CONFIRMIR}, \mathtt{HMACSECIR}, \mathtt{ENCH0IR}, \mathtt{ENCSECIR})).$    wait for Confirm2 message
    $let\ \{\mathtt{ZRTPKEYIR} = \mathcal{K}(\mathtt{S0R}, \mathtt{zrtpi})\}\ in$    compute initiator's ZRTP key
    $let\ \{\mathtt{H0IR} = \mathcal{D}_{\mathtt{ZRTPKEYIR}}(\mathtt{ENCH0IR})\}\ in$    decrypt H0I
    $let\ \{\mathtt{SECIR} = \mathcal{D}_{\mathtt{ZRTPKEYIR}}(\mathtt{ENCSECIR})\}\ in$    decrypt initiator's secret block
    $if\ \mathtt{H1IR} = \mathcal{H}(\mathtt{H0IR})\ then$                   check H1I
    $if\ \mathtt{HMACDHPART2IR} = \mathcal{H}_M((\mathtt{PVIR}, \mathtt{SECRETSIDIR}, \mathtt{H1IR}), \mathtt{H0IR})\ then$    check Confirm2 HMAC
    $if\ \mathtt{HMACSECIR} = \mathcal{H}_M((\mathtt{H0IR}, \mathtt{SECIR}), \mathtt{ZRTPKEYIR})\ then$    check integrity of encrypted part
    $\overline{conf2ack}\langle\mathtt{conf}\rangle.$                         send Conf2ACK message
```

See appendix A for the actual ProVerif code.

3.3 Analysis

Starting from the process above, ProVerif generates a sequence of Horn clauses, *i.e.* the intial knowledge pool of the attacker and the rules he can use to derive terms that do not belong to it.

We declare two terms, `SECI` and `SECR` as secret (not known to the attacker at the start) as follows:

```
private free SECR,SECI.
```

We then instruct ProVerif to see if the attacker can get knowledge of these secrets with the line:

```
query attacker: SECI; attacker: SECR.
```

By doing so, we challenge the adversary to derive the terms that are sent encrypted under the negotiated key: if there is no way that an adversary can derive them by applying the rules, then the protocol is safe.

Results

ProVerif shows the protocol to be secure in a Dolev-Yao network, as the attacker cannot derive the terms that were sent under encryption (see the output of the analysis below): if the key agreement procedure can

be performed, then we have the *formal proof* that an attacker cannot have compromised it and have broken into the session.
Here is the output of the analysis:

```
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_76,y_77),y_77) => x_76
Process:

[ ... ]


-- Secrecy & events.
Starting rules:

[ ... ]


Completing...
nounif attacker:hash(H2IR_1492)/-5000
200 rules inserted. The rule base contains 133 rules. 12 rules in the queue.
400 rules inserted. The rule base contains 187 rules. 52 rules in the queue.
600 rules inserted. The rule base contains 198 rules. 68 rules in the queue.
800 rules inserted. The rule base contains 317 rules. 33 rules in the queue.
Starting query not attacker:SECI[]
RESULT not attacker:SECI[] is true.
Starting query not attacker:SECR[]
RESULT not attacker:SECR[] is true.
```

4 Final remarks

In the present work the protocol run has been modeled as two concurrent processes that interact by exchanging messages, synchronizing on every message exchange.

The model does not bother with all the negotiation procedure of the discovery phase, as this is unessential to prove the security of the protocol: according to the Dolev-Yao model (see section 3.1), the cryptographic functions are idealized, so every algorithm is just as strong as any other; moreover the chosen algorithms are publicly known, as they are sent in clear in the `Commit` message.

The analysis performed on the protocol has *formally proven* that ZRTP is a safe key agreement protocol: two endpoints that use it to agree on a key can be sure that their communications are secured against any attack.

For this to happen it is crucial that there are some pre-shared secrets: if this is not the case, ProVerif shows that a *Man-in-the-Middle attack* is possible.
This is the reason why one needs to use SAS to ensure that this attack has not been performed on the first session between the two agents: in this session a reliable shared secret will be created, and therefore all the subsequent sessions will be secured.
It must be noted that this is true under the assumption that SAS provides an effective way to detect the presence of an attacker. [2]

More in general, the present work highlights the benefits of using the applied π-calculus and ProVerif to reason about cryptographic protocols: the model of the protocol accounts for all the peculiarities of a typical run of the ZRTP protocol and therefore provides a good support for reasoning about ZRTP, in view of future modifications and improvements.

A ProVerif Code (Applied π-calculus Format)

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

(***** Diffie-Hellman exponentials ***************************************)
(*                                                                      *)
(* The following lines declare the constant g and the function exp.    *)
(* The function exp is provided with an equation theory that accounts   *)
(* for its properties.                                                  *)
(*                                                                      *)
(************************************************************************)
data g/0.
fun exp/2.
equation exp(exp(g,y),z)=exp(exp(g,z),y).

(***** Hash, HMAC and key derivation functions **************************)
(*                                                                      *)
(* The following lines declare one-way functions: once they are applied  *)
(* it will not be possible to invert them and derive their arguments,   *)
(* in fact no destructor is given for these functions.                  *)
(*                                                                      *)
(************************************************************************)
fun hash/1.
fun hmac/2.
fun kdf/2.

(***** Encryption and Decryption ****************************************)
(*                                                                      *)
(* The following lines declare the function encrypt and the relative    *)
(* destructor decrypt to model the encryption and decryption operations. *)
(*                                                                      *)
(************************************************************************)
fun encrypt/2.
reduc decrypt(encrypt(x,y),y) = x.

(***** Channels *********************************************************)
(*                                                                      *)
(* The following lines declare the public channels                      *)
(*                                                                      *)
(************************************************************************)
free hellor.
free helloackr.
free helloi.
free helloacki.
free commit.
free dhpart1.
free dhpart2.
free confirm1.
free confirm2.
free conf2ack.
```

```
(* Constants *)
data conf/0.
data confirmi/0.
data confirmr/0.
data hellostringr/0.
data hellostringi/0.
data helloack/0.
data commitstring/0.
data responder/0.
data initiator/0.
data zrtpr/0.
data zrtpi/0.

(***** Secret blocks for Confirm messages ********************************)
(*                                                                       *)
(* The following line declares two free names that are not known by the *)
(* attacker: they will be sent encrypted in the Confirm messages.        *)
(* The challenge for the attacker is to derive those terms from the      *)
(* exchanged messages.                                                   *)
(*                                                                       *)
(*************************************************************************)
private free SECR,SECI.
query attacker: SECI; attacker: SECR.


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

let Initiator=
        new SVI;
        new H0I;
        let H1I=hash(H0I) in
        let H2I=hash(H1I) in
        let H3I=hash(H2I) in
        in(hellor,(HELLOSTRINGRI,H3RI,HMACHELLORI));
        out(helloackr,helloack);
        out(helloi,(hellostringi,H3I,hmac((hellostringi,H3I),H2I)));
        in(helloacki,HELLOACKRI);
        let PVI=exp(g,SVI) in
        let SECRETSIDI=hmac(initiator,secrets) in
        let HVI=hash((PVI,SECRETSIDI,H1I,HELLOSTRINGRI,H3RI)) in
        out(commit,(commitstring,HVI,H2I,hmac((commitstring,HVI,H2I),H1I)));
        in(dhpart1,(PVRI,SECRETSIDRI,H1RI,HMACDHPART1RI));
        if H3RI=hash(hash(H1RI)) then (
        if HMACHELLORI=hmac((HELLOSTRINGRI,H3RI),hash(H1RI)) then (
        if SECRETSIDRI=hmac(responder,secrets) then (
          out(dhpart2,(PVI,SECRETSIDI,H1I,hmac((PVI,SECRETSIDI,H1I),H0I)));
          in(confirm1,(CONFIRMRI,HMACSECRI,ENCH0RI,ENCSECRI));
          let MHI=hash((HELLOSTRINGRI,H3RI,commitstring,HVI,H2I,PVRI,
                SECRETSIDRI,H1RI,PVI,SECRETSIDI,H1I)) in
          let S0I=hash((exp(PVRI,SVI),secrets,MHI)) in
          let ZRTPKEYI=kdf(S0I,zrtpi) in
          let ZRTPKEYRI=kdf(S0I,zrtpr) in
          let H0RI=decrypt(ENCH0RI,ZRTPKEYRI) in
          let SECRI=decrypt(ENCSECRI,ZRTPKEYRI) in
          if H1RI=hash(H0RI) then (
          if HMACDHPART1RI=hmac((PVRI,SECRETSIDRI,H1RI),H0RI) then (
          if hmac((H0RI,SECRI),ZRTPKEYRI)=HMACSECRI then
          (
            out(confirm2,(confirmi,hmac((H0I,SECI),ZRTPKEYI),
                        encrypt(H0I,ZRTPKEYI),encrypt(SECI,ZRTPKEYI)));
            in(conf2ack,CONF)
          )))
        ))).

let Responder=
        new SVR;
        new H0R;
        let H1R=hash(H0R) in
        let H2R=hash(H1R) in
        let H3R=hash(H2R) in
        out(hellor,(hellostringr,H3R,hmac((hellostringr,H3R),H2R)));
        in(helloackr,HELLOACKIR);
        in(helloi,(HELLOSTRINGIR,H3IR,HMACHELLOIR));
        out(helloacki,helloack);
        in(commit,(COMMITSTRINGIR,HVIR,H2IR,HMACCOMMITIR));
        if H3IR=hash(H2IR) then (
        if HMACHELLOIR=hmac((HELLOSTRINGIR,H3IR),H2IR) then
        (
          let PVR=exp(g,SVR) in
          let SECRETSIDR=hmac(responder,secrets) in
          out(dhpart1,(PVR,SECRETSIDR,H1R,hmac((PVR,SECRETSIDR,H1R),H0R)));
          in(dhpart2,(PVIR,SECRETSIDIR,H1IR,HMACDHPART2IR));
          if H2IR=hash(H1IR) then (
          if HMACCOMMITIR=hmac((COMMITSTRINGIR,HVIR,H2IR),H1IR) then (
          if SECRETSIDIR=hmac(initiator,secrets) then (
          if HVIR=hash((PVIR,SECRETSIDIR,H1IR,hellostringr,H3R)) then
          (
                let MHR=hash((hellostringr,H3R,COMMITSTRINGIR,HVIR,H2IR,PVR,
                        SECRETSIDR,H1R,PVIR,SECRETSIDIR,H1IR)) in
                let S0R=hash((exp(PVIR,SVR),secrets,MHR)) in
                let ZRTPKEYR=kdf(S0R,zrtpr) in
                out(confirm1,(confirmr,hmac((H0R,SECR),ZRTPKEYR),
                        encrypt(H0R,ZRTPKEYR),encrypt(SECR,ZRTPKEYR)));
                in(confirm2,(CONFIRMIR,HMACSECIR,ENCH0IR,ENCSECIR));
                let ZRTPKEYIR=kdf(S0R,zrtpi) in
                let H0IR=decrypt(ENCH0IR,ZRTPKEYIR) in
                let SECIR=decrypt(ENCSECIR,ZRTPKEYIR) in
                if H1IR=hash(H0IR) then (
                if HMACDHPART2IR=hmac((PVIR,SECRETSIDIR,H1IR),H0IR) then (
                if hmac((H0IR,SECIR),ZRTPKEYIR)=HMACSECIR then
                (
                  out(conf2ack,conf)
                )))
          ))))
        )).


(* Main *)
process
        !(new secrets;              (* generate the shared secrets *)
        ((!Responder)|(!Initiator)))
```

References

[1] P. Zimmermann, A. Johnston, J. Callas, *ZRTP: Media Path Key Agreement for Secure RTP*, —— , 2009 — http://www.ietf.org/internet-drafts/draft-zimmermann-avt-zrtp-15.txt

[2] Riccardo Bresciani, *The ZRTP Protocol – Security Considerations*, —— , 2007 — http://www.lsv.ens-cachan.fr/Publis/RAPPORTS_LSV/PDF/rr-lsv-2007-20.pdf

[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] D. Fu, J. Solinas, *ECP Groups for IKE and IKEv2*, —— , 2007 — http://www.ietf.org/rfc/rfc4753.txt

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

[11] 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

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

[13] Danny Dolev, Andrew C. Yao, *On the security of public-key protocols*, IEEE Transaction on Information Theory 2(29), 1983

[14] Bruno Blanchet, *ProVerif, Automatic Cryptographic Protocol Verifier — User Manual*, —— , 2009 — http://www.proverif.ens.fr/proverif-manual.pdf

[15] Bruno Blanchet, *Vérification automatique de protocoles cryptographiques: modèle formel et modèle calculatoire*, Mémoire d'habilitation à diriger des recherches, Université Paris-Dauphine, 2008 — http://www.di.ens.fr/~blanchet/publications/BlanchetHDR.pdf

[16] Bruno Blanchet, *An Efficient Cryptographic Protocol Verifier Based on Prolog Rules*, 14th IEEE Computer Security Foundations Workshop (CSFW-14), IEEE Computer Society, 2001 — http://www.di.ens.fr/~blanchet/publications/BlanchetCSFW01.ps.gz

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

[18] 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

[19] 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

[20] 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

Addendum: Some Security Considerations from the Authors of ZRTP

What follows is Section 15 of [1].

This document is all about securely keying SRTP sessions. As such, security is discussed in every section. Most secure phones rely on a Diffie-Hellman exchange to agree on a common session key. But since DH is susceptible to a man-in-the- middle (MiTM) attack, it is common practice to provide a way to authenticate the DH exchange. In some military systems, this is done by depending on digital signatures backed by a centrally-managed PKI. A decade of industry experience has shown that deploying centrally managed PKIs can be a painful and often futile experience. PKIs are just too messy, and require too much activation energy to get them started. Setting up a PKI requires somebody to run it, which is not practical for an equipment provider. A service provider like a carrier might venture down this path, but even then you have to deal with cross-carrier authentication, certificate revocation lists, and other complexities. It is much simpler to avoid PKIs altogether, especially when developing secure commercial products. It is therefore more common for commercial secure phones in the PSTN world to augment the DH exchange with a Short Authentication String (SAS) combined with a hash commitment at the start of the key exchange, to shorten the length of SAS material that must be read aloud. No PKI is required for this approach to authenticating the DH exchange. The AT&T TSD 3600, Eric Blossom's COMSEC secure phones [comsec], PGPfone [pgpfone], and CryptoPhone [cryptophone] are all examples of products that took this simpler lightweight approach.

The main problem with this approach is inattentive users who may not execute the voice authentication procedure, or unattended secure phone calls to answering machines that cannot execute it.

Additionally, some people worry about voice spoofing. But it is a mistake to think this is simply an exercise in voice impersonation (perhaps this could be called the "Rich Little" attack). Although there are digital signal processing techniques for changing a person's voice, that does not mean a man-in-the-middle attacker can safely break into a phone conversation and inject his own short authentication string (SAS) at just the right moment. He doesn't know exactly when or in what manner the users will choose to read aloud the SAS, or in what context they will bring it up or say it, or even which of the two speakers will say it, or if indeed they both will say it. In addition, some methods of rendering the SAS involve using a list of words such as the PGP word list [Juola2], in a manner analogous to how pilots use the NATO phonetic alphabet to convey information. This can make it even more complicated for the attacker, because these words can be worked into the conversation in unpredictable ways. Remember that the attacker places a very high value on not being detected, and if he makes a mistake, he doesn't get to do it over. Some people have raised the question that even if the attacker lacks voice impersonation capabilities, it may be unsafe for people who don't know each other's voices to depend on the SAS procedure. This is not as much of a problem as it seems, because it isn't necessary that they recognize each other by their voice, it is only necessary that they detect that the voice used for the SAS procedure matches the voice in the rest of the phone conversation.

A popular and field-proven approach is used by SSH (Secure Shell) [RFC4251], which Peter Gutmann likes to call the "baby duck" security model. SSH establishes a relationship by exchanging public keys in the initial session, when we assume no attacker is present, and this makes it possible to authenticate all subsequent sessions. A successful MiTM attacker has to have been present in all sessions all the way back to the first one, which is assumed to be difficult for the attacker. ZRTP's key continuity features are actually better than SSH, at least for VoIP, for reasons described in Section 15.1. All this is accomplished without resorting to a centrally-managed PKI.

We use an analogous baby duck security model to authenticate the DH exchange in ZRTP. We don't need to exchange persistent public keys, we can simply cache a shared secret and re-use it to authenticate a long series of DH exchanges for secure phone calls over a long period of time. If we read aloud just one SAS, and then cache a shared secret for later calls to use for authentication, no new voice authentication rituals need to be executed. We just have to remember we did one already.

If one party ever loses this cached shared secret, it is no longer available for authentication of DH exchanges. This cache mismatch situation is easy to detect by the party that still has a surviving shared secret cache entry. If it fails to match, either there is a MiTM attack or one side has lost their shared secret cache entry. The user agent that discovers the cache mismatch must alert the user that a cache mismatch has been detected, and that he must do a verbal comparison of the SAS to distinguish if the mismatch is because of a MiTM attack or because of the other party losing her cache. From that point on, the two parties start over with a new cached shared secret. Then they can go back to omitting the voice authentication on later calls.

A particularly compelling reason why this approach is attractive is that SAS is easiest to implement when a graphical user interface or some sort of display is available, which raises the question of what to do when a display is less conveniently available. For example, some devices that implement ZRTP might have a graphical user interface that is only visible through a web browser, such as a PBX or some other nearby device that implements ZRTP as a "bump-in-the- wire". If we take an approach that greatly reduces the need for a SAS in each and every call, we can operate in products without a graphical user interface with greater ease. Then the SAS can be compared less frequently through a web browser, or it might even be presented as needed to the local user through a locally generated voice prompt, which the local user hears and verbally repeats and compares with the remote party. Using a voice prompt in this way is purely for the local ZRTP user agent to render the SAS to the local user, and is not to be confused with the verbal comparison of the SAS between two human users.

It is a good idea to force your opponent to have to solve multiple problems in order to mount a successful attack. Some examples of widely differing problems we might like to present him with are: Stealing a shared secret from one of the parties, being present on the very first session and every subsequent session to carry out an active MiTM attack, and solving the discrete log problem. We want to force the opponent to solve more than one of these problems to succeed.

ZRTP can use different kinds of shared secrets. Each type of shared secret is determined by a different method. All of the shared secrets are hashed together to form a session key to encrypt the call. An attacker must defeat all of the methods in order to determine the session key.

First, there is the shared secret determined entirely by a Diffie- Hellman key agreement. It changes with every call, based on random numbers. An attacker may attempt a classic DH MiTM attack on this secret, but we can protect against this by displaying and reading aloud an SAS, combined with adding a hash commitment at the beginning of the DH exchange.

Second, there is an evolving shared secret, or ongoing shared secret that is automatically changed and refreshed and cached with every new session. We will call this the cached shared secret, or sometimes the retained shared secret. Each new image of this ongoing secret is a non-invertable function of its previous value and the new secret derived by the new DH agreement. It is possible that no cached shared secret is available, because there were no previous sessions to inherit this value from, or because one side loses its cache.

There are other approaches for key agreement for SRTP that compute a shared secret using information in the signaling. For example, [RFC4567] describes how to carry a MIKEY (Multimedia Internet KEYing) [RFC3830] payload in SDP [RFC4566]. Or RFC 4568 (SDES) [RFC4568] describes directly carrying SRTP keying and configuration information in SDP. ZRTP does not rely on the signaling to compute a shared secret, but if a client does produce a shared secret via the signaling, and makes it available to the ZRTP protocol, ZRTP can make use of this shared secret to augment the list of shared secrets that will be hashed together to form a session key. This way, any security weaknesses that might compromise the shared secret contributed by the signaling will not harm the final resulting session key.

The shared secret provided by the signaling (if available), the shared secret computed by DH, and the cached shared secret are all hashed together to compute the session key for a call. If the cached shared secret is not available, it is omitted from the hash computation. If the signaling provides no shared secret, it is also omitted from the hash computation.

No DH MiTM attack can succeed if the ongoing shared secret is available to the two parties, but not to the attacker. This is because the attacker cannot compute a common session key with either party without knowing the cached secret component, even if he correctly executes a classic DH MiTM attack.

The key continuity features of ZRTP are analogous to those provided by SSH (Secure Shell) [RFC4251], but they differ in one respect. SSH caches public signature keys that never change, and uses a permanent private signature key that must be guarded from disclosure. If someone steals your SSH private signature key, they can impersonate you in all future sessions and mount a successful MiTM attack any time they want.

ZRTP caches symmetric key material used to compute secret session keys, and these values change with each session. If someone steals your ZRTP shared secret cache, they only get one chance to mount a MiTM attack, in the very next session. If they miss that chance, the retained shared secret is refreshed with a new value, and the window of vulnerability heals itself, which means they are locked out of any future opportunities to mount a MiTM attack. This gives ZRTP a "self-healing" feature if any cached key material is compromised.

A MiTM attacker must always be in the media path. This presents a significant operational burden for the attacker in many VoIP usage scenarios, because being in the media path for every call is often harder than being in the signaling path. This will likely create coverage gaps in the attacker's opportunities to mount a MiTM attack. ZRTP's self-healing key continuity features are better than SSH at exploiting any temporary gaps in MiTM attack coverage. Thus, ZRTP quickly recovers from any disclosure of cached key material.

The infamous Debian OpenSSL weak key vulnerability [dsa-1571] (discovered and patched in May 2008) offers a real-world example of why ZRTP's self-healing scheme is a good way to do key continuity. The Debian bug resulted in the production of a lot of weak SSH (and TLS/SSL) keys, which continued to compromise security even after the bug had been patched. In contrast, ZRTP's key continuity scheme adds new entropy to the cached key material with every call, so old deficiencies in entropy are washed away with each new session.

It should be noted that the addition of shared secret entropy from previous sessions can extend the strength of the new session key to AES-256 levels, even if the new session uses Diffie-Hellman keys no larger than DH-3072 or ECDH-256, provided the cached shared secrets were initially established when the wiretapper was not present. This is why AES-256 MAY be used with the smaller DH key sizes in Section 5.1.5, despite the key strength comparisons in Table 2 of [SP800-57-Part1].

Caching shared symmetric key material is also less CPU intensive compared with using digital signatures, which may be important for low-power mobile platforms.

References for this section

[RFC4566] Handley, M., Jacobson, V., and C. Perkins, "SDP: Session Description Protocol", RFC 4566, July 2006.

[SP800-57-Part1] Barker, E., Barker, W., Burr, W., Polk, W., and M. Smid, "Recommendation for Key Management - Part 1: General (Revised)", NIST Special Publication 800-57 - Part 1 Revised March 2007.

[Juola2] Juola, P., "Isolated Word Confusion Metrics and the PGPfone Alphabet", Proceedings of New Methods in Language Processing 1996.

[pgpfone] Zimmermann, P., "PGPfone", http://philzimmermann.com/docs/pgpfone10b7.pdf

[comsec] Blossom, E., "The VP1 Protocol for Voice Privacy Devices Version 1.2", http://www.comsec.com/vp1-protocol.pdf

[cryptophone] "CryptoPhone", http://www.cryptophone.de/

[dsa-1571] "Debian Security Advisory - OpenSSL predictable random number generator", http://www.debian.org/security/2008/dsa-1571

[RFC4251] Ylonen, T. and C. Lonvick, "The Secure Shell (SSH) Protocol Architecture", RFC 4251, January 2006.

[RFC4568] Andreasen, F., Baugher, M., and D. Wing, "Session Description Protocol (SDP) Security Descriptions for Media Streams", RFC 4568, July 2006.

[RFC4567] Arkko, J., Lindholm, F., Naslund, M., Norrman, K., and E. Carrara, "Key Management Extensions for Session Description Protocol (SDP) and Real Time Streaming Protocol (RTSP)", RFC 4567, July 2006.

[RFC3830] Arkko, J., Carrara, E., Lindholm, F., Naslund, M., and K. Norrman, "MIKEY: Multimedia Internet KEYing", RFC 3830, August 2004.