Ephemeral Key Exchange for Peer-to-Peer Voice and Video Communication in Realtime via Real-time Transport Protocol (RTP)

V1.7

G. Brands[1], C.B. Roellgen[2]

2017-11-20

Abstract

The Real-time Transport Protocol (RTP) (RFC 3550) is used extensively in Voice-over Internet Protocol (VoIP) telephony, and video teleconference applications, as well as for web-based television services. Data encryption is not part of the protocol. The feature is instead provided by the Secure Real-time Transport Protocol (SRTP), which has severe shortcomings. Keys for the symmetric cipher are exchanged in the SDP part of SIP INVITE messages in the clear or preferably via TLS, which inevitably causes the SIP server to have detailed knowledge of all session keys. It might even happen that one leg of a telephony session is encrypted, while another is not. This tends to happen when the call is routed via a media gateway to a plain old analog telephone.

A rather simple protocol and key negotiation methology is presented in this document that allows to secure the entire route between two VoIP telephones.

Key words: RTP, Real-time Transport Protocol, protocol, codec, G.711, G.722, opus, Peer-to-peer, P2P, RVB, Chebyshev, polynomial, permutable, real numbers, Diffie-Hellman, key, exchange, encryption, quantum, computer, hash, cipher, SIP, Session Initiation Protocol.

1. RTP and the lack of Peer-to-Peer Data Encryption

As a matter of fact, data encryption plays a minor role in today's voice, video, electronic mail and chat communication. RTP (RFC 3550) [4] does not provide encryption. In order to address this deficiency, the SRTP protocol (Secure RTP, RFC 3711) [5] was developed by Cisco and Ericsson in 2004. SRTP clearly is a compromise between basic security requirements of users and the requirements of state authorities, who reserve the right to tap millions of phone calls simultaneously. Never before in history it has been that easy to supervise the entire population of countries, which can ultimately lead to total control.

In order to exchange keying material between the telephony server and endpoints for SRTP, this data is transported in the SDP part of SIP INVITE messages. Users cannot be sure that SIP messages are encrypted or not. Typically a TLS connection is although established between client and server. Voice data is deciphered by the server and either forwarded in the clear to an endpoint – which can be a plain old analog telephone - or it is re-encrypted. In any case, control data as well as the entire payload information is available at the SIP server (or the media gateway) in the clear.

Users might misleadingly think that their telephony sessions are perfectly secure, which is only the case if users control the telephony server as is likely for governments or big corporations.

While SRTP-enabled voice and video communication software and hardware can be exported freely, Peer-to-Peer

1 Gilbert Brands, D-26736 Krummhörn, e-mail: gilbert(at)gilbertbrands.de

2 Bernd Röllgen, D-35576 Wetzlar, e-mail: roellgen(at)globaliptel.com

encryption can not as it renders tapping of telephone conversations rather difficult for the authorities. Tapping of a multitude of telephone conversions is even rendered impossible.

2. Providing Peer-to-Peer Data Encryption and Authentication

The solution is a key agreement protocol which is executed during call setup in-band in an RTP media stream – typically the audio stream. Any Diffie-Hellman type of key exchange is suitable, although classic DH, as well as ECDH run slowly and the algorithms are not quantum-computer-proof.

The video stream, if present, can easily be encrypted by sharing the keying material among the streams.

The proposed protocol does not rely on SIP signaling for the keying material and no infrastructure is required other than the existing one. Clients that are capable of P2P encryption auto-sense if the remote VoIP client supports the protocol. If it does not, it is up to the application software to decide whether to continue the communication in the clear or to hang up.

Diffie–Hellman-type key exchanges by themselves do not provide protection against a man-in-the-middle attack. Authentication is provided via a very reliable and proven side channel: The Short Authentication String (SAS) method. The SAS value, which is a hash of the negotiated keys, is displayed to both endpoints. In order to authenticate each other, this SAS value is checked by the protocol and it is mandatory to read aloud to the respective peer user over the audio channel. If the values on both ends do indeed match, a man-in-middle (MitM) attack is highly unlikely.

2. The RVB Algorithm

The RVB Algorithm [1][2][3] is comparably new. It was originally conceived as quantum computer-resistant replacement for the Diffie-Hellman key exchange and its elliptic curve variants. RVB although belongs in principle to the class of Diffie-Hellman algorithms. It is thus possible to use it for quantum-computer resistant Electronic Identities with a secret key and a public key, as well as to use it for negotiating a shared secret key between two computers.

2.1 Modus operandi of the RVB public key exchange

The RVB algorithm [1][2][3] uses Chebychev polynomials (abbreviated T-polynomials) of the first order in the interval of real numbers $[-1,1]$, which can be composed commutatively. T-polynomials are solutions of certain differential equations and are recursively defined as:

$$T_0(x)=1 \;\;,\;\; T_1(x)=x$$

$$T_n(x)=2*x*T_{n-1}(x)-T_{n-2}(x)=2*T_1(x)*T_{n-1}(x)-T_{n-2}(x)$$

Besides the recursive form, mathematics further provides in the interval $[-1,1]$ the following analytical definition:

$$T_n(x)=\cos(n*\arccos(x))$$

In the interval $[-1,1]$ T-polynomials come with the advantageous property that is commonly exploited using natural numbers: they form an abelian semigroup:

$$T_a(T_b(x))=T_{a*b}(x)=T_b(T_a(x))$$

The experienced reader will immediately notice that this corresponds exactly to the Diffie-Hellman method. This way we've already got all that is needed to negotiate a shared secret key between two peers Alice and Bob. The two peers agree on a publicly known real number x . The two peers subsequently select two integer numbers random a, b , exchange the two values $T_a(x)$ and $T_b(x)$ and finally compute a shared secret from the values exchanged publically.

T-polynomials can be computed very efficiently with complexity $O(\log(n))$ using matrix multiplication:

$$\begin{pmatrix} T_n(x) \\ T_{n+1}(x) \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ -1 & 2\cdot x \end{pmatrix}^n \cdot \begin{pmatrix} T_0(x) \\ T_1(x) \end{pmatrix}$$

This enables the parameters x, n to contain the same amount of digits and thus to be on an equal footing with each other. Only this allows T-polynomials for use in interactive certificate applications. The proposed protocol thus provides the possibility for extension above of use for Diffie-Hellman type key exchanges.

2.2 Modus operandi of an interactive certificate based on the RVB public key exchange

The following scheme applies to chat as well as e-mail and any other exchange of data via insecure routes. If applied to voice- and video streams, the SAS can be omitted. The following is an outlook to future extensions of the proposed protocol:

The receiver of a message selects a secret integer parameter s at random and publishes two real numbers $x, y=T_s(x)$ e.g. in the form of a public certificate.

The sender in turn selects an arbitrary secret parameter r and computes $Q=N*T_r(y)$, $R=T_r(x)$. Both real numbers are sent to the receiver. The receiver subsequently decodes the message by computing $N=Q/T_s(R)$

Proof:

$$Q=N*T_r(y)=N*T_r(T_s(x))=N*T_{s*r}(x)$$
$$T_s(R)=T_s(T_r(x))=T_{s*r}(x)$$

The methodology corresponds with the El Gamal scheme.

The following table explains the algorithm step by step:

Sender Bob wants to send an encrypted message to Alice. Only Alice shall be able to decipher the message.

For this purpose Alice has previously generated the secret integer number s as well as the real number x at random. After computing $y=T_s(x)$, Alice has published the two public parameters $x, y=T_s(x)$ so that Bob, as well as a multitude of other people, can potentially send encrypted messages to Alice. A key server may serve as repository for such certificate information.

Alice	**Bob**
	Uses Alice's public key which consists of the two numbers x and y .
	Selects a secret and very long integer r at random and computes $R=T_r(x)$.
	Selects a secret message N , which actually represents the key for a symmetric encryption algorithm. The symmetric cipher

	is used to encrypt the text or image or video that Bob wants to send Alices in the first place. Bob computes $Q=N*T_r(y)$.
	The two real numbers R, Q are sent to Alice via an insecure channel.
Computes $N=Q/T_s(R)$ and is now in possession of N , which was originally selected by Bob purely at random. Alice can now set up the symmetric cipher using N as key. By deciphering the actual message (text, image or video) using the symmetric encryption algorithm, Alice is finally able to read the plaintext.	
--	--
Alice has computed: $N=Q/T_s(T_r(x))$ Mathematical transformation yields: $Q=N*T_s(T_r(x))$	Bob has computed: $Q=N*T_r(T_s(x))$ Due to the commutativity of T-polynomials this yields: $Q=N*T_s(T_r(x))$

Table 1: Interactive certificate based on the RVB algorithm

The method requires the sender of a message to select a secret r and to compute $R=T_r(x)$ from that number. This is why, in contrast to RSA, no fixed signatures can be generated. The proposed method is thus “interactive“. The advantage over RSA is that the RVB algorithm is quantum computer proof, which may become the decisive advantage over any conventional method.

The proposed protocol will feature this interactive certificate method under the name “EI“, which stands for “Electronic Identity“.

3. The protocol

The Session Initiation protocol (SIP) is by far the most widely used VoIP protocol. In order to guarantee compatibility with any existing SIP infrastructure, it is neither possible to extend the SIP protocol nor to introduce a new audio codec. A technology that has previously been used successfully in products from Global IP Telecommunications has been extended – the embedding of highly compressed audio in payload of a popular and omnipresent codec that only performs moderate to no data compression. To a SIP server or media gateway, audio data this way appears to be noise, while compatible clients are able to decrypt such data packets and to decode the embedded audio data as well as accompanying control data. While softclients from Global IP Telecommunications were previously only capable of encoding encrypted audio at 8 kHz sampling rate, the concept has been extended to 16 kHz sampling rate with the very same source code.

A multitude of SIP voice clients exist on the market. Most will not support peer-to-peer (P2P) RTP encryption in the beginning. Auto-sensing if a remote VoIP client supports the protocol is vital. A compatible client emits a discovery packet from time to time while most audio packets contain static. After a couple of seconds without adequate response from the peer, the key exchange is stopped and the user is notified appropriately.

3.1 Embedding highly compressed audio in RTP audio packets with fixed size

The two widespread ITU-T standards for audio compression, G.711 [6] and G.722 [7] both feature 160 bytes of space for audio payload. G.711 was released for usage in 1972. Two slightly different versions exist:

1.) µ-law: Non-uniform (logarithmic) quantization with 8 bits at 8 kHz sampling rate is used to represent each sample.

2.) A-law: Linear quantization with 8 bits at 8 kHz sampling rate is used to represent each sample.

µ-law is used primarily in North America, while A-law is in use almost everywhere else. G.711 A-law is currently the most widely used codec in Europe.

HD telephony typically relies on the G.722 ITU-T wideband audio codec, which is capable of encoding voice data with a bandwidth of up to 7kHz at 16 kHz sampling rate.

In order to enable a telephony client to encode highly compressed audio data at the two sampling rates 8 kHz and 16 kHz, the opus audio codec has been selected [8]. The codec, which is officially supported in Matroska, WebM, MPEG-TS and MP4 containers, is utilized in VoIP mode at a constant bitrate of 24000 bps at 8 kHz as well as at 16 kHz sampling rate. Audio payload consequently consumes only 60 bytes of the theoretically available 160 bytes. The proposed protocol reserves 64 bytes for audio payload.

Encrypted audio packets further contain markers and a 32 bit packet counter that guarantees a cycle time of 994 days, which is almost certainly longer than any telephone conversation should possibly last. 80 bytes currently remain unassigned to any purpose. Possible applications include quality of service and authentication.

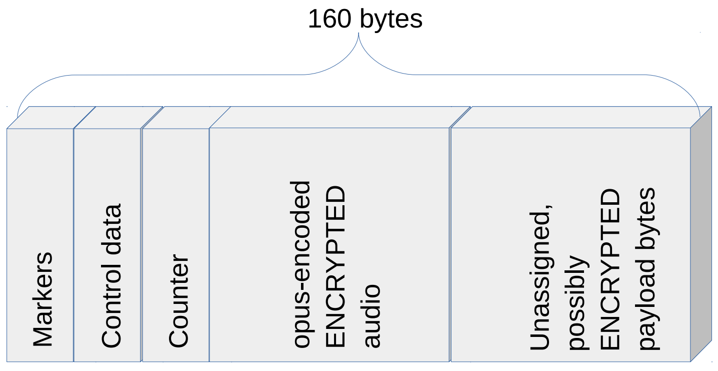

160 bytes
Markers
Control data
Counter
opus-encoded ENCRYPTED audio
Unassigned, possibly ENCRYPTED payload bytes

Fig. 1: G.711 / G.722 RTP packet containing encrypted data

3.2 RVB key negotiation

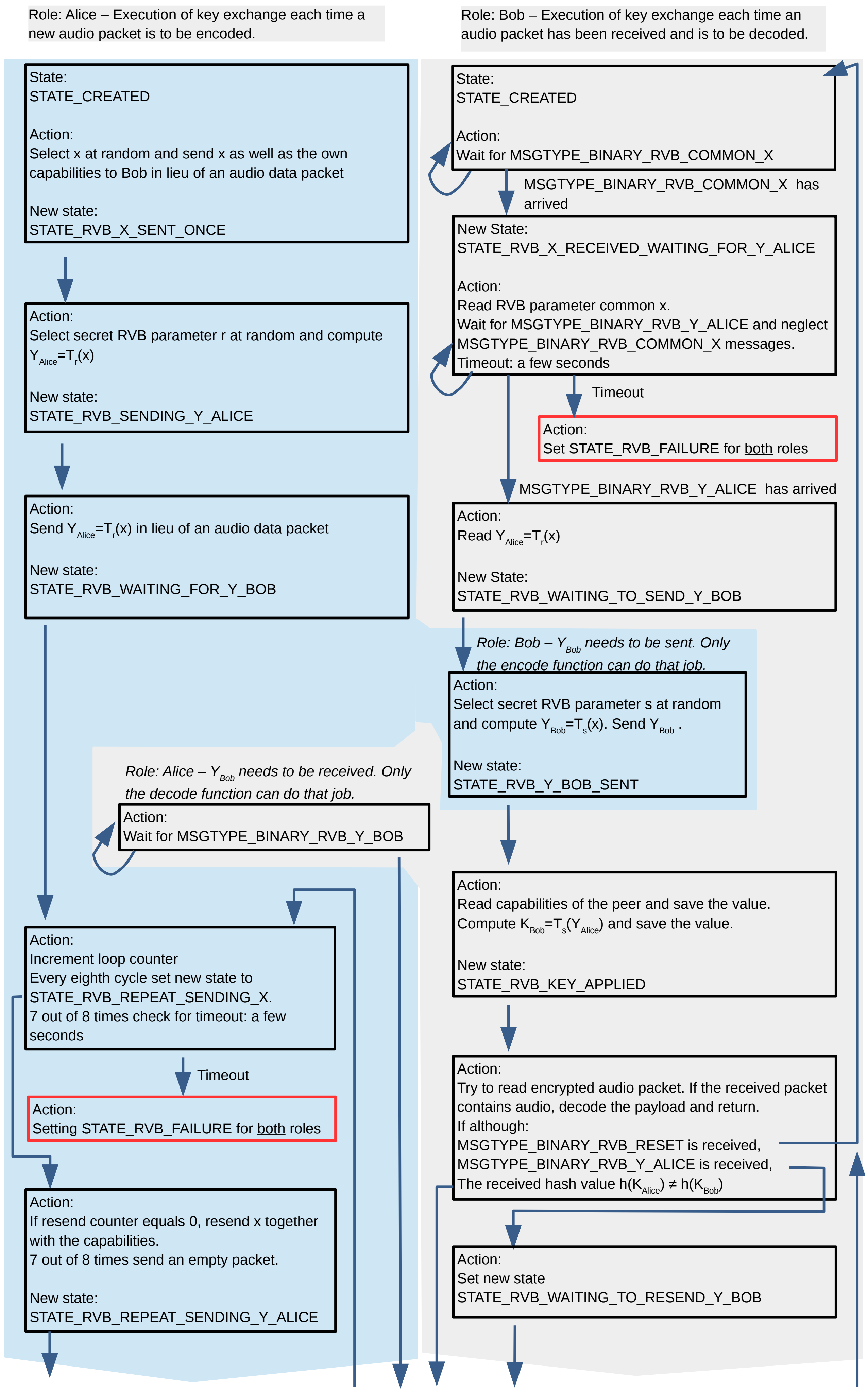
While classic DH algorithms consume a lot of CPU time, the RVB algorithm does not and can thus be implemented without the need to run an extra thread that performs the lengthy computation in the background.

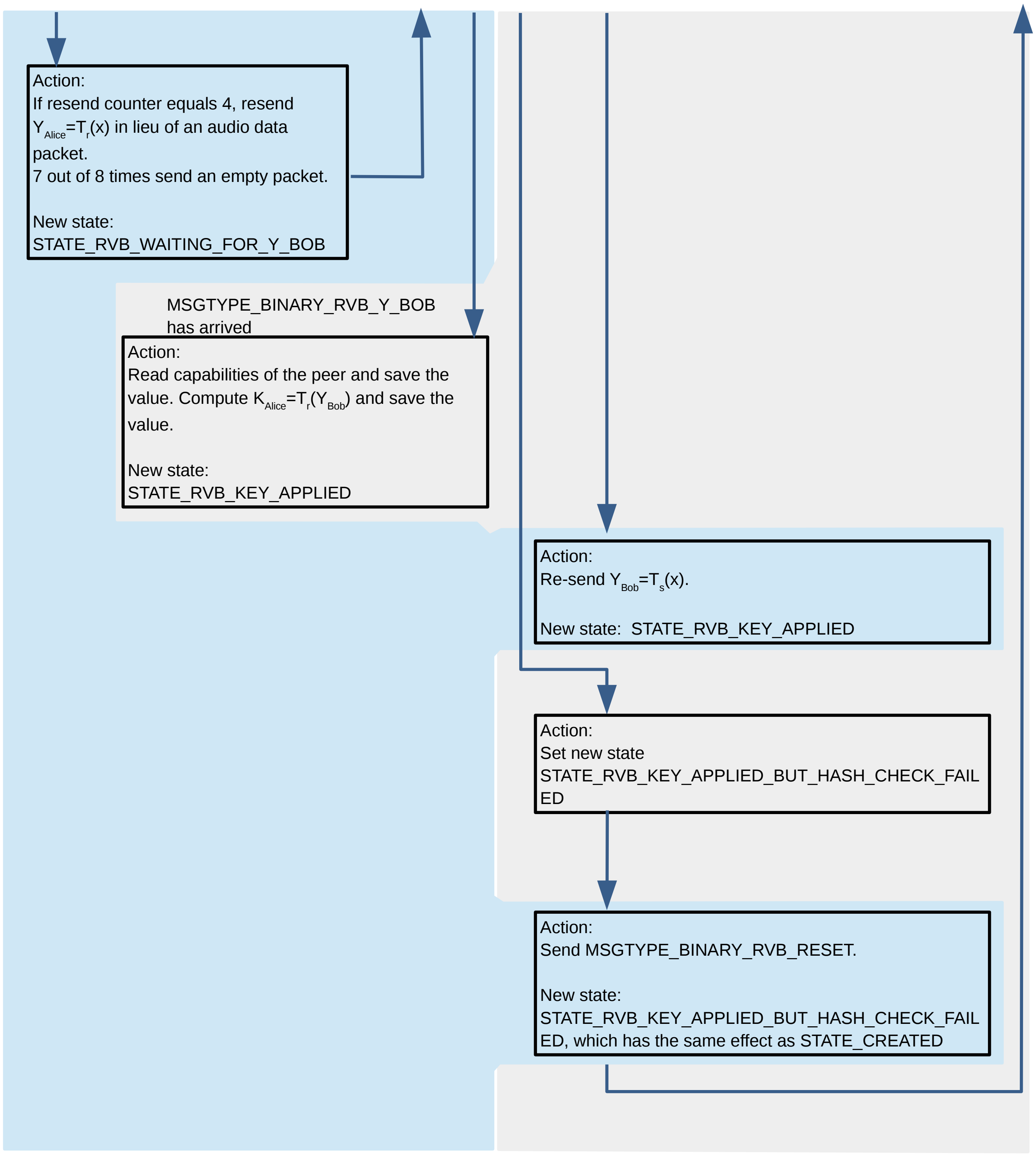
A compatible client initiates the key exchange by transmitting the Common_X_RTP packet. The packet replaces a standard audio packet and contains the publicly known RVB parameter x, as well as capabilities of the compatible client. .

A compatible client that receives the Common_X_RTP packet will act in the role named “Bob” while the compatible client that originally started the key exchange acts in the role named “Alice”.

Both peers simultaneously act in both roles. As a matter of consequence, two shared secret values are computed. Of the two shared secret values which act as shared secret keys, the one that is negotiated in the role “Alice” is used to encrypt outbound messages, while the other, that is negotiated in the role “Bob” is used to decrypt inbound messages. This way it is impossible to run into racing problems which could otherwise occur as two clients that both start a key exchange by sheer coincidence at the very same time could both “think” they were “Alice”.

The following chart depicts how the state engines on both endpoints cooperate. The left side shows the states for role “Alice” while the right side shows the the states for role “Bob”. In both roles it is necessary to send and to receive data.


Role: Alice – Execution of key exchange each time a new audio packet is to be encoded.
Role: Bob – Execution of key exchange each time an audio packet has been received and is to be decoded.
State: STATE_CREATED Action: Select x at random and send x as well as the own capabilities to Bob in lieu of an audio data packet New state: STATE_RVB_X_SENT_ONCE
Action: Select secret RVB parameter r at random and compute $Y_{Alice}=T_r(x)$ New state: STATE_RVB_SENDING_Y_ALICE
Action: Send $Y_{Alice}=T_r(x)$ in lieu of an audio data packet New state: STATE_RVB_WAITING_FOR_Y_BOB
Role: Alice – Y_{Bob} needs to be received. Only the decode function can do that job.
Action: Wait for MSGTYPE_BINARY_RVB_Y_BOB
Action: Increment loop counter Every eighth cycle set new state to STATE_RVB_REPEAT_SENDING_X. 7 out of 8 times check for timeout: a few seconds
Timeout
Action: Setting STATE_RVB_FAILURE for both roles
Action: If resend counter equals 0, resend x together with the capabilities. 7 out of 8 times send an empty packet. New state: STATE_RVB_REPEAT_SENDING_Y_ALICE
State: STATE_CREATED Action: Wait for MSGTYPE_BINARY_RVB_COMMON_X
MSGTYPE_BINARY_RVB_COMMON_X has arrived
New State: STATE_RVB_X_RECEIVED_WAITING_FOR_Y_ALICE Action: Read RVB parameter common x. Wait for MSGTYPE_BINARY_RVB_Y_ALICE and neglect MSGTYPE_BINARY_RVB_COMMON_X messages. Timeout: a few seconds
Timeout
Action: Set STATE_RVB_FAILURE for both roles
MSGTYPE_BINARY_RVB_Y_ALICE has arrived
Action: Read $Y_{Alice}=T_r(x)$ New State: STATE_RVB_WAITING_TO_SEND_Y_BOB
Role: Bob – Y_{Bob} needs to be sent. Only the encode function can do that job.
Action: Select secret RVB parameter s at random and compute $Y_{Bob}=T_s(x)$. Send Y_{Bob} . New state: STATE_RVB_Y_BOB_SENT
Action: Read capabilities of the peer and save the value. Compute $K_{Bob}=T_s(Y_{Alice})$ and save the value. New state: STATE_RVB_KEY_APPLIED
Action: Try to read encrypted audio packet. If the received packet contains audio, decode the payload and return. If although: MSGTYPE_BINARY_RVB_RESET is received, MSGTYPE_BINARY_RVB_Y_ALICE is received, The received hash value $h(K_{Alice}) \neq h(K_{Bob})$
Action: Set new state STATE_RVB_WAITING_TO_RESEND_Y_BOB



Action:
If resend counter equals 4, resend Y_{Alice}=T_r(x) in lieu of an audio data packet.
7 out of 8 times send an empty packet.
New state:
STATE_RVB_WAITING_FOR_Y_BOB
MSGTYPE_BINARY_RVB_Y_BOB has arrived
Action:
Read capabilities of the peer and save the value. Compute K_{Alice}=T_r(Y_{Bob}) and save the value.
New state:
STATE_RVB_KEY_APPLIED
Action:
Re-send Y_{Bob}=T_s(x).
New state: STATE_RVB_KEY_APPLIED
Action:
Set new state STATE_RVB_KEY_APPLIED_BUT_HASH_CHECK_FAILED
Action:
Send MSGTYPE_BINARY_RVB_RESET.
New state:
STATE_RVB_KEY_APPLIED_BUT_HASH_CHECK_FAILED, which has the same effect as STATE_CREATED

Fig. 2 The state engine

It is anything but trivial to enable both roles to act both as sender as well as receiver. The reason for this is the fact that role “Alice” is executed in the context of the encoding G.711 or G.722 codec while the role “Bob” runs in the context of a decoding codec. In most implementations, both codecs do not share any memory. The actual implementation overcomes this issue by providing access to a shared crypto context. The shared crypto context is actually shared with all media streams and the SIP INVITE dialog, which allows to control the key exchange from the client application and to display Short Authentication Strings to the user.

The Common_X_RTP packet is the very first packet of an RVB public key exchange. It is compiled by the encoding codec (role “Alice”).

A compatible client that receives the Common_X_RTP packet acts in the role “Bob”. The packet contains the capabilities of the remote client, markers and the common public RVB parameter x.

The mantissa of x is nothing but a true random number, which is 1216 bit in length. This is equivalent to 19 64-bit unsigned integer numbers or 366 decimal digits. Due to rounding errors that cannot be avoided when computing Chebyshev polynomials, approx. 66% of the number of bits will be inaccurate in the end, which corresponds with an accuracy of 120 decimal digits of the shared secret. 100 decimal digits (which is the targeted accuracy) can be assumed to be identical without the need to perform extensive checks.

Bytes 0, 1, 5 and 6 in the Common_X_RTP packet act as markers. In addition to this, the mantissa is carefully selected so that the entropy of the 152 bytes is high. The so-called “Frequency test”, a simple but powerful randomness test, is utilized to make sure that the number of 1's is not greater than the number of 0's by at maximum four (and vice versa). The test is very basic, but has proved to yield a very good indication in many tests of diverse pseudorandom number generators.

A compatible client will be able to identify the Common_X_RTP packet shortly after its reception, saves the common public RVB parameter x, selects an integer number s at random, computes $T_s(x)$ and sends the public parameter $T_s(x)$ to the peer.

Key negotiation starts with the RVB parameter x, which is a publicly known real number in the interval $[-1,1]$. For the sake of simplicity, the current implementation generates x in the interval $[-0.5,0.5]$. Only the mantissa is copied to the data packet:

Byte number	Encoded data	Remarks
0	0x01	Targetted decimal precision: 100 decimal digits at minimum for the resulting shared secret key
1	MSGTYPE_BINARY_RVB_COMMON_X	
2	8 least significant bits of the capabilites of the local client	
3	Bits 15 .. 8 of the capabilites of the local client	
4	Bits 23 .. 16 of the capabilites of the local client	
5	Bits 31 .. 24 of the capabilites of the local client	
6	0x55	Marker
7	0xaa	Marker
8 .. 159	152 bytes mantissa of x (LSB first)	TTMath library [9]

Table 2: The Common_X_RTP packet

The capabilities of the local client are defined as 32 bit unsigned integer. The 12 least significant bits specify the maximum size of symmetric keys in bytes. For P2P encryption of RTP streams, this number is heavily oversized. An RTP packet that is supposed to contain G.711- or G.722-encoded voice data can at maximum hold 1280 bits (160 bytes). The RVB parameter x consumes 152 bytes and the final shared key can at maximum reach 400 bits in accuracy while 332 bits can be assumed to be identical on both ends. The effective length of classic DH keys would even be less. For P2P encryption of RTP streams, the maximum size of symmetric keys in bytes shall be limited to 127. This number can be encoded in exactly 7 bits, which is just the amount of data that the Y_BOB_RTP packet can accommodate.

Bits 19 .. 12 specify symmetric ciphers that belong to the set of supported ciphers. AES in CBC mode with 128 bit key length is assumed to be supported by any client. So far only bit # 12 is assigned to an additional symmetric cipher. It specifies the Giant Block Size Polymorphic Cipher (GBPMC) [10] with a variable block length ranging between 16 bytes .. 128 kilobytes. If bit 12 is set, the GBPMC cipher takes precedence over the default AES_CBC_128 cipher.

The table below shows the two different ways for the encoding of capabilities:

Bit range	Capabilities: 32 bit unsigned int	Bit range	Capabilities: 15 bit compressed representation
31 .. 21	Reserved	-	-
20	KEYEX_RVB_SUPPORTED	-	-
19 .. 12	Supported symmetric ciphers	14 .. 7	Supported symmetric ciphers
11 .. 0	Maximum supported size of symmetric keys [bytes]	6 ..0	Maximum supported size of symmetric keys [bytes]

Table 3: Capabilities as exchanged by both endpoints of a P2P audio/video real-time session

Soon after dispatching the Common_X_RTP packet, the sender (role “Alice”) will select the secret integer number r and compute with its help the public RVB parameter $Y_{Alice}=T_r(x)$. This parameter is subsequently dispatched to the peer in the Y_ALICE_RTP packet:

Byte number	Encoded data	Remarks
0	0x01	Targetted decimal precision: 100 decimal digits at minimum for the resulting shared secret key
1	MSGTYPE_BINARY_RVB_Y_ALICE	
2	8 least significant bits of the exponent of Y_Alice	
3	8 most significant bits of the exponent of Y_Alice	
4	0 if Y_Alice is positive, 1 if the number is negative	
5	0x00	The byte is not assigned to any purpose
6	0x55	Marker
7	0xaa	Marker
8 .. 159	152 bytes mantissa of Y_Alice (LSB first)	TTMath library [9]

Table 4: The Y_ALICE_RTP packet

The sender (role “Alice”) will repeat sending the Common_X_RTP packet and the Y_ALICE_RTP packet from time to time. In most cases, the sender will dispatch data packets that contain static. A callee who accepts an inbound call will hear noisy packets from time to time if he does not have a compatible client. If the number of static packets was smaller, that user would hear an enormously loud noise, which would be unacceptable.

It is typical for VoIP phone calls that one client already sends voice data while the other is not yet ready to decode audio or vice versa. This is the primary reason why the Common_X_RTP- and Y_ALICE_RTP packets need to be sent repeatedly.

Once the receiver (role “Bob”) has read the remote Common_X_RTP packet, it is able to compute his public RVB parameter $Y_{Bob}=T_s(x)$. This parameter is subsequently dispatched to the peer in the Y_ALICE_RTP packet. Previously, the receiver will have selected the RVB parameter and secret integer number s at random. As soon as the sender (role “Alice”) receives and decodes this message, it knows for sure that the peer is compatible.

Byte number	Encoded data	Remarks
0	0x01	Targetted decimal precision: 100 decimal digits at minimum for the resulting shared secret key
1	MSGTYPE_BINARY_RVB_Y_BOB	
2	8 least significant bits of the exponent of Y_Bob	
3	8 most significant bits of the exponent of Y_Bob	
4	Bit 0 is 0 if Y_Bob is positive, 1 if the number is negative. Bits 7 .. 1 contain the 7 least significant bits of the own capabilites	Own capabilities contains a subset of information stored in the ownCapabilites value. These 7 bits contain the maximum allowed length of symmetric keys [bytes]. E.g. 7: => 56 bit, 32: => 256 bit.
5	Bits 15 .. 8 of the own capabilites	Supported symmetric ciphers
6	0x55	Marker
7	0xaa	Marker
8 .. 159	152 bytes mantissa of Y_Bob (LSB first)	TTMath library [9]

Table 5: The Y_BOB_RTP packet

The capabilities of the receiver (role “Bob”) are only available in compressed form, mainly due to the fact that the Y_BOB_RTP packet is cram-full.

Once the sender (role “Alice”) has received this packet, both clients can compute the shared key K. The sender (role “Alice”) computes $K_{Alice}=T_s(Y_{Bob})$ while the receiver (role “Bob”) computes $K_{Bob}=T_r(Y_{Alice})$. Only the first 103 characters of the character string of K_{Alice} and K_{Bob} are considered as the likelihood for these digits to be identical at both endpoints is very close to 1.

As long as the receiver (role “Bob”) does not receive any encrypted audio data packets, it re-sends Y_BOB_RTP packets. If although an encrypted audio packet is received, the receiver can check the Short Authentication String, which is part of the encrypted audio packet during the first seconds of an encrypted telephone call. The receiver is either sure that the shared key is correct or that a MSGTYPE_BINARY_RVB_RESET messages needs to be dispatched, which causes the peer to initiate a brand new key exchange.

The encrypted audio data packet contains between 16 .. 160 bytes of opus-encoded and encrypted audio data. 50 such packets are dispatched each second. The packets further contain a packet counter that is independent from the RTP packet counter. G.711a/u and G.722 data packets are 160 bytes in length with 79 bytes unused. This payload is reserved for future use. Future extensions might provide authentication, quality of service or stacked key exchanges.

The following table shows the construction of an encrypted audio data packet:

Byte number	Encoded data	Remarks
0	0x01	Targetted decimal precision: 100 decimal digits at minimum for the resulting shared secret key
1	MSGTYPE_ENCRYPTED	
2	8 least significant bits of the message counter	Implementations are free to either perform montonous increment or decrement, as well as to provide random numbers
3	Bits 15 .. 8 of the message counter	
4	Bits 23 .. 16 of the message counter	
5	Bits 31 .. 24 of the message counter	
6	0x55	Marker

7	0xaa	Marker
8	Hash function	The hash function used by the sender to derive the key for the symmetric cipher. Currently supported is Keccak-512 (SHA-3) [11]
9	Symmetric cipher function	The symmetric cipher function used by the sender to encrypt audio data. Currently supported are: - AES in CBC mode with 128 bit key size -GBPMC with key size limited to hash function
10 .. 13	Short Authentication Sring as 32 bit unsigned integer	SAS is derived from the key using Keccak-256. Data packets may contain different information after the first 10 seconds (not implemented so far).
14	Number of key	Currently reserved for dynamic key switching
15 .. 16	Length of decrypted audio [bytes]	Byte #15, bit 7: Is 1 if audio follows => this bit is always set. Byte #15, bit 6: Is 1 if some other payload follows (e.g. INTEGER data). Audio: Currently the opus codec produces 60 bytes of compressed audio per 20ms at 8kHz and 16kHz sample rate and 160 bytes of compressed audio per 20ms at 24kHz sample rate..
17 ..	Compressed and encrypted audio	Mote data might follow. This depends on byte #15, bit 6. If this bit is set, another page follows. So far there exists no definition for such pages.

Table 6: Encrypted audio packet

The Short Authentication String (SAS) that is displayed to the user is the XORed sum of the SAS of role “Alice” and the SAS of role “Bob”. The resulting 32 bit number is the same at both endpoints.

If the receiver (role “Bob”) computes a Short Authentication String (SAS) that differs from the locally computed value, the receiver issues a MSGTYPE_BINARY_RVB_RESET message. As soon as the peer (role “Alice”) reads the message, a brand new key exchange starts and the timeout counter is reset.

Byte number	Encoded data	Remarks
0	0x01	
1	MSGTYPE_BINARY_RVB_RESET	
2 .. 5	Undefined	Future use is possible
6	0x55	
7	0xaa	
8..159	Undefined	Future use is possible

Table 7: Reset message

4. Embedding of key negotiation functionality in existing source code

4.1 RVB key negotiation and encryption of audio data

The interface of the source code has been designed as replacement for the standard codec function encode() and decode(). The example below shows the integration into source code that calls G.722 functions:

```
bool CCodec_PCM_G722::encode(
  IN const std::string &strRawPCMData,
  OUT std::stringlist &strEncodedDataList)
{
  if (!pg722_encoder_internal_state)
    return false;

  // Clear output buffer.
  strEncodedDataList.clear();

  std::string strEncodedData;

  // Set up input buffers.
  int iBytesToCode = (int)strRawPCMData.size();
  short *pInputBuffer = (short *)strRawPCMData.data();

  // Set output buffer.
  unsigned char * pBitstreamBuffer = (unsigned char *) m_strBitstreamBuffer.data();

  int outlen=160;

  if (m_pDialogCryptoContext) {
    if (m_pDialogCryptoContext->iRTPEncryptionType & 0xfffffffe) {
      if (!m_pP2P_Enc)
        m_pP2P_Enc=new P2P_Enc(0,m_pDialogCryptoContext); // role: 0: sender

      if (m_pP2P_Enc)
        outlen=m_pP2P_Enc->P2PencryptAudio(P2P_Enc::G722,pInputBuffer,
                                           iBytesToCode,pBitstreamBuffer,outlen);
      else
        outlen=0;

      if (!outlen)
        outlen = g722_encode(pg722_encoder_internal_state,(uint8_t *)
                             pBitstreamBuffer,(int16_t *)pInputBuffer,(iBytesToCode>>1));
    } // if (m_pDialogCryptoContext->iRTPEncryptionType & 0xfffffffe)
    else
      outlen = g722_encode(pg722_encoder_internal_state,(uint8_t *)
                           pBitstreamBuffer,(int16_t *)pInputBuffer,(iBytesToCode>>1));
  } // if (m_pDialogCryptoContext)
  else
    outlen = g722_encode(pg722_encoder_internal_state,(uint8_t *)
                         pBitstreamBuffer,(int16_t *)pInputBuffer,(iBytesToCode>>1));

  // Add encoded frame to output stream.
  strEncodedData += std::string((char *)pBitstreamBuffer,outlen);

  // Add encoded data to result list.
  strEncodedDataList.push_back(strEncodedData);

  return true;
}
```

```
bool CCodec_PCM_G722::decode(
  IN const std::String &strEncodedData,
  IN bool bMarkerFlag,
  OUT std::stringlist &strDecodedDataList)
{
  if (!pg722_decoder_internal_state)
    return false;

  // Clear the output buffer.
  strDecodedDataList.clear();

  int iBytesToCode = (int) strEncodedData.size();
  char * pInputBuffer = (char *)strEncodedData.data();

  // Set output buffer.
  short * pOutputBuffer = (short *)m_strPCMBuffer.data();

  int out_samples=320;

  if (m_pDialogCryptoContext) {
    if (m_pDialogCryptoContext->iRTPEncryptionType & 0xfffffffe) {
      if (!m_pP2P_Enc)
        m_pP2P_Enc=new P2P_Enc(1,m_pDialogCryptoContext); // role: 1: receiver

      if (m_pP2P_Enc)
        out_samples=m_pP2P_Enc->P2PdecryptAudio(P2P_Enc::G722,(unsigned char*)
                                           pInputBuffer,iBytesToCode,pOutputBuffer,out_samples);
      else
        out_samples=0;

      if (!out_samples)
        out_samples = g722_decode(pg722_decoder_internal_state,(int16_t *)
                                          pOutputBuffer,(uint8_t *)pInputBuffer,iBytesToCode);
    } // if (m_pDialogCryptoContext->iRTPEncryptionType & 0xfffffffe)
    else
      out_samples = g722_decode(pg722_decoder_internal_state,(int16_t *)
                                        pOutputBuffer,(uint8_t *)pInputBuffer,iBytesToCode);
  } // if (m_pDialogCryptoContext)
  else
    out_samples = g722_decode(pg722_decoder_internal_state,(int16_t *)
                                      pOutputBuffer,(uint8_t *)pInputBuffer,iBytesToCode);

  if (out_samples!=320)
    return false;

  // Add encoded frame to output stream.
  strDecodedDataList.push_back(std::string((char *)pOutputBuffer,(out_samples<<1)));

  return true;
}
```

The encoder function P2PencryptAudio() as well as the decoder function P2PdecryptAudio() both return 0 if they cannot handle a data packet. In this case, it is advisable to call the G.711 or G.722 encode- or decode function.

If P2PdecryptAudio() detects a data packet that contains key exchange payload, the function outputs 20ms of sine wave audio in order to give the user some kind of feedback.

4.2 Encryption of video data

Video is typically an add-on to audio communication. This is why the key exchange is performed in the audio stream. Video, if present, although enables for additional functionality due to the fact that payload is somewhat variable in length.

The current implementation simply encrypts video packets and places the encrypted payload – quite often with additional padding – into a new packet. The first two bytes contain the length of the plaintext. This is necessary as padding can add 0 .. 15 bytes for AES.

As long as both keys for the symmetric cipher (role “Alice” AND role “Bob”) are not yet set, the video codec only receives a black picture that is encoded by the codec and sent in the clear to the peer. As soon as the keys are set, real video data is encoded and encrypted.

Literature

[1] G. Brands, C.B. Roellgen, K.U. Vogels, QRKE: Quantum-Resistant Public Key Exchange, 2015, https://www.researchgate.net/publication/282286331_QRKE_Quantum-Resistant_Public_Key_Exchange

[2] G. Brands, C.B. Roellgen, K.U. Vogels, QRKE: Extensions, 2015, https://www.researchgate.net/publication/283213947_QRKE_Extensions

[3] G. Brands, C.B. Roellgen, K.U. Vogels, QRKE: Resistance to Attacks using the Inverse of the Cosine Representation of Chebyshev Polynomials, 2016, https://www.researchgate.net/publication/291945494_QRKE_Resistance_to_Attacks_using_the_Inverse_of_the_Cosine_Representation_of_Chebyshev_Polynomials

[4] H. Schulzrinne, S. Casner, R. Frederick, V. Jacbson, RTP: A Transport Protocol for Real-Time Applications, 2003, https://tools.ietf.org/html/rfc3550

[5] M. Baugher, D. Mcgrew, M.Naslund, E. Carrara, K. Norrman, The Secure Real-time Transport Protocol (SRTP), 2004, https://tools.ietf.org/html/rfc3711

[6] ITU-T, G.711 : Pulse code modulation (PCM) of voice frequencies, 1988, https://www.itu.int/rec/T-REC-G.711

[7] ITU-T, G.722 : G.722 : 7 kHz audio-coding within 64 kbit/s, 2012, https://www.itu.int/rec/T-REC-G.722/e

[8] JM. Valin, K. Vos, T. Terryberry, Definition of the Opus Audio Codec, 2012, https://tools.ietf.org/html/rfc6716

[9] T. Sowa, TTMath mathematic C++ library, 2012, http://www.ttmath.org/

[10] C.B. Roellgen, Approaching the Perfect Cipher by Trespassing the block size confinement – The Polymorphic Giant Block Encryption Algorithm, 2010, http://www.ciphers.de/documents/whitepapers/giant_block_size_polymorphic_cipher.pdf

[11] G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, The Keccak reference , 2011, http://keccak.noekeon.org/Keccak-reference-3.0.pdf