Internet Engineering Task Force (IETF)	P. Zimmermann
Request for Comments: 6189	Zfone Project
Category: Informational	A. Johnston, Ed.
ISSN: 2070-1721	Avaya
	J. Callas
	Apple, Inc.
	April 2011

ZRTP: Media Path Key Agreement for Unicast Secure RTP

Abstract

This document defines ZRTP, a protocol for media path Diffie-Hellman exchange to agree on a session key and parameters for establishing unicast Secure Real-time Transport Protocol (SRTP) sessions for Voice over IP (VoIP) applications. The ZRTP protocol is media path keying because it is multiplexed on the same port as RTP and does not require support in the signaling protocol. ZRTP does not assume a Public Key Infrastructure (PKI) or require the complexity of certificates in end devices. For the media session, ZRTP provides confidentiality, protection against man-in-the-middle (MiTM) attacks, and, in cases where the signaling protocol provides end-to-end integrity protection, authentication. ZRTP can utilize a Session Description Protocol (SDP) attribute to provide discovery and authentication through the signaling channel. To provide best effort SRTP, ZRTP utilizes normal RTP/AVP (Audio-Visual Profile) profiles. ZRTP secures media sessions that include a voice media stream and can also secure media sessions that do not include voice by using an optional digital signature.

Status of This Memo

This document is not an Internet Standards Track specification; it is published for informational purposes.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Not all documents approved by the IESG are a candidate for any level of Internet Standard; see Section 2 of RFC 5741.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at http://www.rfc-editor.org/info/rfc6189.

Table of Contents

TOC

1. Introduction

ZRTP is a key agreement protocol that performs a Diffie-Hellman key exchange during call setup in the media path and is transported over the same port as the **Real-time Transport Protocol (RTP)** [RFC3550] media stream which has been established using a signaling protocol such as **Session Initiation Protocol (SIP)** [RFC3261]. This generates a shared secret, which is then used to generate keys and salt for a **Secure RTP (SRTP)** [RFC3711] session. ZRTP borrows ideas from **[PGPfone]**. A reference implementation of ZRTP is available in **[Zfone]**.

The ZRTP protocol has some nice cryptographic features lacking in many other approaches to media session encryption. Although it uses a public key algorithm, it does not rely on a public key infrastructure (PKI). In fact, it does not use persistent public keys at all. It uses ephemeral Diffie-Hellman (DH) with hash commitment and allows the detection of man-in-the-middle (MiTM) attacks by displaying a short authentication string (SAS) for the users to read and verbally compare over the phone. It has Perfect Forward Secrecy, meaning the keys are destroyed at the end of the call, which precludes retroactively compromising the call by future disclosures of key material. But even if the users are too lazy to bother with short authentication strings, we still get reasonable authentication against a MiTM attack, based on a form of key continuity. It does this by caching some key material to use in the next call, to be mixed in with the next call's DH shared secret, giving it key continuity properties analogous to Secure SHell (SSH). All this is done without reliance on a PKI, key certification, trust models, certificate authorities, or key management complexity that bedevils the email encryption world. It also does not rely on SIP signaling for the key management, and in fact, it does not rely on any servers at all. It performs its key agreements and key management in a purely peer-to-peer manner over the RTP packet stream.

ZRTP can be used and discovered without being declared or indicated in the signaling path. This provides a best effort SRTP capability. Also, this reduces the complexity of implementations and minimizes interdependency between the signaling and media layers. However, when ZRTP is indicated in the signaling via the zrtp-hash SDP attribute, ZRTP has additional useful properties. By sending a hash of the ZRTP Hello message in the signaling, ZRTP provides a useful binding between the signaling and media paths, which is explained in **Section 8.1**. When this is done through a signaling path that has end-to-end integrity protection, the DH exchange is automatically protected from a MiTM attack, which is explained in **Section 8.1.1**.

ZRTP is designed for unicast media sessions in which there is a voice media stream. For multiparty secure conferencing, separate ZRTP sessions may be negotiated between each party and the conference bridge. For sessions lacking a voice media stream, MiTM protection may be provided by the mechanisms in Sections **8.1.1** or **7.2**. In terms of the RTP topologies defined in **[RFC5117]**, ZRTP is designed for Point-to-Point topologies only.

---

2. Terminology

TOC

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in **[RFC2119]**.

In this document, a "call" is synonymous with a "session".

---

3. Overview

TOC

This section provides a description of how ZRTP works. This description is non-normative in nature but is included to build understanding of the protocol.

ZRTP is negotiated the same way a conventional RTP session is negotiated in an offer/answer exchange using the standard RTP/AVP profile. The ZRTP protocol begins after two endpoints have utilized a signaling protocol, such as SIP, and are ready to exchange media. If **Interactive Connectivity Establishment (ICE)** [RFC5245] is being used, ZRTP begins after ICE has completed its connectivity checks.

ZRTP is multiplexed on the same ports as RTP. It uses a unique header that makes it clearly differentiable from RTP or Session Traversal Utilities for NAT (STUN).

ZRTP support can be discovered in the signaling path by the presence of a ZRTP SDP attribute. However, even in cases where this is not received in the signaling, an endpoint can still send ZRTP Hello messages to see if a response is received. If a response is not received, no more ZRTP messages will be sent during this session. This is safe because ZRTP has been designed to be clearly different from RTP and have a similar structure to STUN packets received (sometimes by non-supporting endpoints) during an ICE exchange.

Both ZRTP endpoints begin the ZRTP exchange by sending a ZRTP Hello message to the other endpoint. The purpose of the Hello message is to confirm that the endpoint supports the protocol and to see what algorithms the two ZRTP endpoints have in common.

The Hello message contains the SRTP configuration options and the ZID. Each instance of ZRTP has a unique 96-bit random ZRTP ID or ZID that is generated once at installation time. ZIDs are discovered during the Hello message exchange. The received ZID is used to look up retained shared secrets from previous ZRTP sessions with the endpoint.

A response to a ZRTP Hello message is a ZRTP HelloACK message. The HelloACK message simply acknowledges receipt of the Hello. Since RTP commonly uses best effort UDP transport, ZRTP has retransmission timers in case of lost datagrams. There are two timers, both with exponential backoff mechanisms. One timer is used for retransmissions of Hello messages and the other is used for retransmissions of all other messages after receipt of a HelloACK.

If an integrity-protected signaling channel is available, a hash of the Hello message can be sent. This allows rejection of false ZRTP Hello messages injected by an attacker.

Hello and other ZRTP messages also contain a hash image that is used to link the messages together. This allows rejection of false ZRTP messages injected during an exchange.

3.1. Key Agreement Modes

TOC

After both endpoints exchange Hello and HelloACK messages, the key agreement exchange can begin with the ZRTP Commit message. ZRTP supports a number of key agreement

modes including both Diffie-Hellman and non-Diffie-Hellman modes as described in the following sections.

The Commit message may be sent immediately after both endpoints have completed the Hello/HelloACK discovery handshake, or it may be deferred until later in the call, after the participants engage in some unencrypted conversation. The Commit message may be manually activated by a user interface element, such as a GO SECURE button, which becomes enabled after the Hello/HelloACK discovery phase. This emulates the user experience of a number of secure phones in the Public Switched Telephone Network (PSTN) world **[comsec]**. However, it is expected that most simple ZRTP user agents will omit such buttons and proceed directly to secure mode by sending a Commit message immediately after the Hello/HelloACK handshake.

TOC

3.1.1. Diffie-Hellman Mode Overview

An example ZRTP call flow is shown in **Figure 1**. Note that the order of the Hello/HelloACK exchanges in F1/F2 and F3/F4 may be reversed. That is, either Alice or Bob might send the first Hello message. Note that the endpoint that sends the Commit message is considered the initiator of the ZRTP session and drives the key agreement exchange. The Diffie-Hellman public values are exchanged in the DHPart1 and DHPart2 messages. SRTP keys and salts are then calculated.

The initiator needs to generate its ephemeral key pair before sending the Commit, and the responder generates its key pair before sending DHPart1.

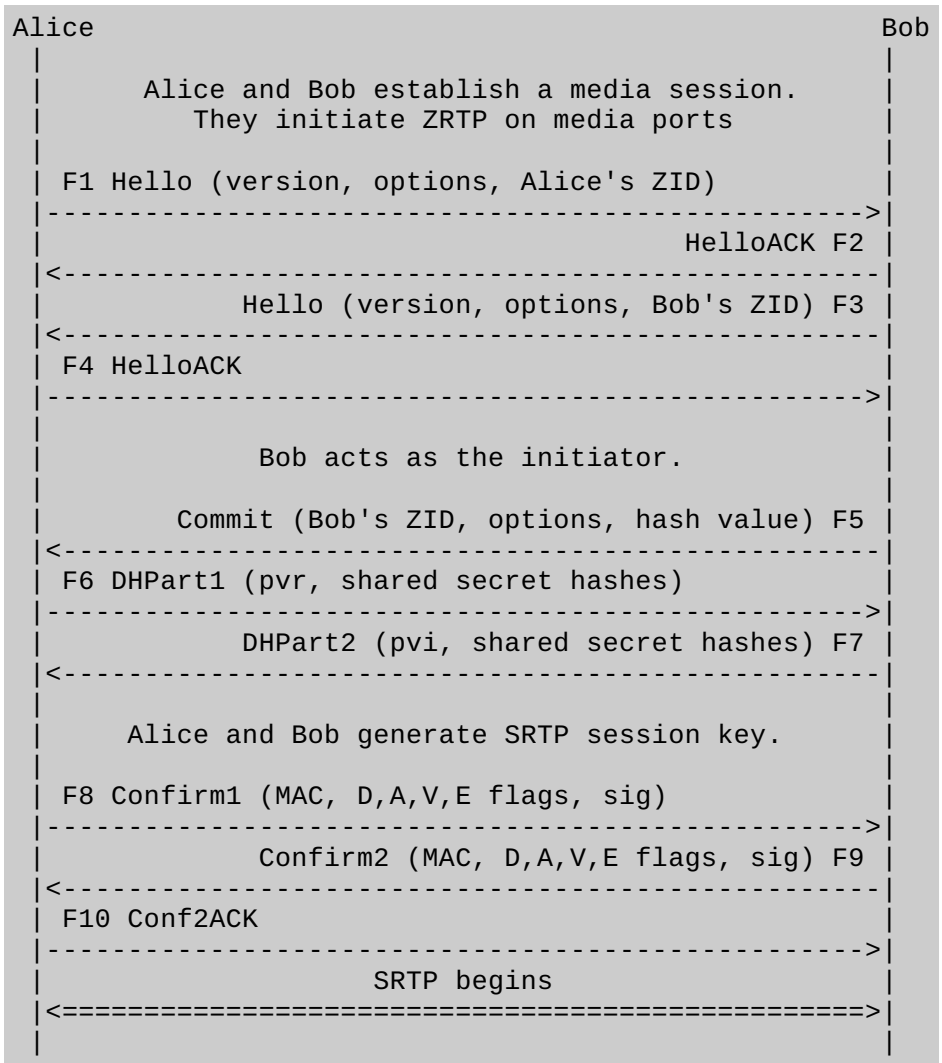


Figure 1: Establishment of an SRTP Session Using ZRTP

ZRTP authentication uses a Short Authentication String (SAS), which is ideally displayed for the human user. Alternatively, the SAS can be authenticated by exchanging an optional digital signature (sig) over the SAS in the Confirm1 or Confirm2 messages (described in **Section 7.2**).

The ZRTP Confirm1 and Confirm2 messages are sent for a number of reasons, not the least of which is that they confirm that all the key agreement calculations were successful and thus the encryption will work. They also carry other information such as the Disclosure flag (D), the Allow Clear flag (A), the SAS Verified flag (V), and the Private Branch Exchange (PBX) Enrollment flag (E). All flags are encrypted to shield them from a passive observer.

TOC

3.1.2. Preshared Mode Overview

In the Preshared mode, endpoints can skip the DH calculation if they have a shared secret from a previous ZRTP session. Preshared mode is indicated in the Commit message and results in the same call flow as Multistream mode. The principal difference between Multistream mode and Preshared mode is that Preshared mode uses a previously cached shared secret, rs1, instead of an active ZRTP Session key as the initial keying material.

This mode could be useful for slow processor endpoints so that a DH calculation does not need to be performed every session. Or, this mode could be used to rapidly re-establish an earlier session that was recently torn down or interrupted without the need to perform another DH calculation.

Preshared mode has forward secrecy properties. If a phone's cache is captured by an opponent, the cached shared secrets cannot be used to recover earlier encrypted calls, because the shared secrets are replaced with new ones in each new call, as in DH mode. However, the captured secrets can be used by a passive wiretapper in the media path to decrypt the next call, if the next call is in Preshared mode. This differs from DH mode, which requires an active MiTM wiretapper to exploit captured secrets in the next call. However, if the next call is missed by the wiretapper, he cannot wiretap any further calls. Thus, it preserves most of the **self-healing properties** of key continuity enjoyed by DH mode.

TOC

3.1.3. Multistream Mode Overview

Multistream mode is an alternative key agreement method used when two endpoints have an established SRTP media stream between them with an active ZRTP Session key. ZRTP can derive multiple SRTP keys from a single DH exchange. For example, an established secure voice call that adds a video stream uses Multistream mode to quickly initiate the video stream without a second DH exchange.

When Multistream mode is indicated in the Commit message, a call flow similar to **Figure 1** is used, but no DH calculation is performed by either endpoint and the DHPart1 and DHPart2 messages are omitted. The Confirm1, Confirm2, and Conf2ACK messages are still sent. Since the cache is not affected during this mode, multiple Multistream ZRTP exchanges can be performed in parallel between two endpoints.

When adding additional media streams to an existing call, only Multistream mode is used. Only one DH operation is performed, just for the first media stream.

TOC

4. Protocol Description

This section begins the normative description of the protocol.

ZRTP MUST be multiplexed on the same ports as the RTP media packets.

To support best effort encryption from the **Media Security Requirements** [RFC5479], ZRTP uses normal RTP/AVP profile (AVP) media lines in the initial offer/answer exchange. The ZRTP SDP attribute a=zrtp-hash defined in **Section 8** SHOULD be used in all offers and answers to indicate support for the ZRTP protocol.

> ZRTP can be utilized by endpoints that do not have a common signaling protocol but both support SRTP and are relying on a gateway for conversion. As such, it is not always possible for the signaling protocol to relay the zrtp-hash as can be done using SIP.

The Secure RTP/AVP (SAVP) profile MAY be used in subsequent offer/answer exchanges after a successful ZRTP exchange has resulted in an SRTP session, or if it is known that the other endpoint supports this profile. Other profiles MAY also be used.

> The use of the RTP/SAVP profile has caused failures in negotiating best effort SRTP due to the limitations on negotiating profiles using SDP. This is why ZRTP supports the RTP/AVP profile and includes its own discovery mechanisms.

In all key agreement modes, the initiator SHOULD NOT send RTP media after sending the Commit message, and it MUST NOT send SRTP media before receiving either the Conf2ACK or the first SRTP media (with a valid SRTP auth tag) from the responder. The responder SHOULD NOT send RTP media after receiving the Commit message, and MUST NOT send SRTP media before receiving the Confirm2 message.

TOC

4.1. Discovery

During the ZRTP discovery phase, a ZRTP endpoint discovers if the other endpoint supports ZRTP and the supported algorithms and options. This information is transported in a Hello message, which is described in **Section 5.2**.

ZRTP endpoints SHOULD include the SDP attribute a=zrtp-hash in offers and answers, as defined in **Section 8**.

The Hello message includes the ZRTP version, Hash Type, Cipher Type, SRTP authentication tag type, Key Agreement Type, and Short Authentication String (SAS) algorithms that are supported. The Hello message also includes a hash image as described in **Section 9**. In addition, each endpoint sends and discovers ZIDs. The received ZID is used later in the protocol as an index into a cache of shared secrets that were previously negotiated and retained between the two parties.

A Hello message can be sent at any time, but it is usually sent at the start of an RTP session to determine if the other endpoint supports ZRTP and also if the SRTP implementations are compatible. A Hello message is retransmitted using timer T1 and an exponential backoff mechanism detailed in **Section 6** until the receipt of a HelloACK message or a Commit message.

The use of the a=zrtp-hash SDP attribute to authenticate the Hello message is described in **Section 8.1**.

If a Hello message, or any other ZRTP message, indicates that there is a synchronization source (SSRC) collision, an **Error message** MUST be sent with the Error Code indicating SSRC collision, and the ZRTP negotiation MUST be terminated. The procedures of RFC 3550, Section 8.2 **[RFC3550]**, SHOULD be followed by both endpoints to resolve this condition, and if it is resolved, a new ZRTP secure session SHOULD be negotiated.

TOC

4.1.1. Protocol Version Negotiation

This specification defines ZRTP version 1.10. Since new versions of ZRTP may be developed in the future, this specification defines a protocol version negotiation in this section.

Each party declares what version of the ZRTP protocol they support via the version field in the Hello message (**Section 5.2**). If both parties have the same version number in their Hello messages, they can proceed with the rest of the protocol. To facilitate both parties reaching this state of protocol version agreement in their Hello messages, ZRTP should use information provided in the signaling layer, if available. If a ZRTP endpoint supports more than one version of the protocol, it SHOULD declare them all in a list of SIP SDP a=zrtp-hash attributes (defined in **Section 8**), listing separate hashes, with separate ZRTP version numbers in each item in the list.

Both parties should inspect the list of ZRTP version numbers supplied by the other party in the SIP SDP a=zrtp-hash attributes. Both parties SHOULD choose the highest version number that appears in both parties' list of a=zrtp-hash version numbers, and use that version for their Hello messages. If both parties use the SIP signaling in this manner, their initial Hello messages will have the same ZRTP version number, provided they both have at least one supported protocol version in common. Before the ZRTP key agreement can proceed, an endpoint MUST have sent and received Hellos with the same protocol version.

It is best if the signaling layer is used to negotiate the protocol version number. However, the a=zrtp-hash SDP attribute is not always present in the SIP packet, as explained in **Section 8.1**. In the absence of any guidance from the signaling layer, an endpoint MUST send the highest supported version in initial Hello messages. If the two parties send different protocol version numbers in their Hello messages, they can reach an agreement to use a common version, if one exists. They iteratively apply the following rules until they both have matching version fields in their Hello messages and the key agreement can proceed:

- If an endpoint receives a Hello message with an unsupported version number that is higher than the endpoint's current Hello message version, the received Hello message MUST be ignored. The endpoint continues to retransmit Hello messages on the standard **retry schedule**.
- If an endpoint receives a Hello message with a version number that is lower than the endpoint's current Hello message, and the endpoint supports a version that is less than or equal to the received version number, the endpoint MUST stop retransmitting the old version number and MUST start sending a Hello message with the highest supported version number that is less than or equal to the received version number.
- If an endpoint receives a Hello message with an unsupported version number that is lower than the endpoint's current Hello message, the endpoint MUST send an Error message (**Section 5.9**) indicating failure to support this ZRTP version.

The above comparisons are iterated until the version numbers match, or until it exits on a failure to match.

> For example, assume that Alice supports protocol versions 1.10 and 2.00, and Bob supports versions 1.10 and 1.20. Alice initially sends a Hello with version 2.00, and Bob initially sends a Hello with version 1.20. Bob ignores Alice's 2.00 Hello and continues to send his 1.20 Hellos. Alice detects that Bob does not support 2.00 and she stops sending her 2.00 Hellos and starts sending a stream of 1.10 Hellos. Bob sees the 1.10 Hello from Alice and stops sending his 1.20 Hellos and switches to sending 1.10 Hellos. At that point, they have converged on using version 1.10 and the protocol proceeds on that basis.

When comparing protocol versions, a ZRTP endpoint MUST include only the first three octets of the version field in the comparison. The final octet is ignored, because it is not significant for interoperability. For example, "1.1 ", "1.10", "1.11", or "1.1a" are all regarded as a version match, because they would all be interoperable versions.

Changes in protocol version numbers are expected to be infrequent after version 1.10. Supporting multiple versions adds code complexity and may introduce security weaknesses in the implementation. The old adage about keeping it simple applies especially to implementing security protocols. Endpoints SHOULD NOT support protocol versions earlier than version 1.10.

TOC

4.1.2. Algorithm Negotiation

A method is provided to allow the two parties to mutually and deterministically choose the same DH key size and algorithm before a Commit message is sent.

Each Hello message lists the algorithms in the order of preference for that ZRTP endpoint. Endpoints eliminate the non-intersecting choices from each of their own lists, resulting in each endpoint having a list of algorithms in common that might or might not be ordered the same as the other endpoint's list. Each endpoint compares the first item on their own list with the first item on the other endpoint's list and SHOULD choose the faster of the two algorithms. For example:

- Alice's full list: DH2k, DH3k, EC25
- Bob's full list: EC38, EC25, DH3k
- Alice's intersecting list: DH3k, EC25
- Bob's intersecting list: EC25, DH3k
- Alice's first choice is DH3k, and Bob's first choice is EC25.
- Thus, both parties choose EC25 (ECDH-256) because it's faster.

To decide which DH algorithm is faster, the following ranking, from fastest to slowest is defined: DH-2048, ECDH-256, DH-3072, ECDH-384, ECDH-521. These are all defined in **Section 5.1.5**.

If both endpoints follow this method, they may each start their DH calculations as soon as they receive the Hello message, and there will be no need for either endpoint to discard their DH calculation if the other endpoint becomes the initiator.

This method is used only to negotiate DH key size. For the rest of the algorithm choices, it's simply whatever the initiator selects from the algorithms in common. Note that the DH key size influences the Hash Type and the size of the symmetric cipher key, as explained in **Section 5.1.5**.

Unfavorable choices will never be made by this method, because each endpoint will omit from their respective lists choices that are too slow or not secure enough to meet their security policy.

TOC

4.2. Commit Contention

After both parties have received compatible Hello messages, a **Commit message** can be sent to begin the ZRTP key exchange. The endpoint that sends the Commit is known as the initiator, while the receiver of the Commit is known as the responder.

If both sides send Commit messages initiating a secure session at the same time, the following rules are used to break the tie:

- If one Commit is for a DH mode while the other is for Preshared mode, then the Preshared Commit MUST be discarded and the DH Commit proceeds.
- If the two Commits are both Preshared mode, and one party has set the MiTM (M) flag in the Hello message and the other has not, the Commit message from the party who set the (M) flag MUST be discarded, and the one who has not set the (M) flag becomes the initiator, regardless of the nonce values. In other words, for Preshared mode, the phone is the initiator and the PBX is the responder.
- If the two Commits are either both DH modes or both non-DH modes, then the Commit message with the lowest hvi (hash value of initiator) value (for DH Commits), or lowest nonce value (for non-DH Commits), MUST be discarded and the other side is the initiator, and the protocol proceeds with the initiator's Commit. The two hvi or nonce values are compared as large unsigned integers in network byte order.

If one Commit is for Multistream mode while the other is for non-Multistream (DH or Preshared) mode, a software error has occurred and the ZRTP negotiation should be terminated. This should never occur because of the constraints on Multistream mode described in **Section 4.4.3**.

In the event that Commit messages are sent by both ZRTP endpoints at the same time, but are received in different media streams, the same resolution rules apply as if they were received on the same stream. The media stream in which the Commit was received or sent will proceed through the ZRTP exchange while the media stream with the discarded Commit must wait for the completion of the other ZRTP exchange.

If a commit contention forces a DH Commit message to be discarded, the responder's DH public value should only be discarded if it does not match the initiator's DH key size. This will not happen if both endpoints choose a common key size via the method described in **Section 4.1.2**.

TOC

4.3. Matching Shared Secret Determination

The following sections describe how ZRTP endpoints generate and/or use the set of shared secrets s1, auxsecret, and pbxsecret through the exchange of the DHPart1 and DHPart2 messages. This doesn't cover the Diffie-Hellman calculations. It only covers the method whereby the two parties determine if they already have shared secrets in common in their caches.

Each ZRTP endpoint maintains a long-term cache of shared secrets that it has previously negotiated with the other party. The ZID of the other party, received in the other party's Hello message, is used as an index into this cache to find the set of shared secrets, if any exist. This cache entry may contain previously retained shared secrets, rs1 and rs2, which give ZRTP its key continuity features. If the other party is a PBX, the cache may also contain a trusted MiTM PBX shared secret, called pbxsecret, defined in **Section 7.3.1**.

The DHPart1 and DHPart2 messages contain a list of hashes of these shared secrets to allow the two endpoints to compare the hashes with what they have in their caches to detect whether the two sides share any secrets that can be used in the calculation of the session key. The use of this shared secret cache is described in **Section 4.9**.

If no secret of a given type is available, a random value is generated and used for that secret to ensure a mismatch in the hash comparisons in the DHPart1 and DHPart2 messages. This prevents an eavesdropper from knowing which types of shared secrets are available between the endpoints.

Section 4.3.1 refers to the auxiliary shared secret auxsecret. The auxsecret shared secret may be defined by the VoIP user agent out-of-band from the ZRTP protocol. In some cases, it may be provided by the signaling layer as srtps, which is defined in **Section 8.2**. If it is not provided by the signaling layer, the auxsecret shared secret may be manually provisioned in other application-specific ways that are out of band, such as computed from a hashed pass phrase by prior agreement between the two parties or supplied by a hardware token. Or, it may be a family key used by an institution to which the two parties both belong. It is a generalized mechanism for providing a shared secret that is agreed to between the two parties out of scope of the ZRTP protocol. It is expected that most typical ZRTP endpoints will rarely use auxsecret.

For both the initiator and the responder, the shared secrets s1, s2, and s3 will be calculated so that they can all be used later to calculate s0 in **Section 4.4.1.4**. Here is how s1, s2, and s3 are calculated by both parties.

The shared secret s1 will be either the initiator's rs1 or the initiator's rs2, depending on which of them can be found in the responder's cache. If the initiator's rs1 matches the responder's rs1 or rs2, then s1 MUST be set to the initiator's rs1. If and only if that match fails, then if the initiator's rs2 matches the responder's rs1 or rs2, then s1 MUST be set to the initiator's rs2. If that match also fails, then s1 MUST be set to null. The complexity of the s1 calculation is to recover from any loss of cache sync from an earlier aborted session, due to the **Two Generals' Problem** [Byzantine].

The shared secret s2 MUST be set to the value of auxsecret if and only if both parties have matching values for auxsecret, as determined by comparing the hashes of auxsecret sent in the DH messages. If they don't match, s2 MUST be set to null.

The shared secret s3 MUST be set to the value of pbxsecret if and only if both parties have matching values for pbxsecret, as determined by comparing the hashes of pbxsecret sent in the DH messages. If they don't match, s3 MUST be set to null.

If s1, s2, or s3 have null values, they are assumed to have a zero length for the purposes of hashing them later during the s0 calculation in **Section 4.4.1.4**.

The comparison of hashes of rs1, rs2, auxsecret, and pbxsecret is described in **Section 4.3.1**.

4.3.1. Calculation and Comparison of Hashes of Shared Secrets

Both parties calculate a set of non-invertible hashes (implemented via the MAC defined in **Section 5.1.2.1**) of shared secrets that may be present in each of their caches. These

hashes are truncated to the leftmost 64 bits:

```
rs1IDr = MAC(rs1, "Responder")

rs2IDr = MAC(rs2, "Responder")

auxsecretIDr = MAC(auxsecret, Responder's H3)

pbxsecretIDr = MAC(pbxsecret, "Responder")

rs1IDi = MAC(rs1, "Initiator")

rs2IDi = MAC(rs2, "Initiator")

auxsecretIDi = MAC(auxsecret, Initiator's H3)

pbxsecretIDi = MAC(pbxsecret, "Initiator")
```

The responder sends rs1IDr, rs2IDr, auxsecretIDr, and pbxsecretIDr in the DHPart1 message. The initiator sends rs1IDi, rs2IDi, auxsecretIDi, and pbxsecretIDi in the DHPart2 message.

The responder uses the locally computed rs1IDi, rs2IDi, auxsecretIDi, and pbxsecretIDi to compare against the corresponding fields in the received DHPart2 message. The initiator uses the locally computed rs1IDr, rs2IDr, auxsecretIDr, and pbxsecretIDr to compare against the corresponding fields in the received DHPart1 message.

From these comparisons, s1, s2, and s3 are calculated per the methods described in **Section 4.3**. The secrets corresponding to matching hashes are kept while the secrets corresponding to the non-matching ones are replaced with a null, which is assumed to have a zero length for the purposes of hashing them later. The resulting s1, s2, and s3 values are used later to calculate s0 in **Section 4.4.1.4**.

For example, consider two ZRTP endpoints who share secrets rs1 and pbxsecret (defined in **Section 7.3.1**). During the comparison, rs1ID and pbxsecretID will match but auxsecretID will not. As a result, s1 = rs1, s2 will be null, and s3 = pbxsecret.

TOC

4.3.2. Handling a Shared Secret Cache Mismatch

A shared secret cache mismatch is defined to mean that we expected a cache match because rs1 exists in our local cache, but we computed a null value for s1 (per the method described in **Section 4.3**).

If one party has a cached shared secret and the other party does not, this indicates one of two possible situations. Either there is a MiTM attack or one of the legitimate parties has lost their cached shared secret by some mishap. Perhaps they inadvertently deleted their cache or their cache was lost or disrupted due to restoring their disk from an earlier backup copy. The party that has the surviving cache entry can easily detect that a cache mismatch has occurred, because they expect their own cached secret to match the other party's cached secret, but it does not match. It is possible for both parties to detect this condition if both parties have surviving cached secrets that have fallen out of sync, due perhaps to one party restoring from a disk backup.

If either party discovers a cache mismatch, the user agent who makes this discovery must treat this as a possible security event and MUST alert their own user that there is a heightened risk of a MiTM attack, and that the user should verbally compare the SAS with the other party to ascertain that no MiTM attack has occurred. If a cache mismatch is detected and it is not possible to compare the SAS, either because the user interface does not support it or because one or both endpoints are unmanned devices, and no other SAS comparison mechanism is available, the session MAY be terminated.

The session need not be terminated on a cache mismatch event if:

- the mechanism described in **Section 8.1.1** is available, which allows authentication of the DH exchange without human assistance, or

- any mechanism is available to determine if the SAS matches. This would require either circumstances that allow human verbal comparisons of the SAS or by use of the OPTIONAL digital signature feature on the SAS hash, as described in **Section 7.2**.

Even if the user interface does not permit an SAS comparison, the human user MUST be warned and may elect to proceed with the call at their own risk.

If and only if a cache mismatch event occurs, the cache update mechanism in **Section 4.6.1** is affected, requiring the user to verify the SAS before the cache is updated. The user will thus be alerted of this security condition on every call until the SAS is verified. This is described in **Section 4.6.1.1**.

Here is a non-normative example of a cache-mismatch alert message from a ZRTP user agent (specifically, **[Zfone]**), designed for a desktop PC graphical user interface environment. It is by no means required that the alert be this detailed:

> We expected the other party to have a shared secret cached from a previous call, but they don't have it. This may mean your partner simply lost his cache of shared secrets, but it could also mean someone is trying to wiretap you. To resolve this question you must check the authentication string with your partner. If it doesn't match, it indicates the presence of a wiretapper.

If the alert is rendered by a robot voice instead of a GUI, brevity may be more important:

> Something's wrong. You must check the authentication string with your partner. If it doesn't match, it indicates the presence of a wiretapper.

A mismatch of auxsecret is handled differently than a mismatch of rs1. An auxsecret mismatch is defined to mean that auxsecret exists locally, but we computed a null value for s2 (per the method described in **Section 4.3**). This mismatch should be made visible to whichever user has auxsecret defined. The mismatch should be made visible to both users if they both have auxsecret defined but they fail to match. The severity of the user notification is implementation dependent. Aborting the session is not required. If auxsecret matches, it should not excuse a mismatch of rs1, which still requires a strong warning to the user.

TOC

4.4. DH and Non-DH Key Agreements

The next step is the generation of a secret for deriving SRTP keying material. ZRTP uses Diffie-Hellman and two non-Diffie-Hellman modes, described in the following subsections.

TOC

4.4.1. Diffie-Hellman Mode

The purpose of the Diffie-Hellman (either Finite Field Diffie-Hellman or Elliptic Curve Diffie-Hellman) exchange is for the two ZRTP endpoints to generate a new shared secret, s0. In addition, the endpoints discover if they have any cached or previously stored shared secrets in common, and it uses them as part of the calculation of the session keys.

Because the DH exchange affects the state of the retained shared secret cache, only one in-process ZRTP DH exchange may occur at a time between two ZRTP endpoints. Otherwise, race conditions and cache integrity problems will result. When multiple media streams are established in parallel between the same pair of ZRTP endpoints (determined by the ZIDs in the Hello messages), only one can be processed. Once that exchange completes with Confirm2 and Conf2ACK messages, another ZRTP DH exchange can begin. This constraint does not apply when Multistream mode key agreement is used since the cached shared secrets are not affected.

TOC

4.4.1.1. Hash Commitment in Diffie-Hellman Mode

From the intersection of the algorithms in the sent and received Hello messages, the initiator chooses a hash, cipher, auth tag, Key Agreement Type, and SAS Type to be used.

A Diffie-Hellman mode is selected by setting the Key Agreement Type in the Commit to one of the DH or Elliptic Curve Diffie-Hellman (ECDH) values from the table in **Section 5.1.5**. In this mode, the key agreement begins with the initiator choosing a fresh random Diffie-Hellman (DH) secret value (svi) based on the chosen Key Agreement Type value, and computing the public value. (Note that to speed up processing, this computation can be done in advance.) For guidance on generating random numbers, see **Section 4.8**.

For Finite Field Diffie-Hellman, the value for the DH generator g, the DH prime p, and the length of the DH secret value, svi, are defined in **Section 5.1.5**.

```
pvi = g^svi mod p
```

where g and p are determined by the Key Agreement Type value. The DH public value pvi value is formatted as a big-endian octet string and fixed to the bit-length of the DH prime; leading zeros MUST NOT be truncated.

For Elliptic Curve DH, pvi is calculated and formatted according to the ECDH specification in **Section 5.1.5**, which refers in detail to certain sections of NIST SP 800-56A **[NIST-SP800-56A]**.

The hash commitment is performed by the initiator of the ZRTP exchange. The hash value of the initiator, hvi, includes a hash of the entire DHPart2 message as shown in **Figure 9** (which includes the Diffie-Hellman public value, pvi), and the responder's Hello message (where '||' means concatenation). The hvi hash is truncated to 256 bits:

```
hvi = hash(initiator's DHPart2 message ||
           responder's Hello message)
```

Note that the Hello message includes the fields shown in **Figure 3**.

The information from the responder's Hello message is included in the hash calculation to prevent a bid-down attack by modification of the responder's Hello message.

The initiator sends the hvi in the Commit message.

The use of hash commitment in the DH exchange constrains the attacker to only one guess to generate the correct **Short Authentication String (SAS)** in his attack, which means the SAS can be quite short. A 16-bit SAS, for example, provides the attacker only one chance out of 65536 of not being detected. Without this hash commitment feature, a MiTM attacker would acquire both the pvi and pvr public values from the two parties before having to choose his own two DH public values for his MiTM attack. He could then use that information to quickly perform a bunch of trial DH calculations for both sides until he finds two with a matching SAS. To raise the cost of this birthday attack, the SAS would have to be much longer. The Short Authentication String would have to become a Long Authentication String, which would be unacceptable to the user. A hash commitment precludes this attack by forcing the MiTM to choose his own two DH public values before learning the public values of either of the two parties.

TOC

4.4.1.2. Responder Behavior in Diffie-Hellman Mode

Upon receipt of the Commit message, the responder generates its own fresh random DH secret value, svr, and computes the public value. (Note that to speed up processing, this computation can be done in advance, with no need to discard this computation if both endpoints chose the same algorithm via **Section 4.1.2**.) For guidance on random number generation, see **Section 4.8**.

For Finite Field Diffie-Hellman, the value for the DH generator g, the DH prime p, and the length of the DH secret value, svr, are defined in **Section 5.1.5**.

```
pvr = g^svr mod p
```

The pvr value is formatted as a big-endian octet string, fixed to the bit-length of the DH prime; leading zeros MUST NOT be truncated.

For Elliptic Curve DH, pvr is calculated and formatted according to the ECDH specification in **Section 5.1.5**, which refers in detail to certain sections of NIST SP 800-56A.

Upon receipt of the DHPart2 message, the responder checks that the initiator's DH public value is not equal to 1 or p-1. An attacker might inject a false DHPart2 message with a value of 1 or p-1 for g^svi mod p, which would cause a disastrously weak final DH result to be computed. If pvi is 1 or p-1, the user SHOULD be alerted of the attack and the protocol exchange MUST be terminated. Otherwise, the responder computes its own value for the hash commitment using the DH public value (pvi) received in the DHPart2 message and its own Hello message and compares the result with the hvi received in the Commit message. If they are different, a MiTM attack is taking place and the user is alerted and the protocol exchange terminated.

The responder then calculates the Diffie-Hellman result:

```
DHResult = pvi^svr mod p
```

TOC

4.4.1.3. Initiator Behavior in Diffie-Hellman Mode

Upon receipt of the DHPart1 message, the initiator checks that the responder's DH public value is not equal to 1 or p-1. An attacker might inject a false DHPart1 message with a value of 1 or p-1 for g^svr mod p, which would cause a disastrously weak final DH result to be computed. If pvr is 1 or p-1, the user should be alerted of the attack and the protocol exchange MUST be terminated.

The initiator then sends a DHPart2 message containing the initiator's DH public value and the set of calculated shared secret IDs as defined in **Section 4.3.1**.

The initiator calculates the same Diffie-Hellman result using:

```
DHResult = pvr^svi mod p
```

TOC

4.4.1.4. Shared Secret Calculation for DH Mode

A hash of the received and sent ZRTP messages in the current ZRTP exchange in the following order is calculated by both parties:

```
total_hash = hash(Hello of responder || Commit ||
                  DHPart1 || DHPart2)
```

Note that only the ZRTP messages (Figures **3**, **5**, **8**, and **9**), not the entire ZRTP packets, are included in the total_hash.

For both the initiator and responder, the DHResult is formatted as a big-endian octet string and fixed to the width of the DH prime; leading zeros MUST NOT be truncated. For example, for a 3072-bit p, DHResult would be a 384 octet value, with the first octet the most significant. DHResult may also be the result of an ECDH calculation, which is discussed in **Section 5.1.5**.

```
Key         | Size of
Agreement   | DHResult
------------------------
DH-3072     | 384 octets
```

```
------------------------
DH-2048    | 256 octets
------------------------
ECDH P-256 |  32 octets
------------------------
ECDH P-384 |  48 octets
------------------------
```

The authors believe the calculation of the final shared secret, s0, is in compliance with the recommendations in Sections 5.8.1 and 6.1.2.1 of **NIST SP 800-56A** [NIST-SP800-56A]. This is done by hashing a concatenation of a number of items, including the DHResult, the ZID's of the initiator (ZIDi) and the responder (ZIDr), the total_hash, and the set of non-null shared secrets as described in **Section 4.3**.

In Section 5.8.1 of **[NIST-SP800-56A]**, NIST requires certain parameters to be hashed together in a particular order, which NIST refers to as: Z, AlgorithmID, PartyUInfo, PartyVInfo, SuppPubInfo, and SuppPrivInfo. In our implementation, our DHResult corresponds to Z, "ZRTP-HMAC-KDF" corresponds to AlgorithmID, our ZIDi and ZIDr correspond to PartyUInfo and PartyVInfo, our total_hash corresponds to SuppPubInfo, and the set of three shared secrets s1, s2, and s3 corresponds to SuppPrivInfo. NIST also requires a 32-bit big-endian integer counter to be included in the hash each time the hash is computed, which we have set to the fixed value of 1 because we only compute the hash once. NIST refers to the final hash output as DerivedKeyingMaterial, which corresponds to our s0 in this calculation.

```
s0 = hash(counter || DHResult || "ZRTP-HMAC-KDF" ||
          ZIDi || ZIDr || total_hash || len(s1) || s1 ||
          len(s2) || s2 || len(s3) || s3)
```

Note that temporary values s1, s2, and s3 were calculated per the methods described in **Section 4.3**. DHResult, s1, s2, and s3 MUST all be erased from memory immediately after they are used to calculate s0.

The length of the DHResult field was implicitly agreed to by the negotiated DH prime size. The length of total_hash is implicitly determined by the negotiated hash algorithm. All of the explicit length fields, len(), in the above hash are 32-bit big-endian integers, giving the length in octets of the field that follows. Some members of the set of shared secrets (s1, s2, and s3) may have lengths of zero if they are null (not shared) and are each preceded by a 4-octet length field. For example, if s2 is null, len(s2) is 0x00000000, and s2 itself would be absent from the hash calculation, which means len(s3) would immediately follow len(s2). While inclusion of ZIDi and ZIDr may be redundant, because they are implicitly included in the total_hash, we explicitly include them here to follow NIST SP 800-56A. The fixed-length string "ZRTP-HMAC-KDF" (not null-terminated) identifies for what purpose the resulting s0 will be used, which is to serve as the key derivation key for the ZRTP HMAC-based key derivation function (KDF) defined in **Section 4.5.1** and used in **Section 4.5.3**.

The authors believe ZRTP DH mode is in full compliance with two relevant NIST documents that cover key derivations. First, Section 5.8.1 of **[NIST-SP800-56A]** computes what NIST refers to as DerivedKeyingMaterial, which ZRTP refers to as s0. This s0 then serves as the key derivation key, which NIST refers to as KI in the key derivation function described in Sections 5 and 5.1 of **[NIST-SP800-108]**, to derive all the rest of the subkeys needed by ZRTP. For ECDH mode, the authors believe the s0 calculation is also in compliance with Section 3.1 of the National Security Agency's (NSA's) **Suite B Implementer's Guide to NIST SP 800-56A** [NSA-Suite-B-Guide-56A].

The **ZRTP key derivation function (KDF)** requires the use of a KDF Context field (per **[NIST-SP800-108]** guidelines), which should include the ZIDi, ZIDr, and a nonce value known to both parties. The total_hash qualifies as a nonce value, because its computation included nonce material from the initiator's Commit message and the responder's Hello message.

```
KDF_Context = (ZIDi || ZIDr || total_hash)
```

At this point in DH mode, the two endpoints proceed to the key derivations of ZRTPSess and

the rest of the keys in **Section 4.5.2**, now that there is a defined s0.

TOC

4.4.2. Preshared Mode

The Preshared key agreement mode can be used to generate SRTP keys and salts without a DH calculation, instead relying on a shared secret from previous DH calculations between the endpoints.

This key agreement mode is useful to rapidly re-establish a secure session between two parties who have recently started and ended a secure session that has already performed a DH key agreement, without performing another lengthy DH calculation, which may be desirable on slow processors in resource-limited environments. Preshared mode MUST NOT be used for adding additional media streams to an existing call. Multistream mode MUST be used for this purpose.

In the most severe resource-limited environments, Preshared mode may be useful with processors that cannot perform a DH calculation in an ergonomically acceptable time limit. Shared key material may be manually provisioned between two such endpoints in advance and still allow a limited subset of functionality. Such a "better than nothing" implementation would have to be regarded as non-compliant with the ZRTP specification, but it could interoperate in Preshared (and if applicable, Multistream) mode with a compliant ZRTP endpoint.

Because Preshared mode affects the state of the retained shared secret cache, only one in-process ZRTP Preshared exchange may occur at a time between two ZRTP endpoints. This rule is explained in more detail in **Section 4.4.1**, and applies for the same reasons as in DH mode.

Preshared mode is only included in this specification to meet the R-REUSE requirement in the **Media Security Requirements** [RFC5479] document. A series of preshared-keyed calls between two ZRTP endpoints should use a DH key exchange periodically. Preshared mode is only used if a cached shared secret has been established in an earlier session by a DH exchange, as discussed in **Section 4.9**.

TOC

4.4.2.1. Commitment in Preshared Mode

Preshared mode is selected by setting the Key Agreement Type to Preshared in the Commit message. This results in the same call flow as Multistream mode. The principal difference between Multistream mode and Preshared mode is that Preshared mode uses a previously cached shared secret, rs1, instead of an active ZRTP Session key, ZRTPSess, as the initial keying material.

Preshared mode depends on having a reliable shared secret in its cache. Before Preshared mode is used, the initial DH exchange that gave rise to the shared secret SHOULD have used at least one of these anti-MiTM mechanisms: 1) A verbal comparison of the SAS, evidenced by the SAS Verified flag, or 2) an **end-to-end integrity-protected delivery of the a=zrtp-hash in the signaling**, or 3) a **digital signature on the sashash**.

TOC

4.4.2.2. Initiator Behavior in Preshared Mode

The Commit message (**Figure 7**) is sent by the initiator of the ZRTP exchange. From the intersection of the algorithms in the sent and received Hello messages, the initiator chooses a hash, cipher, auth tag, Key Agreement Type, and SAS Type to be used.

To assemble a Preshared commit, we must first construct a temporary preshared_key, which is constructed from one of several possible combinations of cached key material, depending on what is available in the shared secret cache. If rs1 is not available in the initiator's cache, then Preshared mode MUST NOT be used.

```
preshared_key = hash(len(rs1) || rs1 ||
                     len(auxsecret) || auxsecret ||
                     len(pbxsecret) || pbxsecret)
```

All of the explicit length fields, len(), in the above hash are 32-bit big-endian integers, giving the length in octets of the field that follows. Some members of the set of shared secrets (rs1, auxsecret, and pbxsecret) may have lengths of zero if they are null (not available), and are each preceded by a 4-octet length field. For example, if auxsecret is null, len(auxsecret) is 0x00000000, and auxsecret itself would be absent from the hash calculation, which means len(pbxsecret) would immediately follow len(auxsecret).

In place of hvi in the Commit message, two smaller fields are inserted by the initiator:

- A random nonce of length 4 words (16 octets).

- A keyID = MAC(preshared_key, "Prsh") truncated to 64 bits.

Note: Since the nonce is used to calculate different SRTP key and salt pairs for each session, a duplication will result in the same key and salt being generated for the two sessions, which would have disastrous security consequences.

TOC

4.4.2.3. Responder Behavior in Preshared Mode

The responder uses the received keyID to search for matching key material in its cache. It does this by computing a preshared_key value and keyID value using the same formula as the initiator, depending on what is available in the responder's local cache. If the locally computed keyID does not match the received keyID in the Commit, the responder recomputes a new preshared_key and keyID from a different subset of shared keys from the cache, dropping auxsecret, pbxsecret, or both from the hash calculation, until a matching preshared_key is found or it runs out of possibilities. Note that rs2 is not included in the process.

If it finds the appropriate matching shared key material, it is used to derive s0 and a new ZRTPSess key, as described in the next section on shared secret calculation, **Section 4.4.2.4**.

If the responder determines that it does not have a cached shared secret from a previous DH exchange, or it fails to match the keyID hash from the initiator with any combination of its shared keys, it SHOULD respond with its own DH Commit message. This would reverse the roles and the responder would become the initiator, because the DH Commit must always "trump" the Preshared Commit message as described in **Section 4.2**. The key exchange would then proceed using DH mode. However, if a severely resource-limited responder lacks the computing resources to respond in a reasonable time with a DH Commit, it MAY respond with a ZRTP Error message (**Section 5.9**) indicating that no shared secret is available.

If both sides send Preshared Commit messages initiating a secure session at the same time, the contention is resolved and the initiator/responder roles are settled according to **Section 4.2**, and the protocol proceeds.

In Preshared mode, both the DHPart1 and DHPart2 messages are skipped. After receiving the Commit message from the initiator, the responder sends the Confirm1 message after calculating this stream's SRTP keys, as described below.

TOC

4.4.2.4. Shared Secret Calculation for Preshared Mode

Preshared mode requires that the s0 and ZRTPSess keys be derived from the preshared_key, and this must be done in a way that guarantees uniqueness for each session. This is done by using nonce material from both parties: the explicit nonce in the initiator's **Preshared Commit message** and the H3 field in the responder's **Hello message**. Thus, both parties force the resulting shared secret to be unique for each session.

A hash of the received and sent ZRTP messages in the current ZRTP exchange for the

current media stream is calculated:

```
total_hash = hash(Hello of responder || Commit)
```

Note that only the ZRTP messages (Figures **3** and **7**), not the entire ZRTP packets, are included in the total_hash.

The **ZRTP key derivation function (KDF)** requires the use of a KDF Context field (per **[NIST-SP800-108]** guidelines), which should include the ZIDi, ZIDr, and a nonce value known to both parties. The total_hash qualifies as a nonce value, because its computation included nonce material from the initiator's Commit message and the responder's Hello message.

```
KDF_Context = (ZIDi || ZIDr || total_hash)
```

The s0 key is derived via the **ZRTP key derivation function** from preshared_key and the nonces implicitly included in the total_hash. The nonces also ensure KDF_Context is unique for each session, which is critical for security.

```
s0 = KDF(preshared_key, "ZRTP PSK",
         KDF_Context, negotiated hash length)
```

The preshared_key MUST be erased as soon as it has been used to calculate s0.

At this point in Preshared mode, the two endpoints proceed to the key derivations of ZRTPSess and the rest of the keys in **Section 4.5.2**, now that there is a defined s0.

TOC

4.4.3. Multistream Mode

The Multistream key agreement mode can be used to generate SRTP keys and salts for additional media streams established between a pair of endpoints. Multistream mode cannot be used unless there is an active SRTP session established between the endpoints, which means a ZRTP Session key is active. This ZRTP Session key can be used to generate keys and salts without performing another DH calculation. In this mode, the retained shared secret cache is not used or updated. As a result, multiple ZRTP Multistream mode exchanges can be processed in parallel between two endpoints.

Multistream mode is also used to resume a secure call that has gone clear using a GoClear message as described in **Section 4.7.2.1**.

When adding additional media streams to an existing call, Multistream mode MUST be used. The first media stream MUST use either DH mode or Preshared mode. Only one DH exchange or Preshared exchange is performed, just for the first media stream. The DH exchange or Preshared exchange MUST be completed for the first media stream before Multistream mode is used to add any other media streams. In a Multistream session, a ZRTP endpoint MUST use the same ZID for all media streams, matching the ZID used in the first media stream.

TOC

4.4.3.1. Commitment in Multistream Mode

Multistream mode is selected by the initiator setting the Key Agreement Type to "Mult" in the Commit message (**Figure 6**). The Cipher Type, Auth Tag Length, and Hash in Multistream mode SHOULD be set by the initiator to the same as the values as in the initial DH Mode Commit. The SAS Type is ignored as there is no SAS authentication in this mode.

> Note: This requirement is needed since some endpoints cannot support different SRTP algorithms for different media streams. However, in the case of

Multistream mode being used to go secure after a GoClear, the requirement to use the same SRTP algorithms is relaxed if there are no other active SRTP sessions.

In place of hvi in the Commit, a random nonce of length 4 words (16 octets) is chosen. Its value MUST be unique for all nonce values chosen for active ZRTP sessions between a pair of endpoints. If a Commit is received with a reused nonce value, the ZRTP exchange MUST be immediately terminated.

Note: Since the nonce is used to calculate different SRTP key and salt pairs for each media stream, a duplication will result in the same key and salt being generated for the two media streams, which would have disastrous security consequences.

If a Commit is received selecting Multistream mode, but the responder does not have a ZRTP Session Key available, the exchange MUST be terminated. Otherwise, the responder proceeds to the next section on shared secret calculation, **Section 4.4.3.2**.

If both sides send Multistream Commit messages at the same time, the contention is resolved and the initiator/responder roles are settled according to **Section 4.2**, and the protocol proceeds.

In Multistream mode, both the DHPart1 and DHPart2 messages are skipped. After receiving the Commit message from the initiator, the responder sends the Confirm1 message after calculating this stream's SRTP keys, as described below.

TOC

4.4.3.2. Shared Secret Calculation for Multistream Mode

In Multistream mode, each media stream requires that a set of keys be derived from the ZRTPSess key, and this must be done in a way that guarantees uniqueness for each media stream. This is done by using nonce material from both parties: the explicit nonce in the initiator's **Multistream Commit message** and the H3 field in the responder's **Hello message**. Thus, both parties force the resulting shared secret to be unique for each media stream.

A hash of the received and sent ZRTP messages in the current ZRTP exchange for the current media stream is calculated:

```
total_hash = hash(Hello of responder || Commit)
```

This refers to the Hello and Commit messages for the current media stream, which is using Multistream mode, not the original media stream that included a full DH key agreement. Note that only the ZRTP messages (Figures **3** and **6**), not the entire ZRTP packets, are included in the hash.

The **ZRTP key derivation function (KDF)** requires the use of a KDF Context field (per **[NIST-SP800-108]** guidelines), which should include the ZIDi, ZIDr, and a nonce value known to both parties. The total_hash qualifies as a nonce value, because its computation included nonce material from the initiator's Commit message and the responder's Hello message.

```
KDF_Context = (ZIDi || ZIDr || total_hash)
```

The current stream's SRTP keys and salts for the initiator and responder are calculated using the ZRTP Session Key ZRTPSess and the nonces implicitly included in the total_hash. The nonces also ensure that KDF_Context will be unique for each media stream, which is critical for security. For each additional media stream, a separate s0 is derived from ZRTPSess via the **ZRTP key derivation function**:

```
s0 = KDF(ZRTPSess, "ZRTP MSK",
         KDF_Context, negotiated hash length)
```

Note that the ZRTPSess key was previously derived from material that also includes a different and more inclusive total_hash from the entire packet sequence that performed the original DH exchange for the first media stream in this ZRTP session.

At this point in Multistream mode, the two endpoints begin key derivations in **Section 4.5.3**.

4.5. Key Derivations

TOC

4.5.1. The ZRTP Key Derivation Function

TOC

To derive keys from a shared secret, ZRTP uses an HMAC-based key derivation function, or KDF. It is used throughout **Section 4.5.3** and in other sections. The HMAC function for the KDF is based on the negotiated hash algorithm defined in **Section 5.1.2**.

The authors believe the ZRTP KDF is in full compliance with the recommendations in **NIST SP 800-108** [NIST-SP800-108]. Section 7.5 of the NIST document describes "key separation", which is a security requirement for the cryptographic keys derived from the same key derivation key. The keys shall be separate in the sense that the compromise of some derived keys will not degrade the security strength of any of the other derived keys or the security strength of the key derivation key. Strong preimage resistance is provided.

The ZRTP KDF runs the NIST pseudorandom function (PRF) in counter mode, with only a single iteration of the counter. The NIST PRF is based on the HMAC function. The ZRTP KDF never has to generate more than 256 bits (or 384 bits for Suite B applications) of output key material, so only a single invocation of the HMAC function is needed.

The ZRTP KDF is defined in this manner, per Sections 5 and 5.1 of **[NIST-SP800-108]**:

```
KDF(KI, Label, Context, L) =
  HMAC(KI, i || Label || 0x00 || Context || L)
```

The HMAC in the KDF is keyed by KI, which is a secret key derivation key that is unknown to the wiretapper (for example, s0). The HMAC is computed on a concatenated set of nonsecret fields that are defined as follows. The first field is a 32-bit big-endian integer counter (i) required by NIST to be included in the HMAC each time the HMAC is computed, which we have set to the fixed value of 0x000001 because we only compute the HMAC once. Label is a string of nonzero octets that identifies the purpose for the derived keying material. The octet 0x00 is a delimiter required by NIST. The NIST KDF formula has a "Context" field that includes ZIDi, ZIDr, and some optional nonce material known to both parties. L is a 32-bit big-endian positive integer, not to exceed the length in bits of the output of the HMAC. The output of the KDF is truncated to the leftmost L bits. If SHA-384 is the negotiated hash algorithm, the HMAC would be HMAC-SHA-384; thus, the maximum value of L would be 384, the negotiated hash length.

The ZRTP KDF is not to be confused with the SRTP KDF defined in **[RFC3711]**.

4.5.2. Deriving ZRTPSess Key and SAS in DH or Preshared Modes

TOC

Both DH mode and Preshared mode (but not Multistream mode) come to this common point in the protocol to derive ZRTPSess and the SAS from s0, via the **ZRTP Key Derivation Function**. At this point, s0 has been calculated, as well as KDF_Context. These calculations are done only for the first media stream, not for Multistream mode.

The ZRTPSess key is used only for these two purposes: 1) to **generate the additional s0 keys** for adding additional media streams to this session in Multistream mode, and 2) to **generate the pbxsecret** that may be cached for use in future sessions. The ZRTPSess key is kept for the duration of the call signaling session between the two ZRTP endpoints. That is,

if there are two separate calls between the endpoints (in SIP terms, separate SIP dialogs), then a ZRTP Session Key MUST NOT be used across the two call signaling sessions. ZRTPSess MUST be destroyed no later than the end of the call signaling session.

```
ZRTPSess = KDF(s0, "ZRTP Session Key",
               KDF_Context, negotiated hash length)
```

Note that KDF_Context is unique for each media stream, but only the first media stream is permitted to calculate ZRTPSess.

There is only one **Short Authentication String (SAS)** computed per call, which is applicable to all media streams derived from a single DH key agreement in a ZRTP session. KDF_Context is unique for each media stream, but only the first media stream is permitted to calculate sashash.

```
sashash = KDF(s0, "SAS", KDF_Context, 256)

sasvalue = sashash [truncated to leftmost 32 bits]
```

Despite the exposure of the SAS to the two parties, the rest of the keying material is protected by the **key separation properties of the KDF**.

ZRTP-enabled VoIP clients may need to support additional forms of communication, such as text chat, instant messaging, or file transfers. These other forms of communication may need to be encrypted, and would benefit from leveraging the ZRTP key exchange used for the VoIP part of the call. In that case, more key material MAY be derived and "exported" from the ZRTP protocol and provided as a shared secret to the VoIP client for these non-VoIP purposes. The application can use this exported key in application-specific ways, outside the scope of the ZRTP protocol.

```
ExportedKey = KDF(s0, "Exported key",
                  KDF_Context, negotiated hash length)
```

Only one ExportedKey is computed per call. KDF_Context is unique for each media stream, but only the first media stream is permitted to calculate ExportedKey.

The application may use this exported key to derive other subkeys for various non-ZRTP purposes, via a KDF using separate KDF label strings defined by the application. This key or its derived subkeys can be used for encryption, or used to authenticate other key exchanges carried out by the application, protected by ZRTP's MiTM defense umbrella. The exported key and its descendants may be used for as long as needed by the application, maintained in a separate crypto context that may outlast the VoIP session.

At this point in DH mode or Preshared mode, the two endpoints proceed on to the key derivations in **Section 4.5.3**, now that there is a defined s0 and ZRTPSess key.

TOC

4.5.3. Deriving the Rest of the Keys from s0

DH mode, Multistream mode, and Preshared mode all come to this common point in the protocol to derive a set of keys from s0. It can be assumed that s0 has been calculated, as well the ZRTPSess key and KDF_Context. A separate s0 key is associated with each media stream.

Subkeys are not drawn directly from s0, as done in NIST SP 800-56A. To enhance key separation, ZRTP uses s0 to key a **Key Derivation Function** based on **[NIST-SP800-108]**. Since s0 already included total_hash in its derivation, it is redundant to use total_hash again in the KDF Context in all the invocations of the KDF keyed by s0. Nonetheless, NIST SP 800-108 always requires KDF Context to be defined for the KDF, and nonce material is required in some KDF invocations (especially for Multistream mode and Preshared mode), so total_hash is included as a nonce in the KDF Context.

Separate SRTP master keys and master salts are derived for use in each direction for each media stream. Unless otherwise specified, ZRTP uses SRTP with no Master Key Identifier (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. Secure RTCP (SRTCP) is also used, deriving the SRTCP keys from the same master keys and salts as SRTP, using the mechanisms specified in **[RFC3711]**, without requiring a separate ZRTP negotiation for RTCP.

The ZRTP initiator encrypts and the ZRTP responder decrypts packets by using srtpkeyi and srtpsalti, while the ZRTP responder encrypts and the ZRTP initiator decrypts packets by using srtpkeyr and srtpsaltr. The SRTP key and salt values are truncated (taking the leftmost bits) to the length determined by the chosen SRTP profile. These are generated by:

```
srtpkeyi = KDF(s0, "Initiator SRTP master key",
               KDF_Context, negotiated AES key length)

srtpsalti = KDF(s0, "Initiator SRTP master salt",
                KDF_Context, 112)

srtpkeyr = KDF(s0, "Responder SRTP master key",
               KDF_Context, negotiated AES key length)

srtpsaltr = KDF(s0, "Responder SRTP master salt",
                KDF_Context, 112)
```

The MAC keys are the same length as the output of the underlying hash function in the KDF and are thus generated without truncation. They are used only by ZRTP and not by SRTP. Different MAC keys are needed for the initiator and the responder to ensure that GoClear messages in each direction are unique and can not be cached by an attacker and reflected back to the endpoint.

```
mackeyi = KDF(s0, "Initiator HMAC key",
              KDF_Context, negotiated hash length)

mackeyr = KDF(s0, "Responder HMAC key",
              KDF_Context, negotiated hash length)
```

ZRTP keys are generated for the initiator and responder to use to encrypt the Confirm1 and Confirm2 messages. They are truncated to the same size as the negotiated SRTP key size.

```
zrtpkeyi = KDF(s0, "Initiator ZRTP key",
               KDF_Context, negotiated AES key length)

zrtpkeyr = KDF(s0, "Responder ZRTP key",
               KDF_Context, negotiated AES key length)
```

All key material is destroyed as soon as it is no longer needed, no later than the end of the call. s0 is erased in **Section 4.6.1**, and the rest of the session key material is erased in Sections **4.7.2.1** and **4.7.3**.

TOC

4.6. Confirmation

The Confirm1 and Confirm2 messages (**Figure 10**) contain the cache expiration interval (defined in **Section 4.9**) for the newly generated retained shared secret. The flagoctet is an 8-bit unsigned integer made up of these flags: the PBX Enrollment flag (E) defined in **Section 7.3.1**, the SAS Verified flag (V) defined in **Section 7.1**, the Allow Clear flag (A) defined in **Section 4.7.2**, and the Disclosure flag (D) defined in **Section 11**.

```
flagoctet =  (E * 2^3) + (V * 2^2) + (A * 2^1) + (D * 2^0)
```

Part of the Confirm1 and Confirm2 messages are encrypted using full-block Cipher Feedback Mode and contain a 128-bit random Cipher FeedBack (CFB) Initialization Vector (IV). The Confirm1 and Confirm2 messages also contain a MAC covering the encrypted part of the Confirm1 or Confirm2 message that includes a string of zeros, the signature length, flag octet, cache expiration interval, signature type block (if present), and **signature** (if present). For the responder:

```
confirm_mac = MAC(mackeyr, encrypted part of Confirm1)
```

For the initiator:

```
confirm_mac = MAC(mackeyi, encrypted part of Confirm2)
```

The mackeyi and mackeyr keys are computed in **Section 4.5.3**.

The exchange is completed when the responder sends either the Conf2ACK message or the responder's first SRTP media packet (with a valid SRTP auth tag). The initiator MUST treat the first valid SRTP media from the responder as equivalent to receiving a Conf2ACK. The responder may respond to Confirm2 with either SRTP media, Conf2ACK, or both, in whichever order the responder chooses (or whichever order the "cloud" chooses to deliver them).

TOC

4.6.1. Updating the Cache of Shared Secrets

After receiving the Confirm messages, both parties must now update their retained shared secret rs1 in their respective caches, provided the following conditions hold:

(1) This key exchange is either DH or Preshared mode, not Multistream mode, which does not update the cache.

(2) Depending on the values of the cache expiration intervals that are received in the two Confirm messages, there are some scenarios that do not update the cache, as explained in **Section 4.9**.

(3) The responder MUST receive the initiator's Confirm2 message before updating the responder's cache.

(4) The initiator MUST receive either the responder's Conf2ACK message or the responder's SRTP media (with a valid SRTP auth tag) before updating the initiator's cache.

The cache update may also be affected by a cache mismatch, according to **Section 4.6.1.1**.

For DH mode only, before updating the retained shared secret rs1 in the cache, each party first discards their old rs2 and copies their old rs1 to rs2. The old rs1 is saved to rs2 because of the risk of session interruption after one party has updated his own rs1 but before the other party has enough information to update her own rs1. If that happens, they may regain cache sync in the next session by using rs2 (per **Section 4.3**). This mitigates the well-known **Two Generals' Problem** [Byzantine]. The old rs1 value is not saved in Preshared mode.

For DH mode and Preshared mode, both parties compute a new rs1 value from s0 via the **ZRTP key derivation function**:

```
rs1 = KDF(s0, "retained secret", KDF_Context, 256)
```

Note that KDF_Context is unique for each media stream, but only the first media stream is

permitted to update rs1.

Each media stream has its own s0. At this point in the protocol for each media stream, the corresponding s0 MUST be erased.

TOC

4.6.1.1. Cache Update Following a Cache Mismatch

If a shared secret cache mismatch (as defined in **Section 4.3.2**) is detected in the current session, it indicates a possible MiTM attack. However, there may be evidence to the contrary, if either one of the following conditions are met:

- Successful use of the mechanism described in **Section 8.1.1**, but only if fully supported by end-to-end integrity-protected delivery of the a=zrtp-hash in the signaling via **SIP Identity** [RFC4474] or better still, Dan Wing's **SIP Identity using Media Path** [SIP-IDENTITY]. This allows authentication of the DH exchange without human assistance.
- A good signature is received and verified using the digital signature feature on the SAS hash, as described in **Section 7.2**, if this feature is supported.

If there is a cache mismatch in the absence of the aforementioned mitigating evidence, the cache update MUST be delayed in the current session until the user verbally compares the SAS with his partner during the call and confirms a successful SAS verify via his user interface as described in **Section 7.1**. If the session ends before that happens, the cache update is not performed, leaving the rs1/rs2 values unmodified in the cache. Regardless of whether a cache mismatch occurs, s0 must still be erased.

If no cache entry exists, as is the case in the initial call, the cache update is handled in the normal fashion.

TOC

4.7. Termination

A ZRTP session is normally terminated at the end of a call, but it may be terminated early by either the Error message or the GoClear message.

TOC

4.7.1. Termination via Error Message

The Error message (**Section 5.9**) is used to terminate an in-progress ZRTP exchange due to an error. The Error message contains an integer Error Code for debugging purposes. The termination of a ZRTP key agreement exchange results in no updates to the cached shared secrets and deletion of all crypto context for that media stream. The ZRTP Session key, ZRTPSess, is only deleted if all ZRTP media streams that are using it are terminated.

Because no key agreement has been reached, the Error message cannot use the same MAC protection as the GoClear message. A denial of service is possible by injecting fake Error messages. (However, even if the Error message were somehow designed with integrity protection, it would raise other questions. What would a badly formed Error message mean if it were sent to report a badly formed message? A good message?)

TOC

4.7.2. Termination via GoClear Message

The GoClear message (**Section 5.11**) is used to switch from SRTP to RTP, usually because the user has chosen to do that by pressing a button. The GoClear uses a MAC of the Message Type Block sent in the GoClear message computed with the mackey derived from the shared secret. This MAC is truncated to the leftmost 64 bits. When sent by the initiator:

```
clear_mac = MAC(mackeyi, "GoClear ")
```

When sent by the responder:

```
clear_mac = MAC(mackeyr, "GoClear ")
```

Both of these MACs are calculated across the 8-octet "GoClear " Message Type Block, including the trailing space.

A GoClear message that does not receive a ClearACK response must be resent. If a GoClear message is received with a bad MAC, ClearACK MUST NOT be sent and the GoClear MUST NOT be acted on by the recipient, but it MAY be processed as a security exception, perhaps by logging or alerting the user.

A ZRTP endpoint MAY choose to accept GoClear messages after the session has switched to SRTP, allowing the session to revert to RTP. This is indicated in the Confirm1 or Confirm2 messages (**Figure 10**) by setting the Allow Clear flag (A). If an endpoint sets the Allow Clear (A) flag in their Confirm message, it indicates that they support receiving GoClear messages.

A ZRTP endpoint that receives a GoClear MUST authenticate the message by checking the clear_mac. If the message authenticates, the endpoint stops sending SRTP packets, and generates a ClearACK in response. It MUST also delete all the crypto key material for all the SRTP media streams, as defined in **Section 4.7.2.1**.

Until confirmation from the user is received (e.g., clicking a button, pressing a dual-tone multi-frequency (DTMF) key, etc.), the ZRTP endpoint MUST NOT resume sending RTP packets. The endpoint then renders to the user an indication that the media session has switched to clear mode and waits for confirmation from the user. This blocks the flow of sensitive discourse until the user is forced to take notice that he's no longer protected by encryption. To prevent pinholes from closing or NAT bindings from expiring, the ClearACK message MAY be resent at regular intervals (e.g., every 5 seconds) while waiting for confirmation from the user. After confirmation of the notification is received from the user, the sending of RTP packets may begin.

After sending a GoClear message, the ZRTP endpoint stops sending SRTP packets. When a ClearACK is received, the ZRTP endpoint deletes the crypto context for the SRTP session, as defined in **Section 4.7.2.1**, and may then resume sending RTP packets.

In the event a ClearACK is not received before the retransmissions of GoClear are exhausted, the key material is deleted, as defined in **Section 4.7.2.1**.

After the users have transitioned from SRTP media back to RTP media (clear mode), they may decide later to return to secure mode by manual activation, usually by pressing a GO SECURE button. In that case, a new secure session is initiated by the party that presses the button, by sending a new Commit message, leading to a new session key negotiation. It is not necessary to send another Hello message, as the two parties have already done that at the start of the call and thus have already discovered each other's ZRTP capabilities. It is possible for users to toggle back and forth between clear and secure modes multiple times in the same session, just as they could in the old days of secure PSTN phones.

TOC

4.7.2.1. Key Destruction for GoClear Message

All SRTP session key material MUST be erased by the receiver of the GoClear message upon receiving a properly authenticated GoClear. The same key destruction MUST be done by the sender of GoClear message, upon receiving the ClearACK. This must be done for the key material for all of the media streams.

All key material that would have been erased at the end of the SIP session MUST be erased, as described in **Section 4.7.3**, with the single exception of ZRTPSess. In this case, ZRTPSess is destroyed in a manner different from the other key material. Both parties replace ZRTPSess with a KDF-derived non-invertible function of itself:

```
ZRTPSess = KDF(ZRTPSess, "New ZRTP Session",
               (ZIDi || ZIDr), negotiated hash length)
```

ZRTPSess will be replaced twice if a session generates separate GoClear messages for both audio and video streams, and the two endpoints need not carry out the replacements in the same order.

The destruction of key material meets the requirements of Perfect Forward Secrecy (PFS), but still preserves a new version of ZRTPSess, so that the user can later re-initiate secure mode during the same session without performing another Diffie-Hellman calculation using Multistream mode, which requires and assumes the existence of ZRTPSess with the same value at both ZRTP endpoints. A new key negotiation after a GoClear SHOULD use a Multistream Commit message.

> Note: Multistream mode is preferred over a Diffie-Hellman mode since this does not require the generation of a new hash chain and a new signaling exchange to exchange new Hello Hash values.

Later, at the end of the entire call, ZRTPSess is finally destroyed along with the other key material, as described in **Section 4.7.3**.

TOC

4.7.3. Key Destruction at Termination

All SRTP session key material MUST be erased by both parties at the end of the call. In particular, the destroyed key material includes the SRTP session keys and salts, SRTP master keys and salts, and all material sufficient to reconstruct the SRTP keys and salts, including ZRTPSess and s0 (although s0 should have been destroyed earlier, in **Section 4.6.1**). This must be done for the key material for all of the media streams. The only exceptions are the cached shared secrets needed for future sessions, including rs1, rs2, and pbxsecret.

TOC

4.8. Random Number Generation

The ZRTP protocol uses random numbers for cryptographic key material, notably for the DH secret exponents and nonces, which must be freshly generated with each session. Whenever a random number is needed, all of the following criteria must be satisfied:

Random numbers MUST be freshly generated, meaning that they must not have been used in a previous calculation.

When generating a random number k of L bits in length, k MUST be chosen with equal probability from the range of [1 < k < 2^L].

It MUST be derived from a physical entropy source, such as radio frequency (RF) noise, acoustic noise, thermal noise, high-resolution timings of environmental events, or other unpredictable physical sources of entropy. One possible source of entropy for a VoIP client would be microphone noise. For a detailed explanation of cryptographic grade random numbers and guidance for collecting suitable entropy, see **[RFC4086]** and Chapter 10 of **"Practical Cryptography"** [Ferguson]. The raw entropy must be distilled and processed through a deterministic random-bit generator (DRBG). Examples of DRBGs may be found in **[NIST-SP800-90]**, in **[Ferguson]**, and in **[RFC5869]**. Failure to use true entropy from the physical environment as a basis for generating random cryptographic key material would lead to a disastrous loss of security.

TOC

4.9. ZID and Cache Operation

Each instance of ZRTP has a unique 96-bit random ZRTP ID, or ZID, that is generated once at installation time. It is used to look up retained shared secrets in a local cache. A single global ZID for a single installation is the simplest way to implement ZIDs. However, it is specifically not precluded for an implementation to use multiple ZIDs, up to the limit of a separate one per callee. This then turns it into a long-lived "association ID" that does not apply to any other associations between a different pair of parties. It is a goal of this protocol to permit both options to interoperate freely. A PBX acting as a trusted man in the middle will also generate

a single ZID and use that ZID for all endpoints behind it, as described in **Section 10**.

There is no protocol mechanism to invalidate a previously used ZID. An endpoint wishing to change ZIDs would simply generate a new one and begin using it.

The ZID should not be hard coded or hard defined in the firmware of a product. It should be randomly generated by the software and stored at installation or initialization time. It should be randomly generated rather than allocated from a preassigned range of ZID values, because 96 bits should be enough to avoid birthday collisions in realistic scenarios.

Each time a new s0 is calculated, a new retained shared secret rs1 is generated and stored in the cache, indexed by the ZID of the other endpoint. This cache updating is described in **Section 4.6.1**. For the new retained shared secret, each endpoint chooses a cache expiration value that is an unsigned 32-bit integer of the number of seconds that this secret should be retained in the cache. The time interval is relative to when the Confirm1 message is sent or received.

The cache intervals are exchanged in the Confirm1 and Confirm2 messages (**Figure 10**). The actual cache interval used by both endpoints is the minimum of the values from the Confirm1 and Confirm2 messages. A value of 0 seconds means the newly computed shared secret SHOULD NOT be stored in the cache, and if a cache entry already exists from an earlier call, the stored cache interval should be set to 0. This means if either Confirm message contains a null cache expiration interval, and there is no cache entry already defined, no new cache entry is created. A value of 0xffffffff means the secret should be cached indefinitely and is the recommended value. If the ZRTP exchange is Multistream mode, the field in the Confirm1 and Confirm2 is set to 0xffffffff and is ignored; the cache is not updated.

The expiration interval need not be used to force the deletion of a shared secret from the cache when the interval has expired. It just means the shared secret MAY be deleted from that cache at any point after the interval has expired without causing the other party to note it as an unexpected security event when the next key negotiation occurs between the same two parties. This means there need not be perfectly synchronized deletion of expired secrets from the two caches, and makes it easy to avoid a race condition that might otherwise be caused by clock skew.

If the expiration interval is not properly agreed to by both endpoints, it may later result in false alarms of MiTM attacks, due to apparent **cache mismatches**.

The relationship between a ZID and a SIP AOR is explained in **Section 12**.

TOC

4.9.1. Cacheless Implementations

It is possible to implement a simplified but nonetheless useful (and still compliant) profile of the ZRTP protocol that does not support any caching of shared secrets. In this case, the users would have to rely exclusively on the verbal SAS comparison for every call. That is, unless MiTM protection is provided by the mechanisms in Section **8.1.1** or **7.2**, which introduce their own forms of complexity.

If a ZRTP endpoint does not support the caching of shared secrets, it MUST set the cache expiration interval to zero, and MUST set the **SAS Verified (V) flag** to false. In addition, because the ZID serves mainly as a cache index, the ZID would not be required to maintain the same value across separate SIP sessions, although there is no reason why it should not.

Cacheless operation would sacrifice the **key continuity** features, as well as **Preshared mode**. Further, if the pbxsecret is also not cached, there would be no **PBX trusted MiTM** features, including the **PBX security enrollment** mechanism.

TOC

5. ZRTP Messages

All ZRTP messages use the message format defined in **Figure 2**. All word lengths referenced in this specification are 32 bits, or 4 octets. All integer fields are carried in network byte order, that is, most-significant byte (octet) first, commonly known as big-endian.

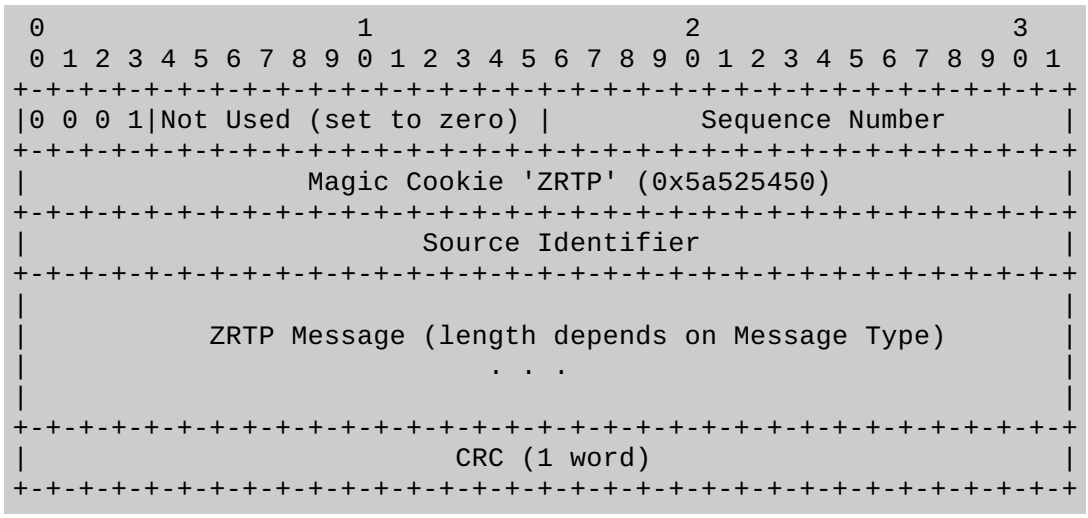


Figure 2: ZRTP Packet Format

The Sequence Number is a count that is incremented for each ZRTP packet sent. The count is initialized to a random value. This is useful in estimating ZRTP packet loss and also detecting when ZRTP packets arrive out of sequence.

The ZRTP Magic Cookie is a 32-bit string that uniquely identifies a ZRTP packet and has the value 0x5a525450.

Source Identifier is the SSRC number of the RTP stream to which this ZRTP packet relates. For cases of forking or forwarding, RTP, and hence ZRTP, may arrive at the same port from several different sources -- each of these sources will have a different SSRC and may initiate an independent ZRTP protocol session. SSRC collisions would be disruptive to ZRTP. SSRC collision handling procedures are described in **Section 4.1**.

This format is clearly identifiable as non-RTP due to the first two bits being zero, which looks like RTP version 0, which is not a valid RTP version number. It is clearly distinguishable from STUN since the Magic Cookies are different. The 12 unused bits are set to zero and MUST be ignored when received. In early versions of this spec, ZRTP messages were encapsulated in RTP header extensions, which made ZRTP an eponymous variant of RTP. In later versions, the packet format changed to make it syntactically distinguishable from RTP.

The ZRTP messages are defined in Figures **3** to **17** and are of variable length.

The ZRTP protocol uses a 32-bit Cyclic Redundancy Check (CRC) as defined in RFC 4960, Appendix B **[RFC4960]**, in each ZRTP packet to detect transmission errors. ZRTP packets are typically transported by UDP, which carries its own built-in 16-bit checksum for integrity, but ZRTP does not rely on it. This is because of the effect of an undetected transmission error in a ZRTP message. For example, an undetected error in the DH exchange could appear to be an active man-in-the-middle attack. A false announcement of this by ZRTP clients can be psychologically distressing. The probability of such a false alarm hinges on a mere 16-bit checksum that usually protects UDP packets, so more error detection is needed. For these reasons, this belt-and-suspenders approach is used to minimize the chance of a transmission error affecting the ZRTP key agreement.

The CRC is calculated across the entire ZRTP packet shown in **Figure 2**, including the ZRTP header and the ZRTP message, but not including the CRC field. If a ZRTP message fails the CRC check, it is silently discarded.

5.1. ZRTP Message Formats

TOC

ZRTP messages are designed to simplify endpoint parsing requirements and to reduce the opportunities for buffer overflow attacks (a good goal of any security extension should be to not introduce new attack vectors).

ZRTP uses a block of 8 octets (2 words) to encode the Message Type. 4-octet (1 word) blocks are used to encode Hash Type, Cipher Type, Key Agreement Type, and Authentication Tag Type. The values in the blocks are ASCII strings that are extended with spaces (0x20) to make them the desired length. Currently defined block values are listed in Tables 1-6.

Additional block values may be defined and used.

ZRTP uses this ASCII encoding to simplify debugging and make it "Wireshark (Ethereal) friendly".

TOC

5.1.1. Message Type Block

Currently, 16 Message Type Blocks are defined -- they represent the set of ZRTP message primitives. ZRTP endpoints MUST support the Hello, HelloACK, Commit, DHPart1, DHPart2, Confirm1, Confirm2, Conf2ACK, SASrelay, RelayACK, Error, ErrorACK, and PingACK message types. ZRTP endpoints MAY support the GoClear, ClearACK, and Ping messages. In order to generate a PingACK message, it is necessary to parse a Ping message. Additional messages may be defined in extensions to ZRTP.

Message Type Block	Meaning
"Hello "	Hello Message
"HelloACK"	HelloACK Message
"Commit "	Commit Message
"DHPart1 "	DHPart1 Message
"DHPart2 "	DHPart2 Message
"Confirm1"	Confirm1 Message
"Confirm2"	Confirm2 Message
"Conf2ACK"	Conf2ACK Message
"Error "	Error Message
"ErrorACK"	ErrorACK Message
"GoClear "	GoClear Message
"ClearACK"	ClearACK Message
"SASrelay"	SASrelay Message
"RelayACK"	RelayACK Message
"Ping "	Ping Message
"PingACK "	PingACK Message

Table 1. Message Type Block Values

TOC

5.1.2. Hash Type Block

The hash algorithm and its related MAC algorithm are negotiated via the Hash Type Block found in the **Hello message** and the **Commit message**.

All ZRTP endpoints MUST support a Hash Type of SHA-256 **[FIPS-180-3]**. SHA-384 SHOULD be supported and MUST be supported if ECDH-384 is used. Additional Hash Types MAY be used, such as the **NIST SHA-3 hash** [SHA-3] when it becomes available. Note that the Hash Type refers to the hash algorithm that will be used throughout the ZRTP key exchange, not the hash algorithm to be used in the SRTP Authentication Tag.

The choice of the negotiated Hash Type is coupled to the Key Agreement Type, as explained in **Section 5.1.5**.

```
Hash Type Block | Meaning
-----------------------------------------------------------
"S256"          | SHA-256 Hash defined in FIPS 180-3
-----------------------------------------------------------
"S384"          | SHA-384 Hash defined in FIPS 180-3
-----------------------------------------------------------
"N256"          | NIST SHA-3 256-bit hash (when published)
-----------------------------------------------------------
"N384"          | NIST SHA-3 384-bit hash (when published)
-----------------------------------------------------------

Table 2. Hash Type Block Values
```

At the time of this writing, the **NIST SHA-3 hashes** [SHA-3] are not yet available. NIST is expected to publish SHA-3 in 2012, as a successor to the SHA-2 hashes in **[FIPS-180-3]**.

TOC

5.1.2.1. Negotiated Hash and MAC Algorithm

ZRTP makes use of message authentication codes (MACs) that are keyed hashes based on the negotiated Hash Type. For the SHA-2 and SHA-3 hashes, the negotiated MAC is the HMAC based on the negotiated hash. This MAC function is also used in the **ZRTP key derivation function**.

The HMAC function is defined in **[FIPS-198-1]**. A discussion of the general security of the HMAC construction may be found in **[RFC2104]**. Test vectors for HMAC-SHA-256 and HMAC-SHA-384 may be found in **[RFC4231]**.

The negotiated Hash Type does not apply to the hash used in the digital signature defined in **Section 7.2**. For example, even if the negotiated Hash Type is SHA-256, the digital signature may use SHA-384 if an Elliptic Curve Digital Signature Algorithm (ECDSA) P-384 signature key is used. Digital signatures are optional in ZRTP.

Except for the aforementioned digital signatures, and the special cases noted in **Section 5.1.2.2**, all the other hashes and MACs used throughout the ZRTP protocol will use the negotiated Hash Type.

A future hash may include its own built-in MAC, not based on the HMAC construct, for example, the **Skein hash function** [Skein]. If NIST chooses such a hash as the SHA-3 winner, Hash Types "N256", and "N384" will still use the related HMAC as the negotiated MAC. If an implementer wishes to use Skein and its built-in MAC as the negotiated MAC, new Hash Types must be used.

TOC

5.1.2.2. Implicit Hash and MAC Algorithm

While most of the hash and MAC usage in ZRTP is defined by the negotiated **Hash Type**, some hashes and MACs must be precomputed prior to negotiations, and thus cannot have their algorithms negotiated during the ZRTP exchange. They are implicitly predetermined to use SHA-256 **[FIPS-180-3]** and HMAC-SHA-256.

These are the hashes and MACs that MUST use the Implicit hash and MAC algorithm:

- The hash chain H0-H3 defined in **Section 9**.

The MACs that are keyed by this hash chain, as defined in **Section 8.1.1**.

The Hello Hash in the a=zrtp-hash attribute defined in **Section 8.1**.

ZRTP defines a method for **negotiating different ZRTP protocol versions**. SHA-256 is the Implicit Hash and HMAC-SHA-256 is the Implicit MAC for ZRTP protocol version 1.10. Future ZRTP protocol versions may, if appropriate, use another hash algorithm as the Implicit Hash, such as the **NIST SHA-3 hash** [SHA-3], when it becomes available. For example, a future SIP packet may list two a=zrtp-hash SDP attributes, one based on SHA-256 for ZRTP version 1.10, and another based on SHA-3 for ZRTP version 2.00.

TOC

5.1.3. Cipher Type Block

The block cipher algorithm is negotiated via the Cipher Type Block found in the **Hello message** and the **Commit message**.

All ZRTP endpoints MUST support AES-128 (AES1) and MAY support AES-192 (AES2), AES-256 (AES3), or other Cipher Types. The Advanced Encryption Standard is defined in **[FIPS-197]**.

The use of AES-128 in SRTP is defined by **[RFC3711]**. The use of AES-192 and AES-256 in SRTP is defined by **[RFC6188]**. The choice of the AES key length is coupled to the Key Agreement Type, as explained in **Section 5.1.5**.

Other block ciphers may be supported that have the same block size and key sizes as AES. If implemented, they may be used anywhere in ZRTP or SRTP in place of the AES, in the same modes of operation and key size. Notably, in counter mode to replace AES-CM in **[RFC3711]** and **[RFC6188]**, as well as in CFB mode to encrypt a portion of the **Confirm message** and **SASrelay message**. ZRTP endpoints MAY support the **TwoFish** [TwoFish] block cipher.

Cipher Type Block	Meaning
"AES1"	AES with 128-bit keys
"AES2"	AES with 192-bit keys
"AES3"	AES with 256-bit keys
"2FS1"	TwoFish with 128-bit keys
"2FS2"	TwoFish with 192-bit keys
"2FS3"	TwoFish with 256-bit keys

Table 3. Cipher Type Block Values

TOC

5.1.4. Auth Tag Type Block

All ZRTP endpoints MUST support HMAC-SHA1 authentication tags for SRTP, with both 32-bit and 80-bit length tags as defined in **[RFC3711]**.

ZRTP endpoints MAY support 32-bit and 64-bit SRTP authentication tags based on the **Skein hash function** [Skein]. The Skein-512-MAC key length is fixed at 256 bits for this application, and the output length is adjustable. The Skein MAC is defined in Sections 2.6 and 4.3 of **[Skein]** and is not based on the HMAC construct. Reference implementations for Skein may be found at **[Skein1]**. A Skein-based MAC is significantly more efficient than HMAC-SHA1, especially for short SRTP payloads.

The Skein MAC key is computed by the SRTP key derivation function, which is also referred to as the AES-CM PRF, or pseudorandom function. This is defined either in **[RFC3711]** or in

[RFC6188], depending on the selected SRTP AES key length. To compute a Skein MAC key, the SRTP PRF output for the authentication key is left untruncated at 256 bits, instead of the usual truncated length of 160 bits (the key length used by HMAC-SHA1).

Auth Tag Type Block	Meaning
"HS32"	32-bit authentication tag based on HMAC-SHA1 as defined in RFC 3711.
"HS80"	80-bit authentication tag based on HMAC-SHA1 as defined in RFC 3711.
"SK32"	32-bit authentication tag based on Skein-512-MAC as defined in [Skein], with 256-bit key, 32-bit MAC length.
"SK64"	64-bit authentication tag based on Skein-512-MAC as defined in [Skein], with 256-bit key, 64-bit MAC length.

Table 4. Auth Tag Type Values

Implementers should be aware that AES-GCM and AES-CCM for SRTP are expected to become available when **[SRTP-AES-GCM]** is published as an RFC. If an implementer wishes to use these modes when they become available, new Auth Tag Types must be added.

TOC

5.1.5. Key Agreement Type Block

All ZRTP endpoints MUST support DH3k, SHOULD support Preshared, and MAY support EC25, EC38, and DH2k.

If a ZRTP endpoint supports multiple concurrent media streams, such as audio and video, it MUST support **Multistream** mode. Also, if a ZRTP endpoint supports the GoClear message (**Section 4.7.2**), it SHOULD support Multistream, to be used if the two parties choose to return to the secure state after going Clear (as explained in **Section 4.7.2.1**).

For Finite Field Diffie-Hellman, ZRTP endpoints MUST use the DH parameters defined in **[RFC3526]**, as follows. DH3k uses the 3072-bit modular exponentiation group (MODP). DH2k uses the 2048-bit MODP group. The DH generator g is 2. The random Diffie-Hellman secret exponent SHOULD be twice as long as the AES key length. If AES-128 is used, the DH secret value SHOULD be 256 bits long. If AES-256 is used, the secret value SHOULD be 512 bits long.

If Elliptic Curve DH is used, the ECDH algorithm and key generation is from **[NIST-SP800-56A]**. The curves used are from **[NSA-Suite-B]**, which uses the same curves as ECDSA defined by **[FIPS-186-3]**, and can also be found in RFC 5114, Sections 2.6 through 2.8 **[RFC5114]**. ECDH test vectors may be found in RFC 5114, appendices A.6 through A.8 **[RFC5114]**. The validation procedures are from **[NIST-SP800-56A]**, Section 5.6.2.6, method 3, Elliptic Curve Cryptography (ECC) Partial Validation. Both the X and Y coordinates of the point on the curve are sent, in the first and second half of the ECDH public value, respectively. The ECDH result returns only the X coordinate, as specified in SP 800-56A. Useful strategies for implementing ECC may be found in **[RFC6090]**.

The choice of the **negotiated hash algorithm** is coupled to the choice of Key Agreement Type. If ECDH-384 (EC38) is chosen as the key agreement, the negotiated hash algorithm MUST be either SHA-384 or the corresponding SHA-3 successor.

The choice of AES key length is coupled to the choice of Key Agreement Type. If EC38 is chosen as the key agreement, AES-256 (AES3) SHOULD be used but AES-192 MAY be used. If DH3k or EC25 is chosen, any AES key size MAY be used. Note that SRTP as defined in **[RFC3711]** only supports AES-128.

DH2k is intended to provide acceptable security for low power applications, or for applications

that require faster key negotiations. NIST asserts in Table 4 of **[NIST-SP800-131A]** that DH-2048 is safe to use through 2013. The security of DH2k can be augmented by implementing ZRTP's **key continuity features**. DH2k SHOULD use AES-128. If an implementor must use slow hardware, DH2k should precede DH3k in the Hello message.

ECDH-521 SHOULD NOT be used, due to disruptive computational delays. These delays may lead to exhaustion of the retransmission schedule, unless both endpoints have very fast hardware. Note that ECDH-521 is not part of NSA Suite B.

ZRTP also defines two non-DH modes, Multistream and Preshared, in which the SRTP key is derived from a shared secret and some nonce material.

The table below lists the pv length in words and DHPart1 and DHPart2 message length in words for each Key Agreement Type Block.

Key Agreement Type Block	pv words	message words	Meaning
"DH3k"	96	117	DH mode with p=3072 bit prime per RFC 3526, Section 4.
"DH2k"	64	85	DH mode with p=2048 bit prime per RFC 3526, Section 3.
"EC25"	16	37	Elliptic Curve DH, P-256 per RFC 5114, Section 2.6
"EC38"	24	45	Elliptic Curve DH, P-384 per RFC 5114, Section 2.7
"EC52"	33	54	Elliptic Curve DH, P-521 per RFC 5114, Section 2.8 (deprecated - do not use)
"Prsh"	-	-	Preshared Non-DH mode
"Mult"	-	-	Multistream Non-DH mode

Table 5. Key Agreement Type Block Values

TOC

5.1.6. SAS Type Block

The SAS Type determines how the SAS is rendered to the user so that the user may verbally compare it with his partner over the voice channel. This allows detection of a MiTM attack.

All ZRTP endpoints MUST support the base32 and MAY support the base256 rendering schemes for the Short Authentication String, and other SAS rendering schemes. See **Section 4.5.2** for how the sasvalue is computed and **Section 7** for how the SAS is used.

SAS Type Block	Meaning
"B32 "	Short Authentication String using base32 encoding
"B256"	Short Authentication String using base256 encoding (PGP Word List)

Table 6. SAS Type Block Values

For the SAS Type of "B256", the most-significant (leftmost) 16 bits of the 32-bit sasvalue are

rendered in network byte order using the **PGP Word List** [pgpwordlist] **[Juola1][Juola2]**.

For the SAS Type of "B32 ", the most-significant (leftmost) 20 bits of the 32-bit sasvalue are rendered as a form of base32 encoding. The leftmost 20 bits of the sasvalue results in four base32 characters that are rendered, most-significant quintet first, to both ZRTP endpoints. Here is a normative pseudocode implementation of the base32 function:

```
char[4] base32(uint32 bits)
{   int i, n, shift;
    char result[4];
    for (i=0,shift=27; i!=4; ++i,shift-=5)
    {   n = (bits>>shift) & 31;
        result[i] = "ybndrfg8ejkmcpqxot1uwisza345h769"[n];
    }
    return result;
}
```

This base32 encoding scheme differs from RFC 4648, and was designed (by Bryce Wilcox-O'Hearn) to represent bit sequences in a form that is convenient for human users to manipulate with minimal ambiguity. The unusually permuted character ordering was designed for other applications that use bit sequences that do not end on quintet boundaries.

TOC

5.1.7. Signature Type Block

The Signature Type Block specifies what signature algorithm is used to sign the SAS as discussed in **Section 7.2**. The 4-octet Signature Type Block, along with the accompanying signature block, are OPTIONAL and may be present in the **Confirm message** or the **SASrelay message**. The signature types are given in the table below.

```
  Signature   | Meaning
  Type Block  |
  ------------------------------------------
  "PGP "      | OpenPGP Signature, per RFC 4880
              |
  ------------------------------------------
  "X509"      | ECDSA, with X.509v3 cert
              | per RFC 5759 and FIPS-186-3
  ------------------------------------------

  Table 7. Signature Type Block Values
```

Additional details on the signature and signing key format may be found in **Section 7.2**. OpenPGP signatures (Signature Type "PGP ") are discussed in **Section 7.2.1**. The ECDSA curves are over prime fields only, drawn from Appendix D.1.2 of **[FIPS-186-3]**. X.509v3 ECDSA Signatures (Signature Type "X509") are discussed in **Section 7.2.2**.

TOC

5.2. Hello Message

The Hello message has the format shown in **Figure 3**.

All ZRTP messages begin with the preamble value 0x505a, then a 16-bit length in 32-bit words. This length includes only the ZRTP message (including the preamble and the length) but not the ZRTP packet header or CRC. The 8-octet Message Type follows the length field.

Next, there is a 4-character string containing the version (ver) of the ZRTP protocol, which is "1.10" for this specification. Next, there is the Client Identifier string (cid), which is 4 words long and identifies the vendor and release of the ZRTP software. The 256-bit hash image H3 is defined in **Section 9**. The next parameter is the ZID, the 96-bit-long unique identifier for

the ZRTP endpoint, defined in **Section 4.9**.

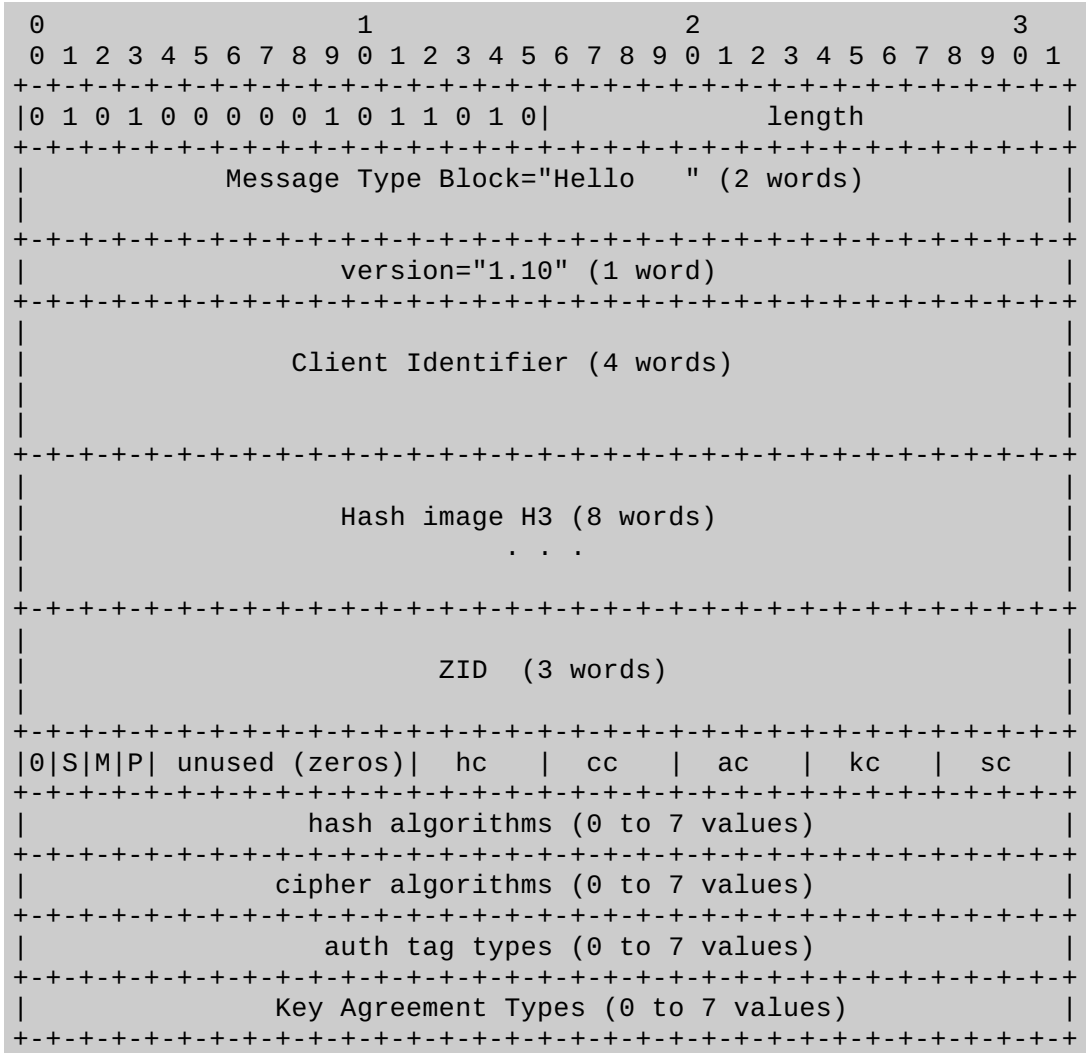
The next four bits include three flag bits:

- The Signature-capable flag (S) indicates this Hello message is sent from a ZRTP endpoint which is able to parse and verify digital signatures, as described in **Section 7.2**. If signatures are not supported, the (S) flag MUST be set to zero.
- The MiTM flag (M) is a Boolean that is set to true if and only if this Hello message is sent from a device, usually a PBX, that has the capability to send an **SASrelay message**.
- The Passive flag (P) is a Boolean normally set to false, and is set to true if and only if this Hello message is sent from a device that is configured to never send a **Commit message**. This would mean it cannot initiate secure sessions, but may act as a responder.

The next 8 bits are unused and SHOULD be set to zero when sent and MUST be ignored on receipt.

Next is a list of supported Hash algorithms, Cipher algorithms, SRTP Auth Tag Types, Key Agreement Types, and SAS Types. The number of listed algorithms are listed for each type: hc=hash count, cc=cipher count, ac=auth tag count, kc=key agreement count, and sc=sas count. The values for these algorithms are defined in Tables 2, 3, 4, 5, and 6. A count of zero means that only the mandatory-to-implement algorithms are supported. Mandatory algorithms MAY be included in the list. The order of the list indicates the preferences of the endpoint. If a mandatory algorithm is not included in the list, it is implicitly added to the end of the list for preference.

The 64-bit MAC at the end of the message is computed across the whole message, not including the MAC, using the MAC algorithm defined in **Section 5.1.2.2**. The MAC key is the sender's H2 (defined in **Section 9**), and thus the MAC cannot be checked by the receiving party until the sender's H2 value is known to the receiving party later in the protocol.



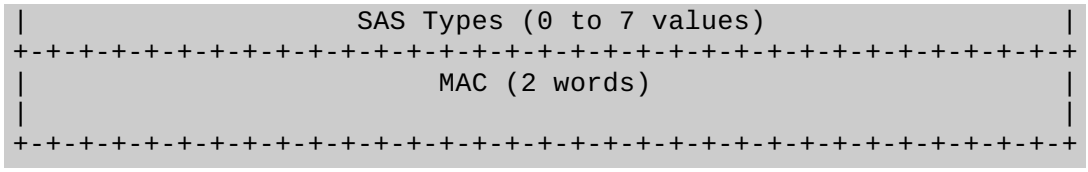


Figure 3: Hello Message Format

TOC

5.3. HelloACK Message

The HelloACK message is used to stop retransmissions of a Hello message. A HelloACK is sent regardless if the version number in the Hello is supported or the algorithm list supported. The receipt of a HelloACK stops retransmission of the Hello message. The format is shown in the figure below. A Commit message may be sent in place of a HelloACK by an Initiator, if a Commit message is ready to be sent promptly.

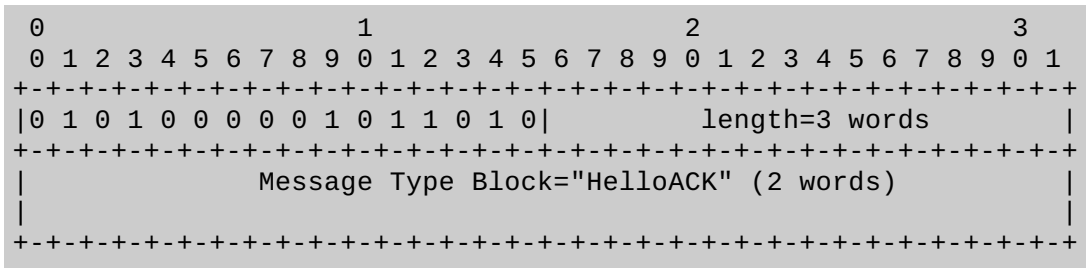


Figure 4: HelloACK Message Format

TOC

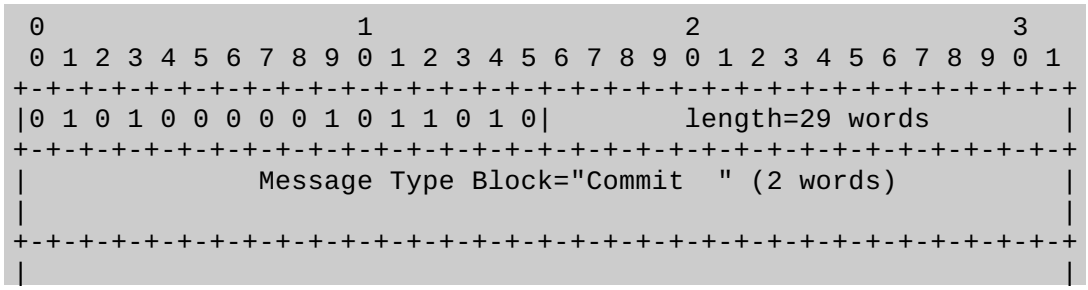
5.4. Commit Message

The Commit message is sent to initiate the key agreement process after both sides have received a Hello message, which means it can only be sent after receiving both a Hello message and a HelloACK message. There are three subtypes of Commit messages, whose formats are shown in Figures **5**, **6**, and **7**.

The Commit message contains the Message Type Block, then the 256-bit hash image H2, which is defined in **Section 9**. The next parameter is the initiator's ZID, the 96-bit-long unique identifier for the ZRTP endpoint, which MUST have the same value as was used in the Hello message.

Next, there is a list of algorithms selected by the initiator (hash, cipher, auth tag type, key agreement, sas type). For a DH Commit, the hash value hvi is a hash of the DHPart2 of the Initiator and the Responder's Hello message, as explained in **Section 4.4.1.1**.

The 64-bit MAC at the end of the message is computed across the whole message, not including the MAC, using the MAC algorithm defined in **Section 5.1.2.2**. The MAC key is the sender's H1 (defined in **Section 9**), and thus the MAC cannot be checked by the receiving party until the sender's H1 value is known to the receiving party later in the protocol.



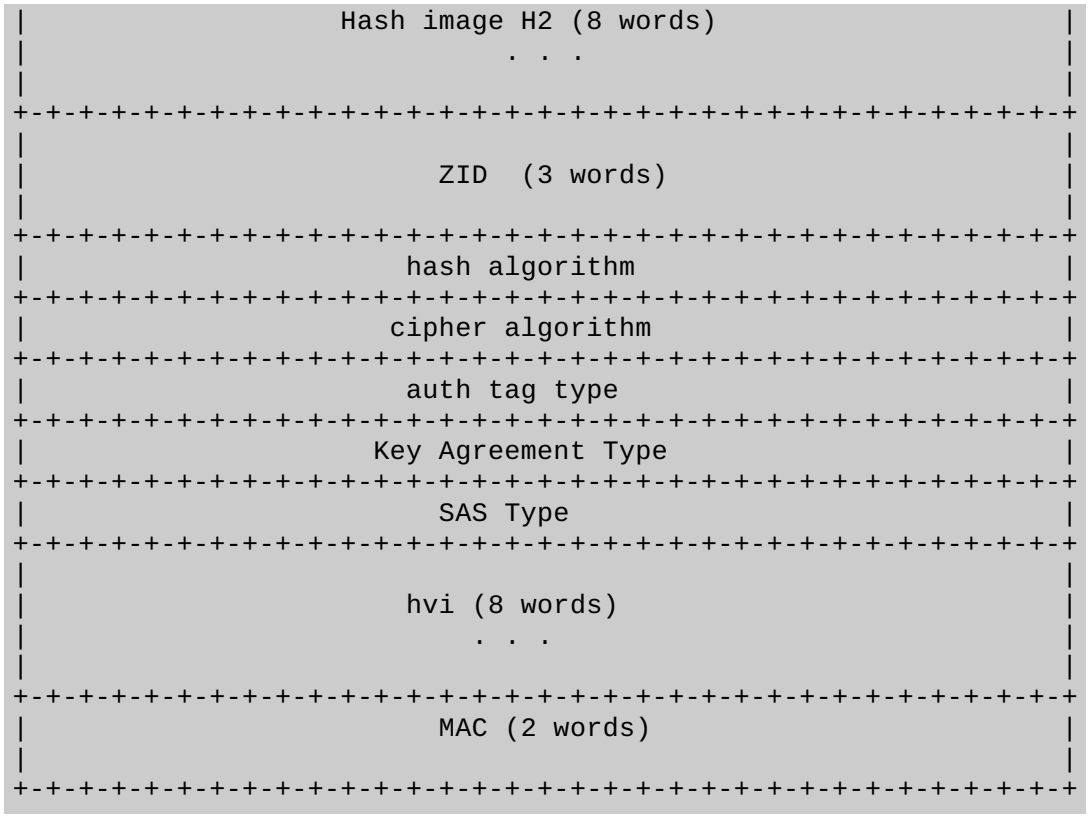


Figure 5: DH Commit Message Format

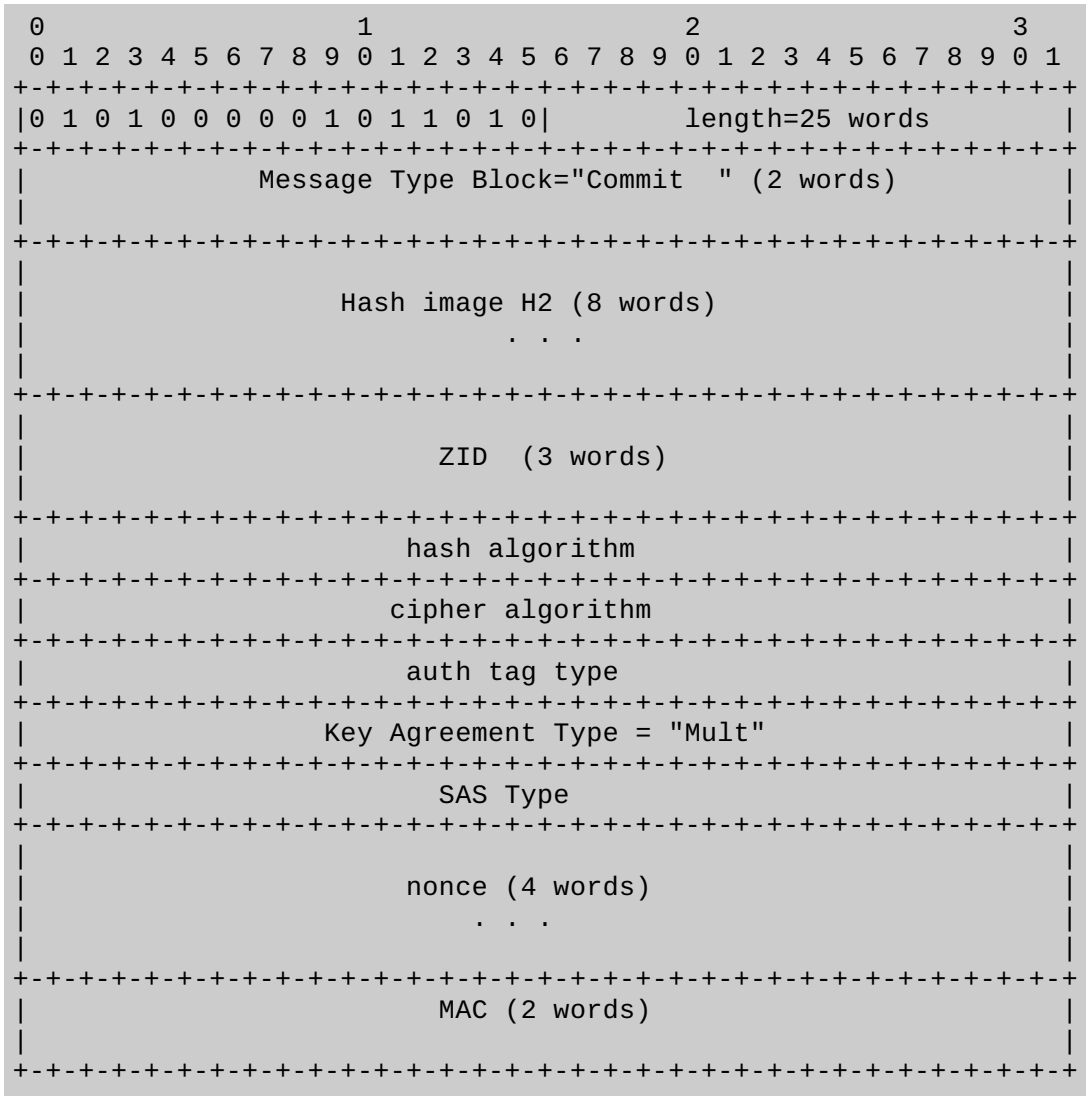


Figure 6: Multistream Commit Message Format

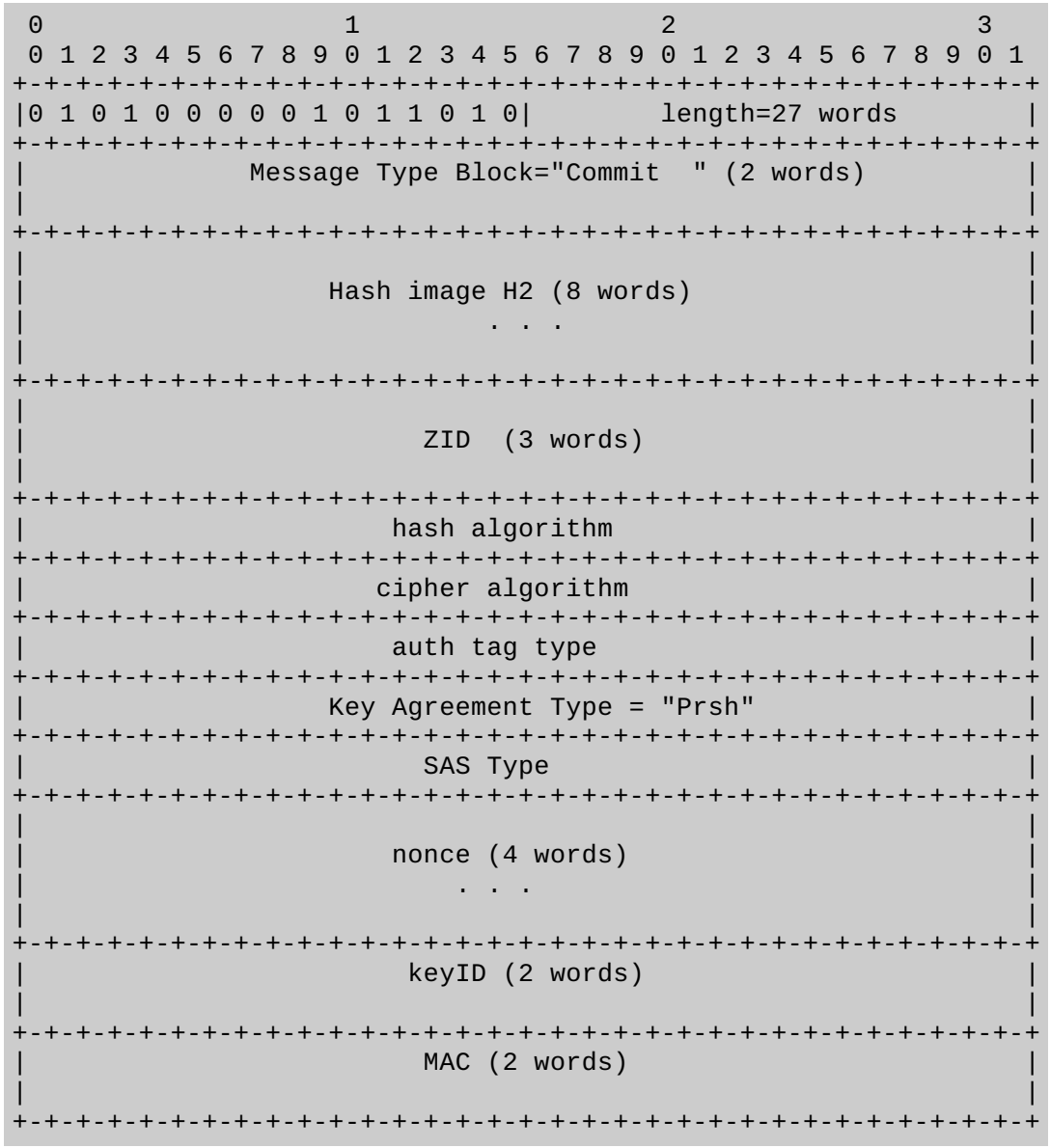


Figure 7: Preshared Commit Message Format

5.5. DHPart1 Message

TOC

The DHPart1 message shown in **Figure 8** begins the DH exchange. It is sent by the Responder if a valid Commit message is received from the Initiator. The length of the pvr value and the length of the DHPart1 message depends on the Key Agreement Type chosen. This information is contained in the table in **Section 5.1.5**. Note that for both Multistream and Preshared modes, no DHPart1 or DHPart2 message will be sent.

The 256-bit hash image H1 is defined in **Section 9**.

The next four parameters are non-invertible hashes (computed in **Section 4.3.1**) of potential shared secrets used in generating the ZRTP secret s0. The first two, rs1IDr and rs2IDr, are the hashes of the responder's two retained shared secrets, truncated to 64 bits. Next, there is auxsecretIDr, a hash of the responder's auxsecret (defined in **Section 4.3**), truncated to 64 bits. The last parameter is a hash of the trusted MiTM PBX shared secret pbxsecret, defined in **Section 7.3.1**.

The 64-bit MAC at the end of the message is computed across the whole message, not including the MAC, using the MAC algorithm defined in **Section 5.1.2.2**. The MAC key is the

sender's H0 (defined in **Section 9**), and thus the MAC cannot be checked by the receiving party until the sender's H0 value is known to the receiving party later in the protocol.

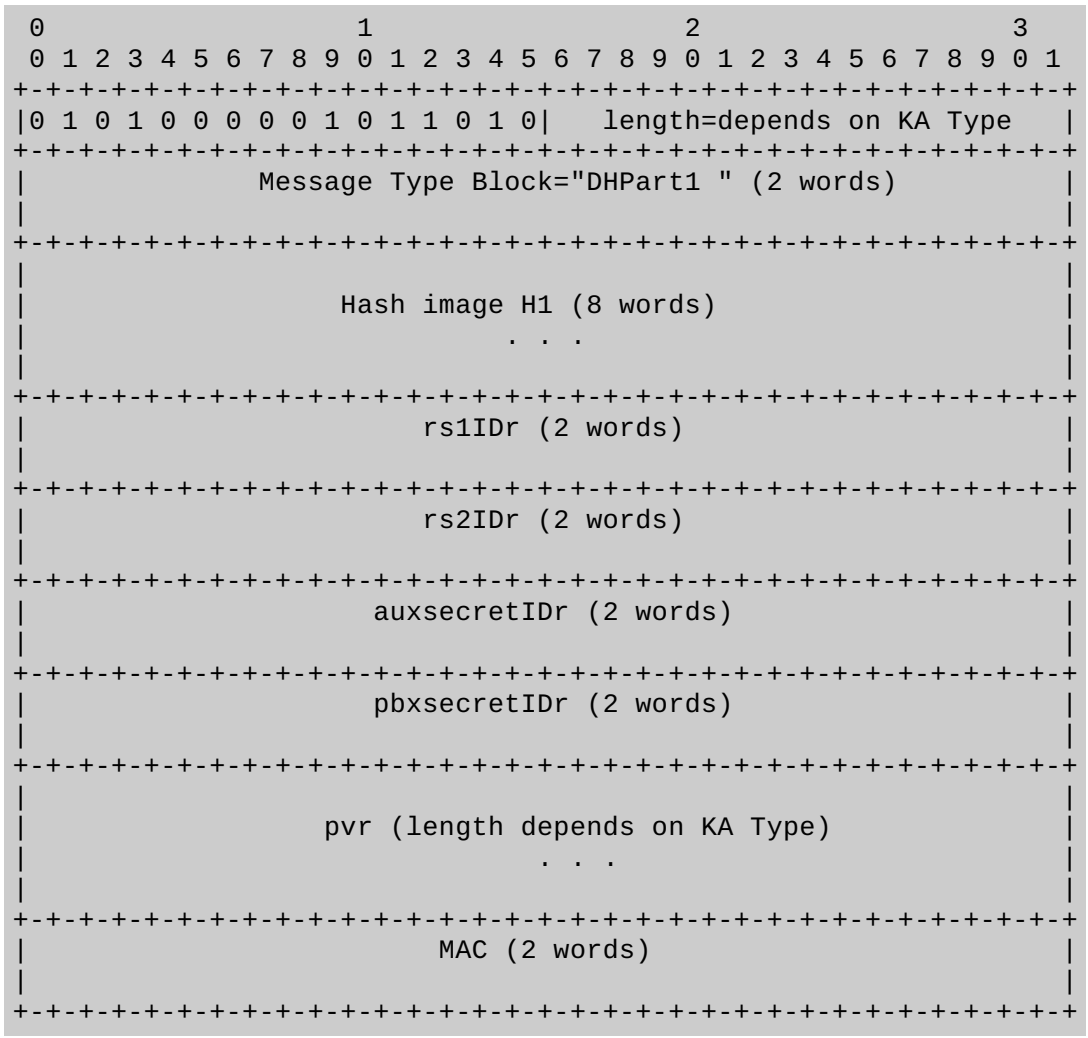


Figure 8: DHPart1 Message Format

5.6. DHPart2 Message

TOC

The DHPart2 message, shown in **Figure 9**, completes the DH exchange. It is sent by the Initiator if a valid DHPart1 message is received from the Responder. The length of the pvi value and the length of the DHPart2 message depends on the Key Agreement Type chosen. This information is contained in the table in **Section 5.1.5**. Note that for both Multistream and Preshared modes, no DHPart1 or DHPart2 message will be sent.

The 256-bit hash image H1 is defined in **Section 9**.

The next four parameters are non-invertible hashes (computed in **Section 4.3.1**) of potential shared secrets used in generating the ZRTP secret s0. The first two, rs1IDi and rs2IDi, are the hashes of the initiator's two retained shared secrets, truncated to 64 bits. Next, there is auxsecretIDi, a hash of the initiator's auxsecret (defined in **Section 4.3**), truncated to 64 bits. The last parameter is a hash of the trusted MiTM PBX shared secret pbxsecret, defined in **Section 7.3.1**.

The 64-bit MAC at the end of the message is computed across the whole message, not including the MAC, using the MAC algorithm defined in **Section 5.1.2.2**. The MAC key is the sender's H0 (defined in **Section 9**), and thus the MAC cannot be checked by the receiving party until the sender's H0 value is known to the receiving party later in the protocol.

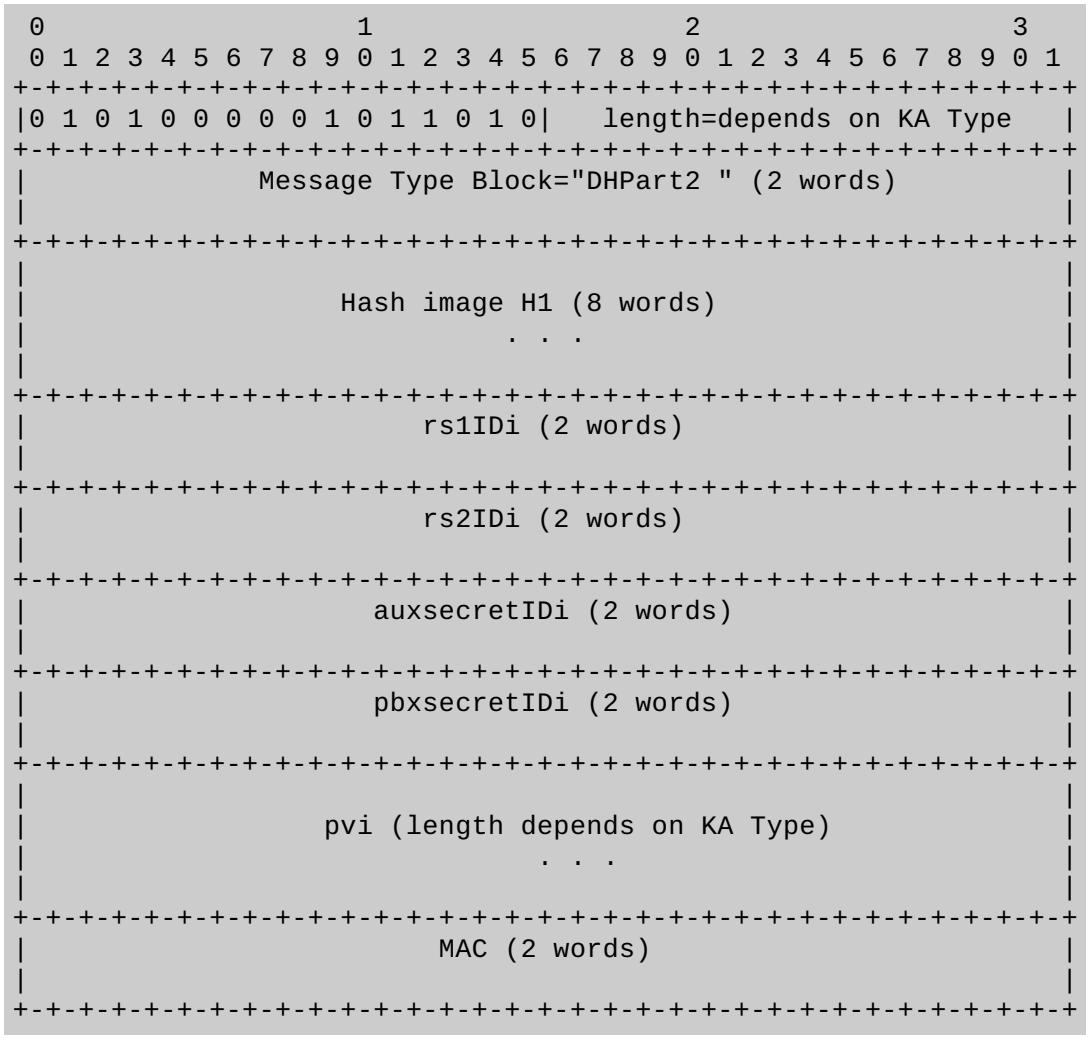


Figure 9: DHPart2 Message Format

5.7. Confirm1 and Confirm2 Messages

TOC

The Confirm1 message is sent by the Responder in response to a valid DHPart2 message after the SRTP session key and parameters have been negotiated. The Confirm2 message is sent by the Initiator in response to a Confirm1 message. The format is shown in **Figure 10**. The message contains the Message Type Block "Confirm1" or "Confirm2". Next, there is the confirm_mac, a MAC computed over the encrypted part of the message (shown enclosed by "====" in **Figure 10**). This confirm_mac is keyed and computed according to **Section 4.6**. The next 16 octets contain the CFB Initialization Vector. The rest of the message is encrypted using CFB and protected by the confirm_mac.

The first field inside the encrypted region is the hash preimage H0, which is defined in detail in **Section 9**.

The next 15 bits are not used and SHOULD be set to zero when sent and MUST be ignored when received in Confirm1 or Confirm2 messages.

The next 9 bits contain the signature length. If no SAS signature (described in **Section 7.2**) is present, all bits are set to zero. The signature length is in words and includes the signature type block. If the calculated signature octet count is not a multiple of 4, zeros are added to pad it out to a word boundary. If no signature is present, the overall length of the Confirm1 or Confirm2 message will be set to 19 words.

The next 8 bits are used for flags. Undefined flags are set to zero and ignored. Four flags are currently defined. The PBX Enrollment flag (E) is a Boolean bit defined in **Section 7.3.1**. The SAS Verified flag (V) is a Boolean bit defined in **Section 7.1**. The Allow Clear flag (A) is a Boolean bit defined in **Section 4.7.2**. The Disclosure Flag (D) is a Boolean bit defined in **Section 11**. The cache expiration interval is defined in **Section 4.9**.

If the signature length (in words) is non-zero, a signature type block will be present along with a signature block. Next, there is the signature block. The signature block includes the signature and the key (or a link to the key) used to generate the signature (**Section 7.2**).

CFB mode [NIST-SP800-38A] is applied with a feedback length of 128 bits, a full cipher block, and the final block is truncated to match the exact length of the encrypted data. The CFB Initialization Vector is a 128-bit random nonce. The block cipher algorithm and the key size are the same as the **negotiated block cipher** for media encryption. CFB is used to encrypt the part of the Confirm1 message beginning after the CFB IV to the end of the message (the encrypted region is enclosed by "====" in **Figure 10**).

The responder uses the zrtpkeyr to encrypt the Confirm1 message. The initiator uses the zrtpkeyi to encrypt the Confirm2 message.

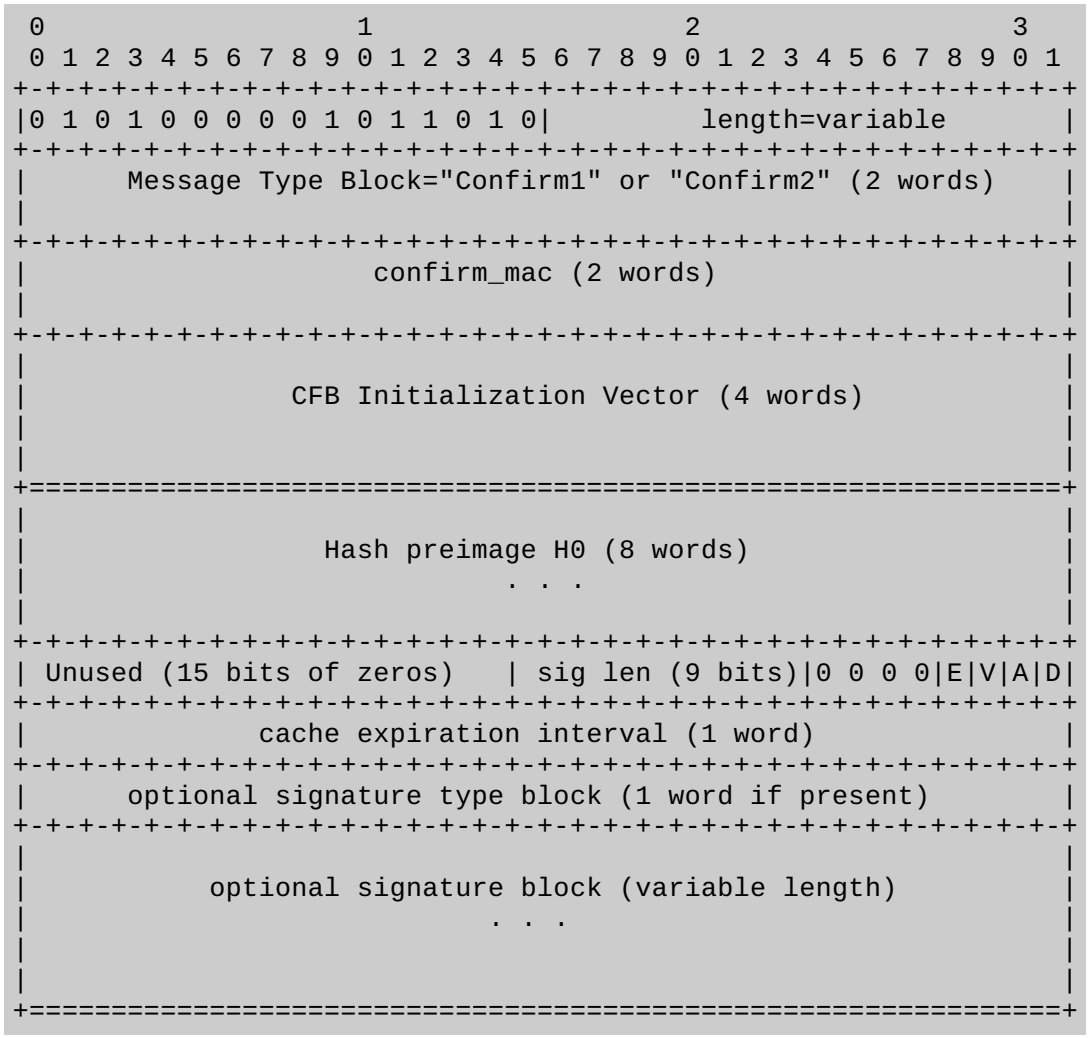


Figure 10: Confirm1 and Confirm2 Message Format

5.8. Conf2ACK Message

TOC

The Conf2ACK message is sent by the Responder in response to a valid Confirm2 message. The message format for the Conf2ACK is shown in the figure below. The receipt of a Conf2ACK stops retransmission of the Confirm2 message. Note that the first SRTP media (with a valid SRTP auth tag) from the responder also stops retransmission of the Confirm2 message.

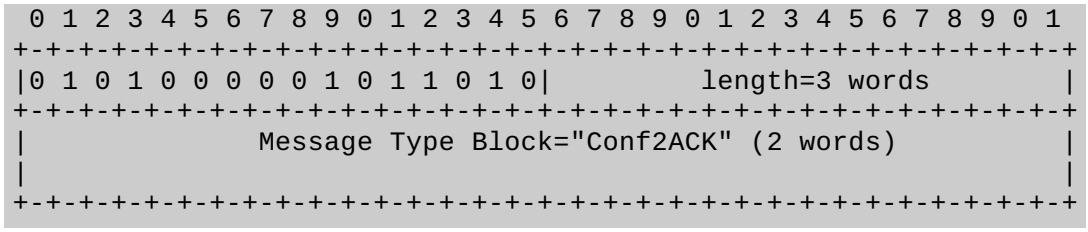


Figure 11: Conf2ACK Message Format

TOC

5.9. Error Message

The Error message is sent to terminate an in-process ZRTP key agreement exchange due to an error. The format is shown in the figure below. The use of the Error message is described in **Section 4.7.1**.

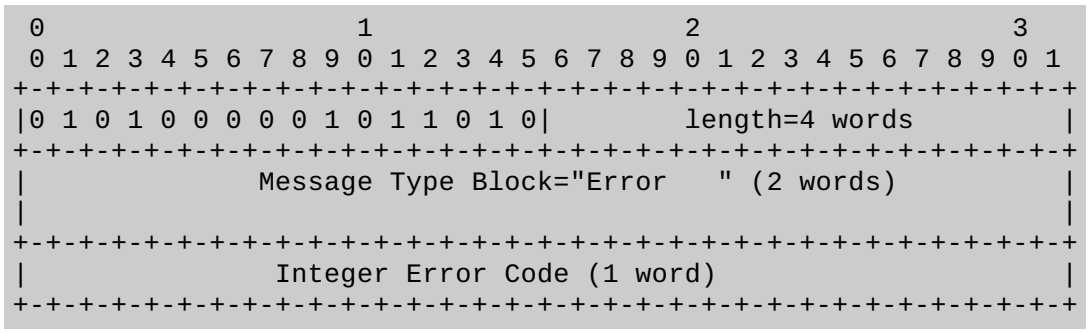


Figure 12: Error Message Format

Defined hexadecimal values for the Error Code are listed in the table below.

Error Code	Meaning
0x10	Malformed packet (CRC OK, but wrong structure)
0x20	Critical software error
0x30	Unsupported ZRTP version
0x40	Hello components mismatch
0x51	Hash Type not supported
0x52	Cipher Type not supported
0x53	Public key exchange not supported
0x54	SRTP auth tag not supported
0x55	SAS rendering scheme not supported
0x56	No shared secret available, DH mode required
0x61	DH Error: bad pvi or pvr (== 1, 0, or p-1)
0x62	DH Error: hvi != hashed data
0x63	Received relayed SAS from untrusted MiTM
0x70	Auth Error: Bad Confirm pkt MAC

```
 ------------------------------------------------------------
  0x80      | Nonce reuse
 ------------------------------------------------------------
  0x90      | Equal ZIDs in Hello
 ------------------------------------------------------------
  0x91      | SSRC collision
 ------------------------------------------------------------
  0xA0      | Service unavailable
 ------------------------------------------------------------
  0xB0      | Protocol timeout error
 ------------------------------------------------------------
  0x100     | GoClear message received, but not allowed
 ------------------------------------------------------------

 Table 8. ZRTP Error Codes
```

TOC

5.10. ErrorACK Message

The ErrorACK message is sent in response to an Error message. The receipt of an ErrorACK stops retransmission of the Error message. The format is shown in the figure below.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 1 0 1 0 0 0 0 0 1 0 1 1 0 1 0|        length=3 words          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              Message Type Block="ErrorACK" (2 words)           |
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 13: ErrorACK Message Format

TOC

5.11. GoClear Message

Support for the GoClear message is OPTIONAL in the protocol, and it is sent to switch from SRTP to RTP. The format is shown in the figure below. The clear_mac is used to authenticate the GoClear message so that bogus GoClear messages introduced by an attacker can be detected and discarded. The use of GoClear is described in **Section 4.7.2**.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 1 0 1 0 0 0 0 0 1 0 1 1 0 1 0|        length=5 words          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              Message Type Block="GoClear " (2 words)           |
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       clear_mac (2 words)                     |
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 14: GoClear Message Format

TOC

5.12. ClearACK Message

Support for the ClearACK message is OPTIONAL in the protocol, and it is sent to acknowledge receipt of a GoClear. A ClearACK is only sent if the clear_mac from the GoClear message is authenticated. Otherwise, no response is returned. The format is shown in the figure below.

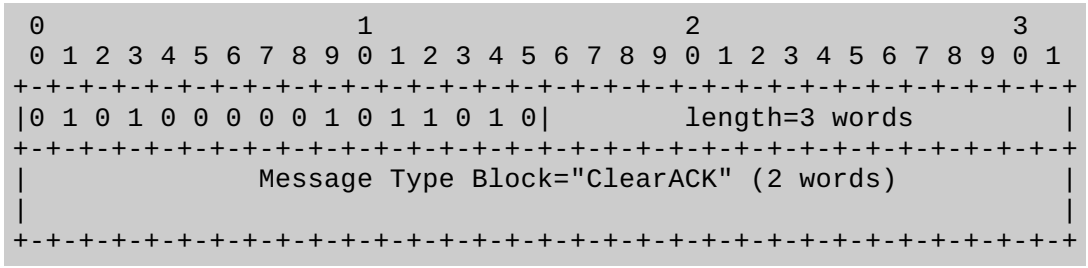


Figure 15: ClearACK Message Format

5.13. SASrelay Message

The SASrelay message is sent by a trusted MiTM, most often a PBX. It is not sent as a response to a packet, but is sent as a self-initiated packet by the **trusted MiTM**. It can only be sent after the rest of the ZRTP key negotiations have completed, after the Confirm messages and their ACKs. It can only be sent after the trusted MiTM has finished key negotiations with the other party, because it is the other party's SAS that is being relayed. It is sent with retry logic until a RelayACK message (**Section 5.14**) is received or the retry schedule has been exhausted.

If a device, usually a PBX, sends an SASrelay message, it MUST have previously declared itself as a MiTM device by setting the MiTM (M) flag in the **Hello message**. If the receiver of the SASrelay message did not previously receive a Hello message with the MiTM (M) flag set, the Relayed SAS SHOULD NOT be rendered. A RelayACK is still sent, but no Error message is sent.

The SASrelay message format is shown in **Figure 16**. The message contains the Message Type Block "SASrelay". Next, there is a MAC computed over the encrypted part of the message (shown enclosed by "====" in **Figure 16**). This MAC is keyed the same way as the confirm_mac in the Confirm messages (see **Section 4.6**). The next 16 octets contain the CFB Initialization Vector. The rest of the message is encrypted using CFB and protected by the MAC.

The next 15 bits are not used and SHOULD be set to zero when sent, and they MUST be ignored when received in SASrelay messages.

The next 9 bits contain the signature length. The trusted MiTM MAY compute a digital signature on the SAS hash, as described in **Section 7.2**, using a persistent signing key owned by the trusted MiTM. If no SAS signature is present, all bits are set to zero. The signature length is in words and includes the signature type block. If the calculated signature octet count is not a multiple of 4, zeros are added to pad it out to a word boundary. If no signature block is present, the overall length of the SASrelay message will be set to 19 words.

The next 8 bits are used for flags. Undefined flags are set to zero and ignored. Three flags are currently defined. The Disclosure Flag (D) is a Boolean bit defined in **Section 11**. The Allow Clear flag (A) is a Boolean bit defined in **Section 4.7.2**. The SAS Verified flag (V) is a Boolean bit defined in **Section 7.1**. These flags are updated values to the same flags provided earlier in the Confirm message, but they are updated to reflect the new flag information relayed by the PBX from the other party.

The next 32-bit word contains the SAS rendering scheme for the relayed sashash, which will be the same rendering scheme used by the other party on the other side of the trusted MiTM. **Section 7.3** describes how the PBX determines whether the ZRTP client regards the PBX as a trusted MiTM. If the PBX determines that the ZRTP client trusts the PBX, the next 8 words contain the sashash relayed from the other party. The first 32-bit word of the sashash

contains the sasvalue, which may be rendered to the user using the specified SAS rendering scheme. If this SASrelay message is being sent to a ZRTP client that does not trust this MiTM, the sashash will be ignored by the recipient and should be set to zeros by the PBX.

If the signature length (in words) is non-zero, a signature type block will be present along with a signature block. Next, there is the signature block. The signature block includes the signature and the key (or a link to the key) used to generate the signature (**Section 7.2**).

CFB mode [NIST-SP800-38A] is applied with a feedback length of 128 bits, a full cipher block, and the final block is truncated to match the exact length of the encrypted data. The CFB Initialization Vector is a 128-bit random nonce. The block cipher algorithm and the key size is same as the **negotiated block cipher** for media encryption. CFB is used to encrypt the part of the SASrelay message beginning after the CFB IV to the end of the message (the encrypted region is enclosed by "====" in **Figure 16**).

Depending on whether the trusted MiTM had taken the role of the initiator or the responder during the ZRTP key negotiation, the SASrelay message is encrypted with zrtpkeyi or zrtpkeyr.

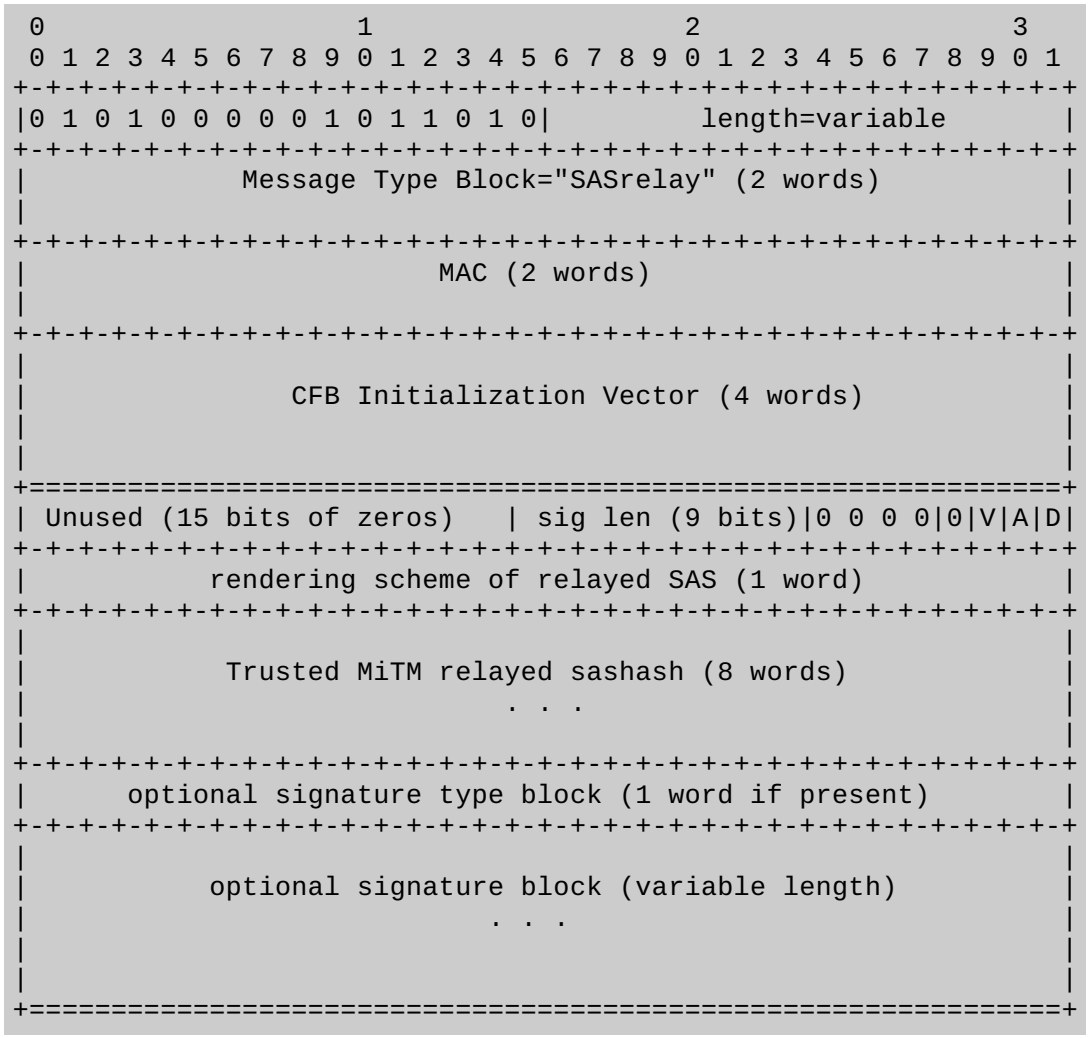


Figure 16: SASrelay Message Format

5.14. RelayACK Message

TOC

The RelayACK message is sent in response to a valid SASrelay message. The message format for the RelayACK is shown in the figure below. The receipt of a RelayACK stops retransmission of the SASrelay message.

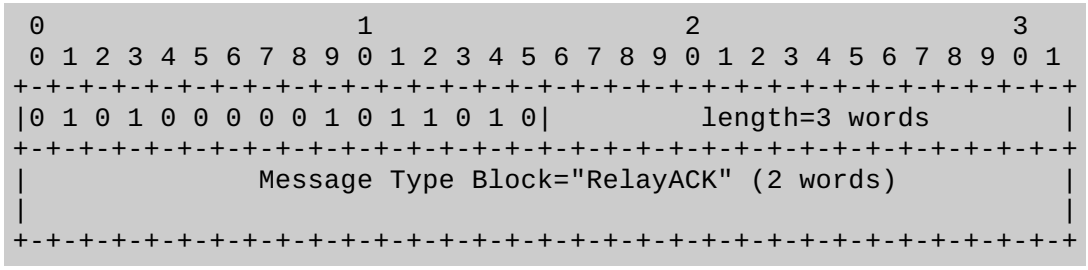


Figure 17: RelayACK Message Format

TOC

5.15. Ping Message

The Ping and PingACK messages are unrelated to the rest of the ZRTP protocol. No ZRTP endpoint is required to generate a Ping message, but every ZRTP endpoint MUST respond to a Ping message with a PingACK message.

Although Ping and PingACK messages have no effect on the rest of the ZRTP protocol, their inclusion in this specification simplifies the design of "bump-in-the-wire" **ZRTP proxies** (notably, **[Zfone]**). It enables proxies to be designed that do not rely on assistance from the signaling layer to map out the associations between media streams and ZRTP endpoints.

Before sending a ZRTP Hello message, a ZRTP proxy MAY send a Ping message as a means to sort out which RTP media streams are connected to particular ZRTP endpoints. Ping messages are generated only by ZRTP proxies. If neither party is a ZRTP proxy, no Ping messages will be encountered. Ping retransmission behavior is discussed in **Section 6**.

The Ping message (**Figure 18**) contains an "EndpointHash", defined in **Section 5.16**.

The Ping message contains a version number that defines what version of PingACK is requested. If that version number is supported by the Ping responder, a PingACK with a format that matches that version will be received. Otherwise, a PingACK with a lower version number may be received.

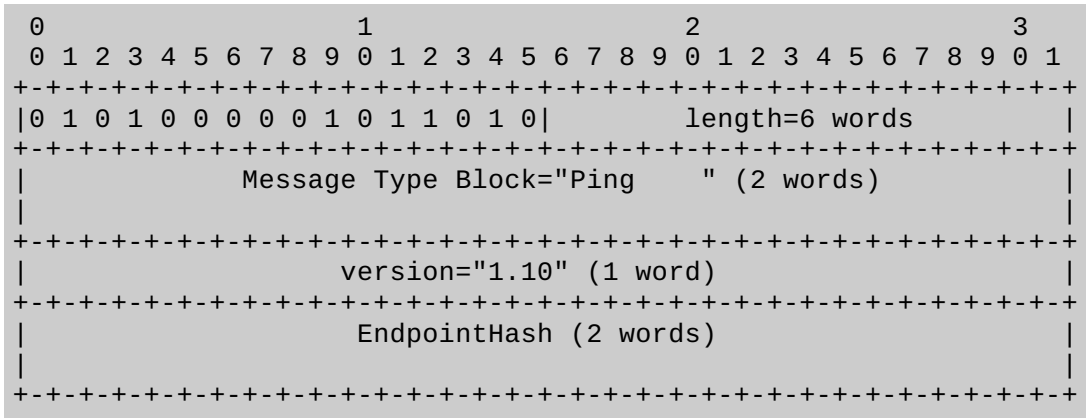


Figure 18: Ping Message Format

TOC

5.16. PingACK Message

A PingACK message is sent only in response to a Ping. A ZRTP endpoint MUST respond to a Ping with a PingACK message. The version of PingACK requested is contained in the Ping message. If that version number is supported, a PingACK with a format that matches that version MUST be sent. Otherwise, if the version number of the Ping is not supported, a PingACK SHOULD be sent in the format of the highest supported version known to the Ping responder. Only version "1.10" is supported in this specification.

The PingACK message carries its own 64-bit EndpointHash, distinct from the EndpointHash of the other party's Ping message. It is REQUIRED that it be highly improbable for two participants in a call to have the same EndpointHash and that an EndpointHash maintains a persistent value between calls. For a normal ZRTP endpoint, such as a ZRTP-enabled VoIP client, the EndpointHash can be just the truncated ZID. For a ZRTP endpoint such as a PBX that has multiple endpoints behind it, the EndpointHash must be a distinct value for each endpoint behind it. It is recommended that the EndpointHash be a truncated hash of the ZID of the ZRTP endpoint concatenated with something unique about the actual endpoint or phone behind the PBX. This may be the SIP URI of the phone, the PBX extension number, or the local IP address of the phone, whichever is more readily available in the application environment:

```
EndpointHash = hash(ZID || SIP URI of the endpoint)

EndpointHash = hash(ZID || PBX extension number of the endpoint)

EndpointHash = hash(ZID || local IP address of the endpoint)
```

Any of these formulae confer uniqueness for the simple case of terminating the ZRTP connection at the VoIP client, or the more complex case of a PBX terminating the ZRTP connection for multiple VoIP phones in a conference call, all sharing the PBX's ZID, but with separate IP addresses behind the PBX. There is no requirement for the same hash function to be used by both parties.

The PingACK message contains the EndpointHash of the sender of the PingACK as well as the EndpointHash of the sender of the Ping. The Source Identifier (SSRC) received in the ZRTP header from the Ping packet (**Figure 2**) is copied into the PingACK message body (**Figure 19**). This SSRC is not the SSRC of the sender of the PingACK.

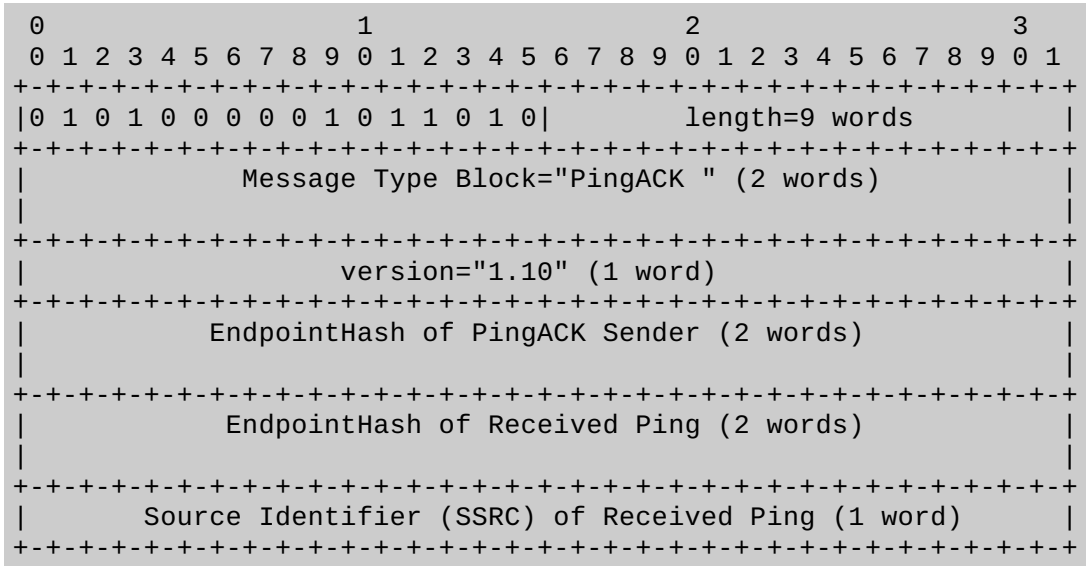


Figure 19: PingACK Message Format

6. Retransmissions

TOC

ZRTP uses two retransmission timers T1 and T2. T1 is used for retransmission of Hello messages, when the support of ZRTP by the other endpoint may not be known. T2 is used in retransmissions of all the other ZRTP messages.

All message retransmissions MUST be identical to the initial message including nonces, public values, etc; otherwise, hashes of the message sequences may not agree.

Practical experience has shown that RTP packet loss at the start of an RTP session can be

extremely high. Since the entire ZRTP message exchange occurs during this period, the defined retransmission scheme is defined to be aggressive. Since ZRTP packets with the exception of the DHPart1 and DHPart2 messages are small, this should have minimal effect on overall bandwidth utilization of the media session.

ZRTP endpoints MUST NOT exceed the bandwidth of the resulting media session as determined by the offer/answer exchange in the signaling layer.

The **Ping message** may follow the same retransmission schedule as the Hello message, but this is not required in this specification. Ping message retransmission is subject to application-specific ZRTP proxy heuristics.

Hello ZRTP messages are retransmitted at an interval that starts at T1 seconds and doubles after every retransmission, capping at 200 ms. T1 has a recommended initial value of 50 ms. A Hello message is retransmitted 20 times before giving up, which means the entire retry schedule for Hello messages is exhausted after 3.75 seconds (50 + 100 + 18*200 ms). Retransmission of a Hello ends upon receipt of a HelloACK or Commit message.

The post-Hello ZRTP messages are retransmitted only by the session initiator -- that is, only Commit, DHPart2, and Confirm2 are retransmitted if the corresponding message from the responder, DHPart1, Confirm1, and Conf2ACK, are not received. Note that the Confirm2 message retransmission can also be stopped by receiving the first SRTP media (with a valid SRTP auth tag) from the responder.

The GoClear, Error, and SASrelay messages may be initiated and retransmitted by either party, and responded to by the other party, regardless of which party is the overall session initiator. They are retransmitted if the corresponding response message ClearACK, ErrorACK, and RelayACK are not received.

Non-Hello (and non-Ping) ZRTP messages are retransmitted at an interval that starts at T2 seconds and doubles after every retransmission, capping at 1200 ms. T2 has a recommended initial value of 150 ms. Each non-Hello message is retransmitted 10 times before giving up, which means the entire retry schedule is exhausted after 9.45 seconds (150 + 300 + 600 + 7*1200 ms). Only the initiator performs retransmissions. Each message has a response message that stops retransmissions, as shown in the table below. The higher values of T2 means that retransmissions will likely occur only in the event of packet loss.

Message	Acknowledgement Message
Hello	HelloACK or Commit
Commit	DHPart1 or Confirm1
DHPart2	Confirm1
Confirm2	Conf2ACK or SRTP media
GoClear	ClearACK
Error	ErrorACK
SASrelay	RelayACK
Ping	PingACK

Table 9. Retransmitted ZRTP Messages and Responses

The retry schedule must handle not only packet loss, but also slow or heavily loaded peers that need additional time to perform their DH calculations. The following mitigations are recommended:

- Slow or heavily loaded ZRTP endpoints that are at risk of taking too long to perform their DH calculation SHOULD use a HelloACK message instead of a Commit message to reply to a Hello from the other party.
- If a ZRTP endpoint has evidence that the other party is a ZRTP endpoint, by receiving a Hello message or Ping message, or by receiving a Hello Hash in the signaling layer, it SHOULD extend its own Hello retry schedule to span at least 12 seconds of retries. If this extended Hello retry schedule is exhausted without receiving a HelloACK or Commit message, a late Commit message from the peer SHOULD still be accepted.

These recommended retransmission intervals are designed for a typical broadband Internet connection. In some high-latency communication channels, such as those provided by some mobile phone environments or geostationary satellites, a different retransmission schedule

may be used. The initial value for the T1 or T2 retransmission timer should be increased to be no less than the round-trip time provided by the communications channel. It should take into account the time required to transmit the entire message and the entire reply, as well as a reasonable time estimate to perform the DH calculation.

ZRTP has its own retransmission schedule because it is carried along with RTP, usually over UDP. In unusual cases, RTP can run over a non-UDP transport, such as TCP or DCCP, which provides its own built-in retransmission mechanism. It may be hard for the ZRTP endpoint to detect that TCP is being used if media relays are involved. The ZRTP endpoint may be sending only UDP, but there may be a media relay along the media path that converts from UDP to TCP for part of the journey. Or, if the ZRTP endpoint is sending TCP, the media relay might be converting from TCP to UDP. There have been empirical observations of this in the wild. In cases where TCP is used, ZRTP and TCP might together generate some extra retransmissions. It is tempting to avoid this effect by eliminating the ZRTP retransmission schedule when connected to a TCP channel, but that would risk failure of the protocol, because it may not be TCP all the way to the remote ZRTP endpoint. It only takes a few packets to complete a ZRTP exchange, so trying to optimize out the extra retransmissions in that scenario is not worth the risk.

After receiving a Commit message, but before receiving a Confirm2 message, if a ZRTP responder receives no ZRTP messages for more than 10 seconds, the responder MAY send a protocol timeout Error message and terminate the ZRTP protocol.

TOC

7. Short Authentication String

This section will discuss the implementation of the Short Authentication String, or SAS in ZRTP. The SAS can be verbally compared by the human users reading the string aloud, or it can be compared by validating an OPTIONAL digital signature (described in **Section 7.2**) exchanged in the Confirm1 or Confirm2 messages.

The use of **hash commitment in the DH exchange** constrains the attacker to only one guess to generate the correct SAS in his attack, which means the SAS can be quite short. A 16-bit SAS, for example, provides the attacker only one chance out of 65536 of not being detected. How the hash commitment enables the SAS to be so short is explained in **Section 4.4.1.1**.

There is only one SAS value computed per call. That is the SAS value for the first media stream established, which is calculated in **Section 4.5.2**. This SAS applies to all media streams for the same session.

The SAS SHOULD be rendered to the user for authentication. The rendering of the SAS value through the user interface at both endpoints depends on the SAS Type agreed upon in the Commit message. See **Section 5.1.6** for a description of how the SAS is rendered to the user.

The SAS is not treated as a secret value, but it must be compared to see if it matches at both ends of the communications channel. The two users verbally compare it using their human voices, human ears, and human judgement. If it doesn't match, it indicates the presence of a MiTM attack.

It is worse than useless and absolutely unsafe to rely on a robot voice from the remote endpoint to compare the SAS, because a robot voice can be trivially forged by a MiTM. The SAS verbal comparison can only be done with a real live human at the remote endpoint.

TOC

7.1. SAS Verified Flag

The SAS Verified flag (V) is set based on the user indicating that SAS comparison has been successfully performed. The SAS Verified flag is exchanged securely in the Confirm1 and Confirm2 messages (**Figure 10**) of the next session. In other words, each party sends the SAS Verified flag from the previous session in the Confirm message of the current session. It is perfectly reasonable to have a ZRTP endpoint that never sets the SAS Verified flag, because it would require adding complexity to the user interface to allow the user to set it. The SAS Verified flag is not required to be set, but if it is available to the client software, it

allows for the possibility that the client software could render to the user that the SAS verify procedure was carried out in a previous session.

Regardless of whether there is a user interface element to allow the user to set the SAS Verified flag, it is worth caching a shared secret, because doing so reduces opportunities for an attacker in the next call.

If at any time the users carry out the SAS comparison procedure, and it actually fails to match, then this means there is a very resourceful MiTM. If this is the first call, the MiTM was there on the first call, which is impressive enough. If it happens in a later call, it also means the MiTM must also know the cached shared secret, because you could not have carried out any voice traffic at all unless the session key was correctly computed and is also known to the attacker. This implies the MiTM must have been present in all the previous sessions, since the initial establishment of the first shared secret. This is indeed a resourceful attacker. It also means that if at any time he ceases his participation as a MiTM on one of your calls, the protocol will detect that the cached shared secret is no longer valid -- because it was really two different shared secrets all along, one of them between Alice and the attacker, and the other between the attacker and Bob. The continuity of the cached shared secrets makes it possible for us to detect the MiTM when he inserts himself into the ongoing relationship, as well as when he leaves. Also, if the attacker tries to stay with a long lineage of calls, but fails to execute a DH MiTM attack for even one missed call, he is permanently excluded. He can no longer resynchronize with the chain of cached shared secrets.

A user interface element (i.e., a checkbox or button) is needed to allow the user to tell the software the SAS verify was successful, causing the software to set the SAS Verified flag (V), which (together with our cached shared secret) obviates the need to perform the SAS procedure in the next call. An additional user interface element can be provided to let the user tell the software he detected an actual SAS mismatch, which indicates a MiTM attack. The software can then take appropriate action, clearing the SAS Verified flag, and erase the cached shared secret from this session. It is up to the implementer to decide if this added user interface complexity is warranted.

If the SAS matches, it means there is no MiTM, which also implies it is now safe to trust a cached shared secret for later calls. If inattentive users don't bother to check the SAS, it means we don't know whether there is or is not a MiTM, so even if we do establish a new cached shared secret, there is a risk that our potential attacker may have a subsequent opportunity to continue inserting himself in the call, until we finally get around to checking the SAS. If the SAS matches, it means no attacker was present for any previous session since we started propagating cached shared secrets, because this session and all the previous sessions were also authenticated with a continuous lineage of shared secrets.

TOC

7.2. Signing the SAS

In most applications, it is desirable to avoid the added complexity of a PKI-backed digital signature, which is why ZRTP is designed not to require it. Nonetheless, in some applications, it may be hard to arrange for two human users to verbally compare the SAS. Or, an application may already be using an existing PKI and wants to use it to augment ZRTP.

To handle these cases, ZRTP allows for an OPTIONAL signature feature, which allows the SAS to be checked without human participation. The SAS MAY be signed and the signature sent inside the Confirm1, Confirm2 (**Figure 10**), or SASrelay (**Figure 16**) messages. The **signature type**, length of the signature, and the key used to create the signature (or a link to it) are all sent along with the signature. The signature is calculated across the entire SAS hash result (sashash), from which the sasvalue was derived. The signatures exchanged in the encrypted Confirm1, Confirm2, or SASrelay messages MAY be used to authenticate the ZRTP exchange. A signature may be sent only in the initial media stream in a DH or ECDH ZRTP exchange, not in Multistream mode.

Although the signature is sent, the material that is signed, the sashash, is not sent with it in the Confirm message, since both parties have already independently calculated the sashash. That is not the case for the SASrelay message, which must relay the sashash.

To avoid unnecessary signature calculations, a signature SHOULD NOT be sent if the other ZRTP endpoint did not set the (S) flag in the **Hello message**.

Note that the choice of hash algorithm used in the digital signature is independent of the

hash used in the sashash. The sashash is determined by the negotiated **Hash Type**, while the hash used by the digital signature is separately defined by the digital signature algorithm. For example, the sashash may be based on SHA-256, while the digital signature might use SHA-384, if an ECDSA P-384 key is used.

If the sashash (which is always truncated to 256 bits) is shorter than the signature hash, the security is not weakened because the hash commitment precludes the attacker from searching for sashash collisions.

ECDSA algorithms may be used with either OpenPGP-formatted keys, or X.509v3 certificates. If the ZRTP key exchange is ECDH, and the SAS is signed, then the signature SHOULD be ECDSA, and SHOULD use the same size curve as the ECDH exchange if an ECDSA key of that size is available.

If a ZRTP endpoint supports incoming signatures (evidenced by setting the (S) flag in the Hello message), it SHOULD be able to parse signatures from the other endpoint in OpenPGP format and MUST be able to parse them in X.509v3 format. If the incoming signature is in an unsupported format, or the trust model does not lead to a trusted introducer or a trusted certificate authority (CA), another authentication method may be used if available, such as the SAS compare, or a cached shared secret from a previous session. If none of these methods are available, it is up to the ZRTP user agent and the user to decide whether to proceed with the call, after the user is informed.

Both ECDSA and DSA **[FIPS-186-3]** have a feature that allows most of the signature calculation to be done in advance of the session, reducing latency during call setup. This is useful for low-power mobile handsets.

ECDSA is preferred because it has compact keys as well as compact signatures. If the signature along with its public key certificate are insufficiently compact, the Confirm message may become too long for the maximum transmission unit (MTU) size, and UDP fragmentation may result. Some firewalls and NATs may discard fragmented UDP packets, which would cause the ZRTP exchange to fail. It is RECOMMENDED that a ZRTP endpoint avoid sending signatures if they would cause UDP fragmentation. For a discussion on MTU size and PMTU discovery, see **[RFC1191]** and **[RFC1981]**.

From a packet-size perspective, ECDSA and DSA both produce equally compact signatures for a given signature strength. DSA keys are much bigger than ECDSA keys, but in the case of OpenPGP signatures, the public key is not sent along with the signature.

All signatures generated MUST use only NIST-approved hash algorithms, and MUST avoid using SHA1. This applies to both OpenPGP and X.509v3 signatures. NIST-approved hash algorithms are found in **[FIPS-180-3]** or its SHA-3 successor. All ECDSA curves used throughout this spec are over prime fields, drawn from Appendix D.1.2 of **[FIPS-186-3]**.

TOC

7.2.1. OpenPGP Signatures

If the **SAS Signature Type** specifies an OpenPGP signature ("PGP "), the signature-related fields are arranged as follows.

The first field after the 4-octet Signature Type Block is the OpenPGP signature. The format of this signature and the algorithms that create it are specified by **[RFC4880]**. The signature is comprised of a complete OpenPGP version 4 signature in binary form (not Radix-64), as specified in RFC 4880, Section 5.2.3, enclosed in the full OpenPGP packet syntax. The length of the OpenPGP signature is parseable from the signature, and depends on the type and length of the signing key.

If OpenPGP signatures are supported, an implementation SHOULD NOT generate signatures using any other signature algorithm except DSA or ECDSA (ECDSA is a reserved algorithm type in RFC 4880), but MAY accept other signature types from the other party. DSA signatures with keys shorter than 2048 bits or longer than 3072 bits MUST NOT be generated.

Implementers should be aware that ECDSA signatures for OpenPGP are expected to become available when the work in progress **[ECC-OpenPGP]** becomes an RFC. Any use of ECDSA signatures in ZRTP SHOULD NOT generate signatures using ECDSA key sizes other than P-224, P-256, and P-384, as defined in **[FIPS-186-3]**.

RFC 4880, Section 5.2.3.18, specifies a way to embed, in an OpenPGP signature, a URI of the preferred key server. The URI should be fully specified to obtain the public key of the signing key that created the signature. This URI MUST be present. It is up to the recipient of the signature to obtain the public key of the signing key and determine its validity status using the OpenPGP trust model discussed in **[RFC4880]**.

The contents of **Figure 20** lie inside the encrypted region of the **Confirm message** or the **SASrelay message**.

The total length of all the material in **Figure 20**, including the key server URI, must not exceed 511 32-bit words (2044 octets). This length, in words, is stored in the signature length field in the Confirm or SASrelay message containing the signature. It is desirable to avoid UDP fragmentation, so the URI should be kept short.

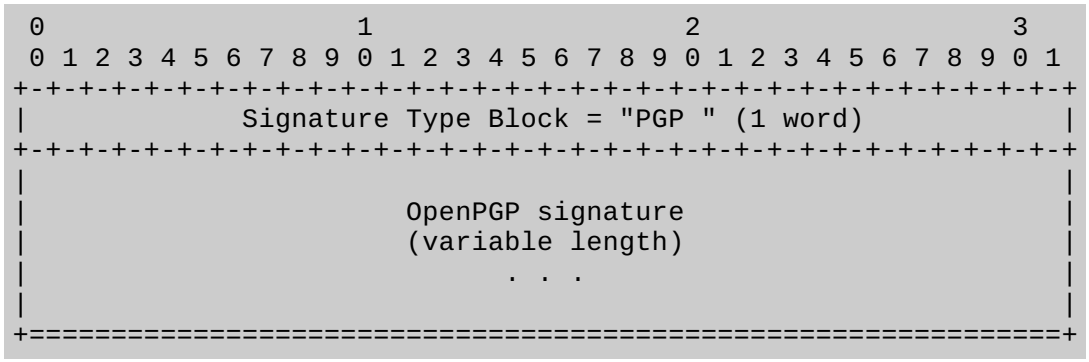


Figure 20: OpenPGP Signature Format

TOC

7.2.2. ECDSA Signatures with X.509v3 Certs

If the **SAS Signature Type** is "X509", the ECDSA signature-related fields are arranged as follows.

The first field after the 4-octet Signature Type Block is the DER encoded X.509v3 certificate (the signed public key) of the ECDSA signing key that created the signature. The format of this certificate is specified by the NSA's **Suite B Certificate and CRL Profile** [RFC5759].

Following the X.509v3 certificate at the next word boundary is the ECDSA signature itself. The size of this field depends on the size and type of the public key in the aforementioned certificate. The format of this signature and the algorithms that create it are specified by **[FIPS-186-3]**. The signature is comprised of the ECDSA signature output parameters (r, s) in binary form, concatenated, in network byte order, with no truncation of leading zeros. The first half of the signature is r and the second half is s. If ECDSA P-256 is specified, the signature fills 16 words (64 octets), 32 octets each for r and s. If ECDSA P-384 is specified, the signature fills 24 words (96 octets), 48 octets each for r and s.

It is up to the recipient of the signature to use information in the certificate and path discovery mechanisms to trace the chain back to the root CA. It is recommended that end user certificates issued for secure telephony should contain appropriate path discovery links to facilitate this.

Figure 21 shows a certificate and an ECDSA signature. All this material lies inside the encrypted region of the **Confirm message** or the **SASrelay message**.

The total length of all the material in **Figure 21**, including the X.509v3 certificate, must not exceed 511 32-bit words (2044 octets). This length, in words, is stored in the signature length field in the Confirm or SASrelay message containing the signature. It is desirable to avoid UDP fragmentation, so the certificate material should be kept to a much smaller size than this. End user certs issued for this purpose should minimize the size of extraneous material such as legal notices.

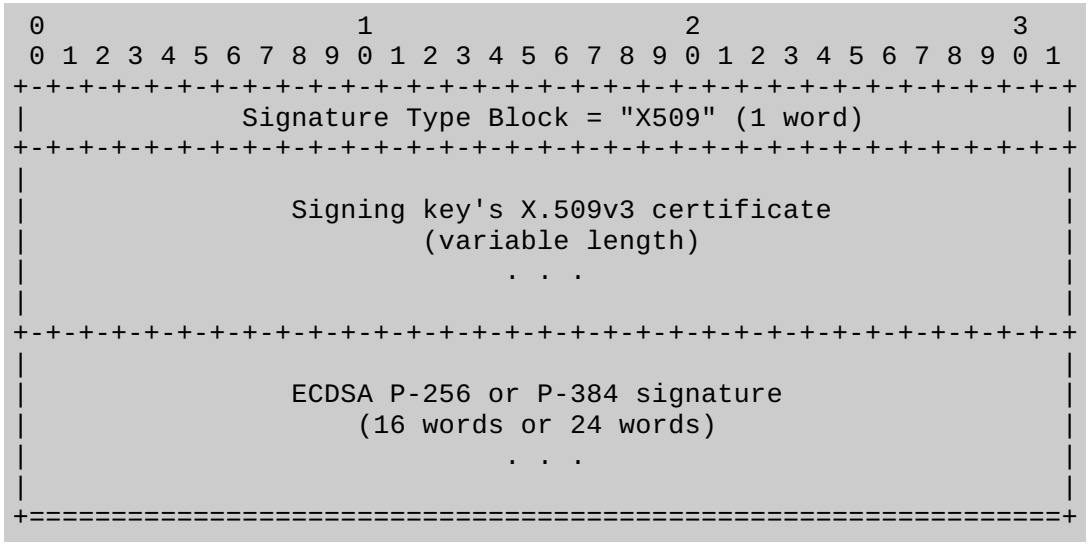


Figure 21: X.509v3 ECDSA Signature Format

TOC

7.2.3. Signing the SAS without a PKI

It's not strictly necessary to use a PKI to back the public key that signs the SAS. For example, it is possible to use a self-signed X.509v3 certificate or an OpenPGP key that is not signed by any other key. In this scenario, the same key continuity technique used by **SSH** [RFC4251] may be used. The public key is cached locally the first time it is encountered, and when the same public key is encountered again in subsequent sessions, it's deemed not to be a MiTM attack. If there is no MiTM attack in the first session, there cannot be a MiTM attack in any subsequent session. This is exactly how SSH does it.

Of course, the security rests on the assumption that the MiTM did not attack in the first session. That assumption seems to work most of the time in the SSH world. The user would have to be warned the first time a public key is encountered, just as in SSH. If possible, the SAS should be checked before the user consents to caching the new public key. If the SAS matches in the first session, there is no MiTM, and it's safe to cache the public key. If no SAS comparison is possible, it's up to the user, or up to the application, to decide whether to take a leap of faith and proceed. That's how SSH works most of the time, because SSH users don't have the chance to verbally compare an SAS with anyone.

For a phone that is SIP-registered to a PBX, it may be provisioned with the public key of the PBX, using a trusted automated provisioning process. Even without a PKI, the phone knows that the public key is the correct one, since it was provisioned into the phone by a trusted provisioning mechanism. This makes it easy for the phone to access several automated services commonly offered by a PBX, such as voice mail or a conference bridge, where there is no human at the PBX to do a verbal SAS compare. The same provisioning may be used to preload the pbxsecret into the phone, which is discussed in **Section 7.3.1**.

TOC

7.3. Relaying the SAS through a PBX

ZRTP is designed to use end-to-end encryption. The two parties' verbal comparison of the short authentication string (SAS) depends on this assumption. But in some PBX environments, such as Asterisk, there are usage scenarios that have the PBX acting as a trusted MiTM, which means there are two back-to-back ZRTP connections with separate session keys and separate SASs.

For example, imagine that Bob has a ZRTP-enabled VoIP phone that has been registered with his company's PBX, so that it is regarded as an extension of the PBX. Alice, whose phone is not associated with the PBX, might dial the PBX from the outside, and a ZRTP connection is negotiated between her phone and the PBX. She then selects Bob's extension from the company directory in the PBX. The PBX makes a call to Bob's phone (which might be offsite, many miles away from the PBX through the Internet) and a separate ZRTP connection is negotiated between the PBX and Bob's phone. The two ZRTP sessions have different session

keys and different SASs, which would render the SAS useless for verbal comparison between Alice and Bob. They might even mistakenly believe that a wiretapper is present because of the SAS mismatch, causing undue alarm.

ZRTP has a mechanism for solving this problem by having the PBX relay the Alice/PBX SAS to Bob, sending it through to Bob in a special SASrelay message as defined in **Section 5.13**, which is sent after the PBX/Bob ZRTP negotiation is complete, after the Confirm messages. Only the PBX, acting as a special trusted MiTM (trusted by the recipient of the SASrelay message), will relay the SAS. The SASrelay message protects the relayed SAS from tampering via an included MAC, similar to how the Confirm message is protected. Bob's ZRTP-enabled phone accepts the relayed SAS for rendering only because Bob's phone had previously been configured to trust the PBX. This special trusted relationship with the PBX can be established through a special **security enrollment procedure**. After that enrollment procedure, the PBX is treated by Bob as a special trusted MiTM. This results in Alice's SAS being rendered to Bob, so that Alice and Bob may verbally compare them and thus prevent a MiTM attack by any other untrusted MiTM.

A real "bad-guy" MiTM cannot exploit this protocol feature to mount a MiTM attack and relay Alice's SAS to Bob, because Bob has not previously carried out a special registration ritual with the bad guy. The relayed SAS would not be rendered by Bob's phone, because it did not come from a trusted PBX. The recognition of the special trust relationship is achieved with the prior establishment of a special shared secret between Bob and his PBX, which is called pbxsecret (defined in **Section 7.3.1**), also known as the trusted MiTM key.

The trusted MiTM key can be stored in a special cache at the time of the initial enrollment (which is carried out only once for Bob's phone), and Bob's phone associates this key with the ZID of the PBX, while the PBX associates it with the ZID of Bob's phone. After the enrollment has established and stored this trusted MiTM key, it can be detected during subsequent ZRTP session negotiations between the PBX and Bob's phone, because the PBX and the phone MUST pass the hash of the trusted MiTM key in the DH message. It is then used as part of the key agreement to calculate s0.

The PBX can determine whether it is trusted by the ZRTP user agent of a phone. The presence of a shared trusted MiTM key in the key negotiation sequence indicates that the phone has been enrolled with this PBX and therefore trusts it to act as a trusted MiTM. During a key agreement with two other ZRTP endpoints, the PBX may have a shared trusted MiTM key with both endpoints, only one endpoint, or neither endpoint. If the PBX has a shared trusted MiTM key with neither endpoint, the PBX MUST NOT relay the SAS. If the PBX has a shared trusted MiTM key with only one endpoint, the PBX MUST relay the SAS from one party to the other by sending an SASrelay message to the endpoint with which it shares a trusted MiTM key. If the PBX has a (separate) shared trusted MiTM key with each of the endpoints, the PBX MUST relay the SAS to only one endpoint, not both endpoints.

> Note: In the case of a PBX sharing trusted MiTM keys with both endpoints, it does not matter which endpoint receives the relayed SAS as long as only one endpoint receives it.

The relayed SAS fields contain the SAS rendering type and the complete sashash. The receiver absolutely MUST NOT render the relayed SAS if it does not come from a specially trusted ZRTP endpoint. The security of the ZRTP protocol depends on not rendering a relayed SAS from an untrusted MiTM, because it may be relayed by a MiTM attacker. See the SASrelay message definition (**Figure 16**) for further details.

To ensure that both Alice and Bob will use the same SAS rendering scheme after the keys are negotiated, the PBX also sends the SASrelay message to the unenrolled party (which does not regard this PBX as a trusted MiTM), conveying the SAS rendering scheme, but not the sashash, which it sets to zero. The unenrolled party will ignore the relayed SAS field, but will use the specified SAS rendering scheme.

It is possible to route a call through two ZRTP-enabled PBXs using this scheme. Assume Alice is a ZRTP endpoint who trusts her local PBX in Atlanta, and Bob is a ZRTP endpoint who trusts his local PBX in Biloxi. The call is routed from Alice to the Atlanta PBX to the Biloxi PBX to Bob. Atlanta would relay the Atlanta-Biloxi SAS to Alice because Alice is enrolled with Atlanta, and Biloxi would relay the Atlanta-Biloxi SAS to Bob because Bob is enrolled with Biloxi. The two PBXs are not assumed to be enrolled with each other in this example. Both Alice and Bob would view and verbally compare the same relayed SAS, the Atlanta-Biloxi SAS. No more than two trusted MiTM nodes can be traversed with this relaying scheme. This behavior is extended to two PBXs that are enrolled with each other, via this rule: In the case of a PBX sharing trusted MiTM keys with both endpoints (i.e., both enrolled with this PBX),

one of which is another PBX (evidenced by the M-flag) and one of which is a non-PBX, the MiTM PBX must always relay the PBX-to-PBX SAS to the non-PBX endpoint.

A ZRTP endpoint phone that trusts a PBX to act as a trusted MiTM is effectively delegating its own policy decisions of algorithm negotiation to the PBX.

When a PBX is between two ZRTP endpoints and is terminating their media streams at the PBX, the PBX presents its own ZID to the two parties, eclipsing the ZIDs of the two parties from each other. For example, if several different calls are routed through such a PBX to several different ZRTP-enabled phones behind the PBX, only a single ZID is presented to the calling party in every case -- the ZID of the PBX itself.

The next section describes the initial enrollment procedure that establishes a special shared secret, a trusted MiTM key, between a PBX and a phone, so that the phone will learn to recognize the PBX as a trusted MiTM.

TOC

7.3.1. PBX Enrollment and the PBX Enrollment Flag

Both the PBX and the endpoint need to know when enrollment is taking place. One way of doing this is to set up an enrollment extension on the PBX that a newly configured endpoint would call and establish a ZRTP session. The PBX would then play audio media that offers the user an opportunity to configure his phone to trust this PBX as a trusted MiTM. The PBX calculates and stores the trusted MiTM shared secret in its cache and associates it with this phone, indexed by the phone's ZID. The trusted MiTM PBX shared secret is derived from ZRTPSess via the **ZRTP key derivation function** in this manner:

```
pbxsecret = KDF(ZRTPSess, "Trusted MiTM key",
                (ZIDi || ZIDr), 256)
```

The pbxsecret is calculated for the whole ZRTP session, not for each stream within a session, thus the KDF Context field in this case does not include any stream-specific nonce material.

The PBX signals the enrollment process by setting the PBX Enrollment flag (E) in the Confirm message (**Figure 10**). This flag is used to trigger the ZRTP endpoint's user interface to prompt the user to see if it wants to trust this PBX and calculate and store the pbxsecret in the cache. If the user decides to respond by activating the appropriate user interface element (a menu item, checkbox, or button), his ZRTP user agent calculates pbxsecret using the same formula, and saves it in a special cache entry associated with this PBX.

During a PBX enrollment, the GoClear features are disabled. If the (E) flag is set by the PBX, the PBX MUST NOT set the Allow Clear (A) flag. Thus, (E) implies not (A). If a received Confirm message has the (E) flag set, the (A) flag MUST be disregarded and treated as false.

If the user elects not to enroll, perhaps because he dialed a wrong number or does not yet feel comfortable with this PBX, he can simply hang up and not save the pbxsecret in his cache. The PBX will have it saved in the PBX cache, but that will do no harm. The SASrelay scheme does not depend on the PBX trusting the phone. It only depends on the phone trusting the PBX. It is the phone (the user) who is at risk if the PBX abuses its MiTM privileges.

An endpoint MUST NOT store the pbxsecret in the cache without explicit user authorization.

After this enrollment process, the PBX and the ZRTP-enabled phone both share a secret that enables the phone to recognize the PBX as a trusted MiTM in future calls. This means that when a future call from an outside ZRTP-enabled caller is relayed through the PBX to this phone, the phone will render a relayed SAS from the PBX. If the SASrelay message comes from a MiTM that does not know the pbxsecret, the phone treats it as a bad-guy MiTM, and refuses to render the relayed SAS. Regardless of which party initiates any future phone calls through the PBX, the enrolled phone or the outside phone, the PBX will relay the SAS to the enrolled phone.

This enrollment procedure is designed primarily for phones that are already associated with the PBX -- enterprise phones that are "behind" the PBX. It is not intended for the countless outside phones that are not registered to this PBX's SIP server. It should be regarded as part

of the installation and provisioning process for a new phone in the organization.

There are more streamlined methods to configure ZRTP user agents to trust a PBX. In large scale deployments, the pbxsecret may be configured into the phone by an automated provisioning process, which may be less burdensome for the users and less error prone. This specification does not require a manual enrollment process. Any process that results in a pbxsecret to be computed and shared between the PBX and the phone will suffice, as long as the user is made aware that this puts the PBX in a position to wiretap the calls.

It is recommended that a ZRTP client not proceed with the PBX enrollment procedure without evidence that a MiTM attack is not taking place during the enrollment session. It would be especially damaging if a MiTM tricks the client into enrolling with the wrong PBX. That would enable the malevolent MiTM to wiretap all future calls without arousing suspicion, because he would appear to be trusted.

TOC

8. Signaling Interactions

This section discusses how ZRTP, SIP, and SDP work together.

Note that ZRTP may be implemented without coupling with the SIP signaling. For example, ZRTP can be implemented as a "bump in the wire" or as a "bump in the stack" in which RTP sent by the SIP User Agent (UA) is converted to ZRTP. In these cases, the SIP UA will have no knowledge of ZRTP. As a result, the signaling path discovery mechanisms introduced in this section should not be definitive -- they are a hint. Despite the absence of an indication of ZRTP support in an offer or answer, a ZRTP endpoint SHOULD still send Hello messages.

ZRTP endpoints that have control over the signaling path include a ZRTP SDP attributes in their SDP offers and answers. The ZRTP attribute, a=zrtp-hash, is used to indicate support for ZRTP and to convey a hash of the Hello message. The hash is computed according to **Section 8.1**.

Aside from the advantages described in **Section 8.1**, there are a number of potential uses for this attribute. It is useful when signaling elements would like to know when ZRTP may be utilized by endpoints. It is also useful if endpoints support multiple methods of SRTP key management. The ZRTP attribute can be used to ensure that these key management approaches work together instead of against each other. For example, if only one endpoint supports ZRTP, but both support another method to key SRTP, then the other method will be used instead. When used in parallel, an SRTP secret carried in an a=keymgt **[RFC4567]** or a=crypto **[RFC4568]** attribute can be used as a shared secret for the srtps computation defined in **Section 8.2**. The ZRTP attribute is also used to signal to an intermediary ZRTP device not to act as a ZRTP endpoint, as discussed in **Section 10**.

The a=zrtp-hash attribute can only be included in the SDP at the media level since Hello messages sent in different media streams will have unique hashes.

The ABNF for the ZRTP attribute is as follows:

```
zrtp-attribute   = "a=zrtp-hash:" zrtp-version zrtp-hash-value

zrtp-version     = token

zrtp-hash-value  = 1*(HEXDIG)
```

Here's an example of the ZRTP attribute in an initial SDP offer or answer used at the media level, using the <allOneLine> convention defined in RFC 4475, Section 2.1 **[RFC4475]**:

```
v=0
o=bob 2890844527 2890844527 IN IP4 client.biloxi.example.com
s=
c=IN IP4 client.biloxi.example.com
t=0 0
m=audio 3456 RTP/AVP 97 33
a=rtpmap:97 iLBC/8000
```

```
  a=rtpmap:33 no-op/8000
<allOneLine>
  a=zrtp-hash:1.10 fe30efd02423cb054e50efd0248742ac7a52c8f91bc2
  df881ae642c371ba46df
</allOneLine>
```

A mechanism for carrying this same zrtp-hash information in the Jingle signaling protocol is defined in **[XEP-0262]**.

It should be safe to send ZRTP messages even when there is no evidence in the signaling that the other party supports it, because ZRTP has been designed to be clearly different from RTP, having a similar structure to STUN packets sent during an ICE exchange.

8.1. Binding the Media Stream to the Signaling Layer via the Hello Hash

TOC

Tying the media stream to the signaling channel can help prevent a third party from inserting false media packets. If the signaling layer contains information that ties it to the media stream, false media streams can be rejected.

To accomplish this, the entire Hello message (**Figure 3**) is hashed, using the hash algorithm defined in **Section 5.1.2.2**. The ZRTP packet framing from **Figure 2** is not included in the hash. The resulting hash image is made available without truncation to the signaling layer, where it is transmitted as a hexadecimal value in the SIP channel using the SDP attribute a=zrtp-hash, defined in this specification. Assuming **Section 5.1.2.2** defines a 256-bit hash length, the a=zrtp-hash field in the SDP attribute carries 64 hexadecimal digits. Each media stream (audio or video) will have a separate Hello message, and thus will require a separate a=zrtp-hash in an SDP attribute. The recipient of the SIP/SDP message can then use this hash image to detect and reject false Hello messages in the media channel, as well as identify which media stream is associated with this SIP call. Each Hello message hashes uniquely, because it contains the H3 field derived from a random nonce, defined in **Section 9**.

The Hello Hash as an SDP attribute is not a REQUIRED feature, because some ZRTP endpoints do not have the ability to add SDP attributes to the signaling. For example, if ZRTP is implemented in a hardware bump-in-the-wire device, it might only have the ability to modify the media packets, not the SIP packets, especially if the SIP packets are integrity protected and thus cannot be modified on the wire. If the SDP has no hash image of the ZRTP Hello message, the recipient's ZRTP user agent cannot check it, and thus will not be able to reject Hello messages based on this hash.

After the Hello Hash is used to properly identify the ZRTP Hello message as belonging to this particular SIP call, the rest of the ZRTP message sequence is protected from false packet injection by other protection mechanisms, such as the hash chaining mechanism defined in **Section 9**.

An attacker who controls only the signaling layer, such as an uncooperative VoIP service provider, may be able to deny service by corrupting the hash of the Hello message in the SDP attribute, which would force ZRTP to reject perfectly good Hello messages. If there is reason to believe this is happening, the ZRTP endpoint MAY allow Hello messages to be accepted that do not match the hash image in the SDP attribute.

Even in the absence of SIP integrity protection, the inclusion of the a=zrtp-hash SDP attribute, when coupled with the hash chaining mechanism defined in **Section 9**, meets the R-ASSOC requirement in the **Media Security Requirements** [RFC5479], which requires:

> ...a mechanism for associating key management messages with both the signaling traffic that initiated the session and with protected media traffic. It is useful to associate key management messages with call signaling messages, as this allows the SDP offerer to avoid performing CPU-consuming operations (e.g., Diffie-Hellman or public key operations) with attackers that have not seen the signaling messages.

The a=zrtp-hash SDP attribute becomes especially useful if the SDP is integrity-protected end-to-end by **SIP Identity** [RFC4474] or better still, Dan Wing's **SIP Identity using Media Path** [SIP-IDENTITY]. This leads to an ability to stop MiTM attacks independent of ZRTP's SAS

mechanism, as explained in **Section 8.1.1**.

8.1.1. Integrity-Protected Signaling Enables Integrity-Protected DH Exchange

TOC

If and only if the signaling path and the SDP is protected by some form of end-to-end integrity protection, such as one of the abovementioned mechanisms, so that it can guarantee delivery of the a=zrtp-hash attribute without any tampering by a third party, and if there is good reason to trust the signaling layer to protect the interests of the end user, it is possible to authenticate the key exchange and prevent a MiTM attack. This can be done without requiring the users to verbally compare the SAS, by using the hash chaining mechanism defined in **Section 9** to provide a series of MAC keys that protect the entire ZRTP key exchange. Thus, an end-to-end integrity-protected signaling layer automatically enables an integrity-protected Diffie-Hellman exchange in ZRTP, which in turn means immunity from a MiTM attack. Here's how it works.

The integrity-protected SIP SDP contains a hash commitment to the entire Hello message. The Hello message contains H3, which provides a hash commitment for the rest of the hash chain H0-H2 (**Section 9**). The Hello message is protected by a 64-bit MAC, keyed by H2. The Commit message is protected by a 64-bit MAC, keyed by H1. The DHPart1 or DHPart2 messages are protected by a 64-bit MAC, keyed by H0. The MAC protecting the Confirm messages is computed by a different MAC key derived from the resulting key agreement. Each message's MAC is checked when the MAC key is received in the next message. If a bad MAC is discovered, it MUST be treated as a security exception indicating a MiTM attack, perhaps by logging or alerting the user, and MUST NOT be treated as a random error. Random errors are already discovered and quietly rejected by bad CRCs (**Figure 2**).

The Hello message must be assembled before any hash algorithms are negotiated, so an implicit predetermined hash algorithm and MAC algorithm (both defined in **Section 5.1.2.2**) must be used. All of the aforementioned MACs keyed by the hashes in the aforementioned hash chain MUST be computed with the MAC algorithm defined in **Section 5.1.2.2**, with the MAC truncated to 64 bits.

The **Media Security Requirements** [RFC5479] R-EXISTING requirement can be fully met by leveraging a certificate-backed PKI in the signaling layer to integrity protect the delivery of the a=zrtp-hash SDP attribute. This would thereby protect ZRTP against a MiTM attack, without requiring the user to check the SAS, without adding any explicit signatures or signature keys to the ZRTP key exchange and without any extra public key operations or extra packets.

Without an end-to-end integrity-protection mechanism in the signaling layer to guarantee delivery of the a=zrtp-hash SDP attribute without modification by a third party, these MACs alone will not prevent a MiTM attack. In that case, ZRTP's built-in SAS mechanism will still have to be used to authenticate the key exchange. At the time of this writing, very few deployed VoIP clients offer a fully implemented SIP stack that provides end-to-end integrity protection for the delivery of SDP attributes. Also, end-to-end signaling integrity becomes more problematic if **E.164 numbers** [RFC3824] are used in SIP. Thus, real-world implementations of ZRTP endpoints will continue to depend on SAS authentication for quite some time. Even after there is widespread availability of SIP user agents that offer integrity protected delivery of SDP attributes, many users will still be faced with the fact that the signaling path may be controlled by institutions that do not have the best interests of the end user in mind. In those cases, SAS authentication will remain the gold standard for the prudent user.

Even without SIP integrity protection, the **Media Security Requirements** [RFC5479] R-ACT-ACT requirement can be met by ZRTP's SAS mechanism. Although ZRTP may benefit from an integrity-protected SIP layer, it is fortunate that ZRTP's self-contained MiTM defenses do not actually require an integrity-protected SIP layer. ZRTP can bypass the delays and problems that SIP integrity faces, such as E.164 number usage, and the complexity of building and maintaining a PKI.

In contrast, **DTLS-SRTP** [RFC5764] appears to depend heavily on end-to-end integrity protection in the SIP layer. Further, DTLS-SRTP must bear the additional cost of a signature calculation of its own, in addition to the signature calculation the SIP layer uses to achieve its integrity protection. ZRTP needs no signature calculation of its own to leverage the signature calculation carried out in the SIP layer.

8.2. Deriving the SRTP Secret (srtps) from the Signaling Layer

TOC

The shared secret calculations defined in **Section 4.3** make use of the SRTP secret (srtps), if it is provided by the signaling layer.

It is desirable for only one SRTP key negotiation protocol to be used, and that protocol should be ZRTP. But in the event the signaling layer negotiates its own SRTP master key and salt, using the SDP Security Descriptions (**SDES** [RFC4568]) or **[RFC4567]**, it can be passed from the signaling to the ZRTP layer and mixed into ZRTP's own shared secret calculations, without compromising security by creating a dependency on the signaling for media encryption.

ZRTP computes srtps from the SRTP master key and salt parameters provided by the signaling layer in this manner, truncating the result to 256 bits:

```
srtps = KDF(SRTP master key, "SRTP Secret",
            (ZIDi || ZIDr || SRTP master salt), 256)
```

It is expected that the srtps parameter will be rarely computed or used in typical ZRTP endpoints, because it is likely and desirable that ZRTP will be the sole means of negotiating SRTP keys, needing no help from **[RFC4568]** or **[RFC4567]**. If srtps is computed, it will be stored in the auxiliary shared secret auxsecret, defined in **Section 4.3** and used in **Section 4.3.1**.

8.3. Codec Selection for Secure Media

TOC

Codec selection is negotiated in the signaling layer. If the signaling layer determines that ZRTP is supported by both endpoints, this should provide guidance in codec selection to avoid variable bitrate (VBR) codecs that leak information.

When voice is compressed with a VBR codec, the packet lengths vary depending on the types of sounds being compressed. This leaks a lot of information about the content even if the packets are encrypted, regardless of what encryption protocol is used **[Wright1]**. It is RECOMMENDED that VBR codecs be avoided in encrypted calls. It is not a problem if the codec adapts the bitrate to the available channel bandwidth. The vulnerable codecs are the ones that change their bitrate depending on the type of sound being compressed.

It also appears that voice activity detection (VAD) leaks information about the content of the conversation, but to a lesser extent than VBR. This effect can be mitigated by lengthening the VAD hangover time by a random amount between 1 and 2 seconds, if this is feasible in your application. Only short bursts of speech would benefit from lengthening the VAD hangover time.

The security problems of VBR and VAD are addressed in detail by the guidelines in **[VBR-AUDIO]**. It is RECOMMENDED that ZRTP endpoints follow these guidelines.

9. False ZRTP Packet Rejection

TOC

An attacker who is not in the media path may attempt to inject false ZRTP protocol packets, possibly to effect a denial-of-service attack or to inject his own media stream into the call. VoIP, by its nature, invites various forms of denial-of-service attacks and requires protocol features to reject such attacks. While bogus SRTP packets may be easily rejected via the SRTP auth tag field, that can only be applied after a key agreement is completed. During the ZRTP key negotiation phase, other false packet rejection mechanisms are needed. One such mechanism is the use of the total_hash in the final shared secret calculation, but that can only detect false packets after performing the computationally expensive Diffie-Hellman calculation.

A lot of work has been done on the analysis of denial-of-service attacks, especially from

attackers who are not in the media path. Such an attacker might inject false ZRTP packets to force a ZRTP endpoint to engage in an endless series of pointless and expensive DH calculations. To detect and reject false packets cheaply and rapidly as soon as they are received, ZRTP uses a one-way hash chain, which is a series of successive hash images. Before each session, the following values are computed:

```
H0 = 256-bit random nonce (different for each party)

H1 = hash (H0)

H2 = hash (H1)

H3 = hash (H2)
```

This one-way hash chain MUST use the hash algorithm defined in **Section 5.1.2.2**, truncated to 256 bits. Each 256-bit hash image is the preimage of the next, and the sequence of images is sent in reverse order in the ZRTP packet sequence. The hash image H3 is sent in the Hello message, H2 is sent in the Commit message, H1 is sent in the DHPart1 or DHPart2 messages, and H0 is sent in the Confirm1 or Confirm2 messages. The initial random H0 nonces that each party generates MUST be unpredictable to an attacker and unique within a ZRTP session, which thereby forces the derived hash images H1-H3 to also be unique and unpredictable.

The recipient checks if the packet has the correct hash preimage, by hashing it and comparing the result with the hash image for the preceding packet. Packets that contain an incorrect hash preimage MUST NOT be used by the recipient, but they MAY be processed as security exceptions, perhaps by logging or alerting the user. As long as these bogus packets are not used, and correct packets are still being received, the protocol SHOULD be allowed to run to completion, thereby rendering ineffective this denial-of-service attack.

Note that since H2 is sent in the Commit message, and the initiator does not receive a Commit message, the initiator computes the responder's missing H2 by hashing the responder's H1. An analogous interpolation is performed by both parties to handle the skipped DHPart1 and DHPart2 messages in **Preshared** or **Multistream** modes.

Because these hash images alone do not protect the rest of the contents of the packet they reside in, this scheme assumes the attacker cannot modify the packet contents from a legitimate party, which is a reasonable assumption for an attacker who is not in the media path. This covers an important range of denial-of-service attacks. For dealing with the remaining set of attacks that involve packet modification, other mechanisms are used, such as the total_hash in the final shared secret calculation, and the hash commitment in the Commit message.

Hello messages injected by an attacker may be detected and rejected by the inclusion of a hash of the Hello message in the signaling, as described in **Section 8**. This mechanism requires that each Hello message be unique, and the inclusion of the H3 hash image meets that requirement.

If and only if an integrity-protected signaling channel is available, the MACs that are keyed by this hash chaining scheme can be used to authenticate the entire ZRTP key exchange, and thereby prevent a MiTM attack, without relying on the users verbally comparing the SAS. See **Section 8.1.1** for details.

Some ZRTP user agents allow the user to manually switch to clear mode (via the GoClear message) in the middle of a secure call, and then later initiate secure mode again. Many consumer client products will omit this feature, but those that allow it may return to secure mode again in the same media stream. Although the same chain of hash images will be reused and thus rendered ineffective the second time, no real harm is done because the new SRTP session keys will be derived in part from a cached shared secret, which was safely protected from the MiTM in the previous DH exchange earlier in the same session.

---

10. Intermediary ZRTP Devices

TOC

This section discusses the operation of a ZRTP endpoint that is actually an intermediary. For

example, consider a device that proxies both signaling and media between endpoints. There are three possible ways in which such a device could support ZRTP.

An intermediary device can act transparently to the ZRTP protocol. To do this, a device MUST pass non-RTP protocols multiplexed on the same port as RTP (to allow ZRTP and STUN). This is the RECOMMENDED behavior for intermediaries as ZRTP and SRTP are best when done end-to-end.

An intermediary device could implement the ZRTP protocol and act as a ZRTP endpoint on behalf of non-ZRTP endpoints behind the intermediary device. The intermediary could determine on a call-by-call basis whether the endpoint behind it supports ZRTP based on the presence or absence of the ZRTP SDP attribute flag (a=zrtp-hash). For non-ZRTP endpoints, the intermediary device could act as the ZRTP endpoint using its own ZID and cache. This approach SHOULD only be used when there is some other security method protecting the confidentiality of the media between the intermediary and the inside endpoint, such as IPsec or physical security.

The third mode, which is NOT RECOMMENDED, is for the intermediary device to attempt to back-to-back the ZRTP protocol. The only exception to this case is where the intermediary device is a trusted element providing services to one of the endpoints -- e.g., a Private Branch Exchange or PBX. In this mode, the intermediary would attempt to act as a ZRTP endpoint towards both endpoints of the media session. This approach MUST NOT be used except as described in **Section 7.3** as it will always result in a detected MiTM attack and will generate alarms on both endpoints and likely result in the immediate termination of the session. The PBX MUST uses a single ZID for all endpoints behind it.

In cases where centralized media mixing is taking place, the SAS will not match when compared by the humans. This situation can sometimes be known in the SIP signaling by the presence of the isfocus feature tag **[RFC4579]**. As a result, when the isfocus feature tag is present, the DH exchange can be authenticated by the mechanism defined in **Section 8.1.1** or by **validating signatures** in the Confirm or SASrelay messages. For example, consider an audio conference call with three participants Alice, Bob, and Carol hosted on a conference bridge in Dallas. There will be three ZRTP encrypted media streams, one encrypted stream between each participant and Dallas. Each will have a different SAS. Each participant will be able to validate their SAS with the conference bridge by using signatures optionally present in the Confirm messages (described in **Section 7.2**). Or, if the signaling path has end-to-end integrity protection, each DH exchange will have automatic MiTM protection by using the mechanism in **Section 8.1.1**.

SIP feature tags can also be used to detect if a session is established with an automaton such as an Interactive Voice Response (IVR), voicemail system, or speech recognition system. The display of SAS strings to users should be disabled in these cases.

It is possible that an intermediary device acting as a ZRTP endpoint might still receive ZRTP Hello and other messages from the inside endpoint. This could occur if there is another inline ZRTP device that does not include the ZRTP SDP attribute flag. An intermediary acting as a ZRTP endpoint receiving ZRTP Hello and other messages from the inside endpoint MUST NOT pass these ZRTP messages.

TOC

11. The ZRTP Disclosure Flag

There are no back doors defined in the ZRTP protocol specification. The designers of ZRTP would like to discourage back doors in ZRTP-enabled products. However, despite the lack of back doors in the actual ZRTP protocol, it must be recognized that a ZRTP implementer might still deliberately create a rogue ZRTP-enabled product that implements a back door outside the scope of the ZRTP protocol. For example, they could create a product that discloses the SRTP session key generated using ZRTP out-of-band to a third party. They may even have a legitimate business reason to do this for some customers.

For example, some environments have a need to monitor or record calls, such as stock brokerage houses who want to discourage insider trading, or special high-security environments with special needs to monitor their own phone calls. We've all experienced automated messages telling us that "This call may be monitored for quality assurance". A ZRTP endpoint in such an environment might unilaterally disclose the session key to someone monitoring the call. ZRTP-enabled products that perform such out-of-band disclosures of the session key can undermine public confidence in the ZRTP protocol, unless

we do everything we can in the protocol to alert the other user that this is happening.

If one of the parties is using a product that is designed to disclose their session key, ZRTP requires them to confess this fact to the other party through a protocol message to the other party's ZRTP client, which can properly alert that user, perhaps by rendering it in a graphical user interface. The disclosing party does this by sending a Disclosure flag (D) in Confirm1 and Confirm2 messages as described in **Section 5.7**.

Note that the intention here is to have the Disclosure flag identify products that are designed to disclose their session keys, not to identify which particular calls are compromised on a call-by-call basis. This is an important legal distinction, because most government sanctioned wiretap regulations require a VoIP service provider to not reveal which particular calls are wiretapped. But there is nothing illegal about revealing that a product is designed to be wiretap-friendly. The ZRTP protocol mandates that such a product "out" itself.

You might be using a ZRTP-enabled product with no back doors, but if your own graphical user interface tells you the call is (mostly) secure, except that the other party is using a product that is designed in such a way that it may have disclosed the session key for monitoring purposes, you might ask him what brand of secure telephone he is using, and make a mental note not to purchase that brand yourself. If we create a protocol environment that requires such back-doored phones to confess their nature, word will spread quickly, and the "invisible hand" of the free market will act. The free market has effectively dealt with this in the past.

Of course, a ZRTP implementer can lie about his product having a back door, but the ZRTP standard mandates that ZRTP-compliant products MUST adhere to the requirement that a back door be confessed by sending the Disclosure flag to the other party.

There will be inevitable comparisons to Steve Bellovin's 2003 April fool joke, when he submitted **RFC 3514** [RFC3514], which defined the "Evil bit" in the IPv4 header, for packets with "evil intent". But we submit that a similar idea can actually have some merit for securing VoIP. Sure, one can always imagine that some implementer will not be fazed by the rules and will lie, but they would have lied anyway even without the Disclosure flag. There are good reasons to believe that it will improve the overall percentage of implementations that at least tell us if they put a back door in their products, and may even get some of them to decide not to put in a back door at all. From a civic hygiene perspective, we are better off with having the Disclosure flag in the protocol.

If an endpoint stores or logs SRTP keys or information that can be used to reconstruct or recover SRTP keys after they are no longer in use (i.e., the session is active), or otherwise discloses or passes SRTP keys or information that can be used to reconstruct or recover SRTP keys to another application or device, the Disclosure flag D MUST be set in the Confirm1 or Confirm2 message.

TOC

11.1. Guidelines on Proper Implementation of the Disclosure Flag

Some implementers have asked for guidance on implementing the Disclosure flag. Some people have incorrectly thought that a connection secured with ZRTP cannot be used in a call center, with voluntary voice recording, or even with a voicemail system. Similarly, some potential users of ZRTP have over considered the protection that ZRTP can give them. These guidelines clarify both concerns.

The ZRTP Disclosure flag only governs the ZRTP/SRTP stream itself. It does not govern the underlying RTP media stream, nor the actual media itself. Consequently, a PBX that uses ZRTP may provide conference calls, call monitoring, call recording, voicemail, or other PBX features and still say that it does not disclose the ZRTP key material. A video system may provide DVR features and still say that it does not disclose the ZRTP key material. The ZRTP Disclosure flag, when not set, means only that the ZRTP cryptographic key material stays within the bounds of the ZRTP subsystem.

If an application has a need to disclose the ZRTP cryptographic key material, the easiest way to comply with the protocol is to set the flag to the proper value. The next easiest way is to overestimate disclosure. For example, a call center that commonly records calls might choose to set the Disclosure flag even though all recording is an analog recording of a call (and thus outside the ZRTP scope) because it sets an expectation with clients that their calls might be recorded.

Note also that the ZRTP Disclosure Flag does not require an implementation to preclude hacking or malware. Malware that leaks ZRTP cryptographic key material does not create a liability for the implementer from non-compliance with the ZRTP specification.

A user of ZRTP should note that ZRTP is not a panacea against unauthorized recording. ZRTP does not and cannot protect against an untrustworthy partner who holds a microphone up to the speaker. It does not protect against someone else being in the room. It does not protect against analog wiretaps in the phone or in the room. It does not mean your partner has not been hacked with spyware. It does not mean that the software has no flaws. It means that the ZRTP subsystem is not knowingly leaking ZRTP cryptographic key material.

---

12. Mapping between ZID and AOR (SIP URI)

TOC

The role of the ZID in the management of the local cache of shared secrets is explained in **Section 4.9**. A particular ZID is associated with a particular ZRTP endpoint, typically a VoIP client. A single SIP URI (also known as an Address-of-Record, or AOR) may be hosted on several different soft VoIP clients, desktop phones, and mobile handsets, and each of them will have a different ZID. Further, a single VoIP client may have several SIP URIs configured into its profiles, but only one ZID. There is not a one-to-one mapping between a ZID and a SIP URI. A single SIP URI may be associated with several ZIDs, and a single ZID may be associated with several SIP URIs on the same client.

Not only that, but ZRTP is independent of which signaling protocol is used. It works equally well with SIP, Jingle, H.323, or any proprietary signaling protocol. Thus, a ZRTP ZID has little to do with SIP, per se, which means it has little to do with a SIP URI.

Even though a ZID is associated with a device, not a human, it is often the case that a ZRTP endpoint is controlled mainly by a particular human. For example, it may be a mobile phone. To get the full benefit of the key continuity features, a local cache entry (and thus a ZID) should be associated with some sort of name of the remote party. That name could be a human name, or it could be made more precise by specifying which ZRTP endpoint he's using. For example "Jon Callas", or "Jon Callas on his iPhone", or "Jon on his iPad", or "Alice on her office phone". These name strings can be stored in the local cache, indexed by ZID, and may have been initially provided by the local user by hand. Or the local cache entry may contain a pointer to an entry in the local address book. When a secure session is established, if a prior session has established a cache entry, and the new session has a matching cache entry indexed by the same ZID, and the SAS has been previously verified, the person's name stored in that cache entry should be displayed.

If the remote ZID originates from a PBX, the displayed name would be the name of that PBX, which might be the name of the company who owns that PBX.

If it is desirable to associate some key material with a particular AOR, **digital signatures** may be used, with public key certificates that associate the signature key with an AOR. If more than one ZRTP endpoint shares the same AOR, they may all use the same signature key and provide the same public key certificate with their signatures.

---

13. IANA Considerations

TOC

This specification defines a new **SDP** [RFC4566] attribute in **Section 8**.

```
Contact name:           Philip Zimmermann <prz@mit.edu>

Attribute name:         "zrtp-hash"

Type of attribute:      Media level

Subject to charset:     Not

Purpose of attribute:   The 'zrtp-hash' indicates that a
                        UA supports the ZRTP protocol and
                        provides a hash of the ZRTP Hello
```

```
                          message.  The ZRTP protocol
                          version number is also specified.

  Allowed attribute values:  Hex
```

---

14. Media Security Requirements

TOC

This section discuses how ZRTP meets all RTP security requirements discussed in the **Media Security Requirements** [RFC5479] document without any dependencies on other protocols or extensions, unlike **DTLS-SRTP** [RFC5764] which requires additional protocols and mechanisms.

R-FORK-RETARGET is met since ZRTP is a media path key agreement protocol.

R-DISTINCT is met since ZRTP uses ZIDs and allows multiple independent ZRTP exchanges to proceed.

R-HERFP is met since ZRTP is a media path key agreement protocol.

R-REUSE is met using the Multistream and Preshared modes.

R-AVOID-CLIPPING is met since ZRTP is a media path key agreement protocol.

R-RTP-CHECK is met since the ZRTP packet format does not pass the RTP validity check.

R-ASSOC is met using the a=zrtp-hash SDP attribute in INVITEs and responses (**Section 8.1**).

R-NEGOTIATE is met using the Commit message.

R-PSTN is met since ZRTP can be implemented in Gateways.

R-PFS is met using ZRTP Diffie-Hellman key agreement methods.

R-COMPUTE is met using the Hello/Commit ZRTP exchange.

R-CERTS is met using the verbal comparison of the SAS.

R-FIPS is met since ZRTP uses only FIPS-approved algorithms in all relevant categories. The authors believe ZRTP is compliant with **[NIST-SP800-56A]**, **[NIST-SP800-108]**, **[FIPS-198-1]**, **[FIPS-180-3]**, **[NIST-SP800-38A]**, **[FIPS-197]**, and **[NSA-Suite-B]**, which should meet the FIPS-140 validation requirements set by **[FIPS-140-2-Annex-A]** and **[FIPS-140-2-Annex-D]**.

R-DOS is met since ZRTP does not introduce any new denial-of-service attacks.

R-EXISTING is met since ZRTP can support the use of certificates or keys.

R-AGILITY is met since the set of hash, cipher, SRTP authentication tag type, key agreement method, SAS type, and signature type can all be extended and negotiated.

R-DOWNGRADE is met since ZRTP has protection against downgrade attacks.

R-PASS-MEDIA is met since ZRTP prevents a passive adversary with access to the media path from gaining access to keying material used to protect SRTP media packets.

R-PASS-SIG is met since ZRTP prevents a passive adversary with access to the signaling path from gaining access to keying material used to protect SRTP media packets.

R-SIG-MEDIA is met using the a=zrtp-hash SDP attribute in INVITEs and responses.

R-ID-BINDING is met using the a=zrtp-hash SDP attribute (**Section 8.1**).

R-ACT-ACT is met using the a=zrtp-hash SDP attribute in INVITEs and responses.

R-BEST-SECURE is met since ZRTP utilizes the RTP/AVP profile and hence best effort SRTP in every case.

R-OTHER-SIGNALING is met since ZRTP can utilize modes in which there is no dependency on the signaling path.

R-RECORDING is met using the ZRTP Disclosure flag.

R-TRANSCODER is met if the transcoder operates as a trusted MitM (i.e., a PBX).

R-ALLOW-RTP is met due to ZRTP's best effort encryption.

TOC

15. Security Considerations

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 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]**, and the GSMK 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.

Some questions have been raised about voice spoofing during the short authentication string (SAS) comparison. 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 MiTM attacker can safely break into a phone conversation and inject his own 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. If the session also includes video (an increasingly common usage scenario), the MiTM may be further deterred by the difficulty of making the lips sync with the voice-spoofed SAS. The PGP word list is designed to make each word phonetically distinct, which also tends to create distinctive lip movements. 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.

A question has been raised regarding the safety of the SAS procedure for people who don't know each other's voices, because it may allow an attack from a MiTM even if he lacks voice impersonation capabilities. This is not as much of a problem as it seems, because it isn't necessary that users recognize each other by their voice. It is only necessary that they detect that the voice used for the SAS procedure doesn't match the voice in the rest of the phone conversation.

Special consideration must be given to secure phone calls with automated systems that cannot perform a verbal SAS comparison between two humans (e.g., a voice mail system). If

a well-functioning PKI is available to all parties, it is recommended that credentials be provisioned at the automated system sufficient to use one of the automatic MiTM detection mechanisms from **Section 8.1.1** or **Section 7.2**. Or rely on a previously established cached shared secret (pbxsecret or rs1 or both), backed by a human-executed SAS comparison during an initial call. Note that it is worse than useless and absolutely unsafe to rely on a robot voice from the remote endpoint to compare the SAS, because a robot voice can be trivially forged by a MiTM. However, a robot voice may be safe to use strictly locally for a different purpose. A ZRTP user agent may render its locally computed SAS to the local user via a robot voice if no visual display is available, provided the user can readily determine that the robot voice is generated locally, not from the remote endpoint.

A popular and field-proven approach to MiTM protection 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 verbally compare 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 (normative language is in **Section 4.3.2**). Voice confirmation is absolutely essential in this situation. 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.

Precautions must be observed when using a trusted MiTM device such as a trusted PBX, as described in **Section 7.3**. Make sure you really trust that this PBX will never be compromised before establishing it as a trusted MiTM, because it is in a position to wiretap calls for any phone that trusts it. It is "licensed" to be in a position to wiretap. You are safer to try to arrange the connection topology to route the media directly between the two ZRTP peers, not through a trusted PBX. Real end-to-end encryption is preferred.

The security of the SAS mechanism depends on the user verifying it verbally with his peer at the other endpoint. There is some risk the user will not be so diligent and may ignore the SAS. For a discussion on how users become habituated to security warnings in the PKI certificate world, see **[Sunshine]**. Part of the problems discussed in that paper are from the habituation syndrome common to most warning messages, and part of them are from the fact that users simply don't understand trust models. Fortunately, ZRTP doesn't need a trust model to use the SAS mechanism, so it's easier for the user to grasp the idea of comparing the SAS verbally with the other party; it's easier than understanding a trust model, at least. Also, the verbal comparison of the SAS gets both users involved, and they will notice a mismatch of the SAS. Also, the ZRTP user agent will know when the SAS has been previously verified because of the **SAS verified flag (V)**, and only ask the user to verify it when needed. After it has been verified once, the key continuity features make it unnecessary to verify it again.

TOC

15.1. Self-Healing Key Continuity Feature

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 can 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 opportunities. Thus, ZRTP quickly recovers from any disclosure of cached key material.

In systems that use a persistent private signature key, such as SSH, the stored signature key is usually protected from disclosure by encryption that requires a user-supplied high-entropy passphrase. This arrangement may be acceptable for a diligent user with a desktop computer sitting in an office with a full ASCII keyboard. But it would be prohibitively inconvenient and unsafe to type a high-entropy passphrase on a mobile phone's numeric keypad while driving a car. Users will reject any scheme that requires the use of a passphrase on such a platform, which means mobile phones carry an elevated risk of compromise of stored key material, and thus would especially benefit from the self-healing aspects of ZRTP's key continuity features.

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 **[NIST-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.

Unlike the long-lived non-updated key material used by SSH, the dynamically updated shared secrets of ZRTP may lose sync if traditional backup/restore mechanisms are used. This limitation is a consequence of the otherwise beneficial aspects of this approach to key continuity, and it is partially mitigated by ZRTP's **built-in cache backup logic**.

---

16. Acknowledgments

TOC

The authors would like to thank Bryce "Zooko" Wilcox-O'Hearn and Colin Plumb for their contributions to the design of this protocol. Also, thanks to Hal Finney, Viktor Krikun, Werner Dittmann, Dan Wing, Sagar Pai, David McGrew, Colin Perkins, Dan Harkins, David Black, Tim Polk, Richard Harris, Roni Even, Jon Peterson, and Robert Sparks for their helpful comments and suggestions. Thanks to Lily Chen at NIST for her assistance in ensuring compliance with NIST SP800-56A and SP800-108.

The use of one-way hash chains to key HMACs in ZRTP is similar to Adrian Perrig's **TESLA protocol** [TESLA].

---

17. References

TOC

17.1. Normative References

TOC

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication," RFC 2104, February 1997 (TXT).

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels," BCP 14, RFC 2119, March 1997 (TXT, HTML, XML).

[RFC3526] Kivinen, T. and M. Kojo, "More Modular Exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE)," RFC 3526, May 2003 (TXT).

[RFC3550] Schulzrinne, H., Casner, S., Frederick, R., and V. Jacobson, "RTP: A Transport Protocol for Real-Time Applications," STD 64, RFC 3550, July 2003 (TXT, PS, PDF).

[RFC3711] Baugher, M., McGrew, D., Naslund, M., Carrara, E., and K. Norrman, "The Secure Real-time Transport Protocol (SRTP)," RFC 3711, March 2004 (TXT).

[RFC4231] Nystrom, M., "Identifiers and Test Vectors for HMAC-SHA-224, HMAC-SHA-256, HMAC-SHA-384, and HMAC-SHA-512," RFC 4231, December 2005 (TXT).

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

[RFC4880] Callas, J., Donnerhacke, L., Finney, H., Shaw, D., and R. Thayer, "OpenPGP Message Format," RFC 4880, November 2007 (TXT).

[RFC4960] Stewart, R., "Stream Control Transmission Protocol," RFC 4960, September 2007 (TXT).

[RFC5114] Lepinski, M. and S. Kent, "Additional Diffie-Hellman Groups for Use with IETF Standards," RFC 5114, January 2008 (TXT).

[RFC5479] Wing, D., Fries, S., Tschofenig, H., and F. Audet, "Requirements and Analysis of Media Security Management Protocols," RFC 5479, April 2009 (TXT).

[RFC5759] Solinas, J. and L. Zieglar, "Suite B Certificate and Certificate Revocation List (CRL) Profile," RFC 5759, January 2010 (TXT).

[RFC6188] McGrew, D., "The Use of AES-192 and AES-256 in Secure RTP," RFC 6188, March 2011 (TXT).

[FIPS-140-2-Annex-A] "Annex A: Approved Security Functions for FIPS PUB 140-2," NIST FIPS PUB 140-2 Annex A, January 2011.

[FIPS-140-2-Annex-D] "Annex D: Approved Key Establishment Techniques for FIPS PUB 140-2," NIST FIPS PUB 140-2 Annex D, January 2011.

[FIPS-180-3] "Secure Hash Standard (SHS)," NIST FIPS PUB 180-3, October 2008.

[FIPS-186-3] "Digital Signature Standard (DSS)," NIST FIPS PUB 186-3, June 2009.

[FIPS-197] "Advanced Encryption Standard (AES)," NIST FIPS PUB 197, November 2001.

[FIPS-198-1] "The Keyed-Hash Message Authentication Code (HMAC)," NIST FIPS PUB 198-1, July 2008.

[NIST-SP800-38A] Dworkin, M., "Recommendation for Block Cipher Modes of Operation," NIST Special Publication 800-38A, 2001 Edition.

[NIST-SP800-56A] Barker, E., Johnson, D., and M. Smid, "Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography," NIST Special Publication 800-56A Revision 1, March 2007.

[NIST-SP800-90] Barker, E. and J. Kelsey, "Recommendation for Random Number Generation Using Deterministic Random Bit Generators," NIST Special Publication 800-90 (Revised), March 2007.

[NIST-SP800-108] Chen, L., "Recommendation for Key Derivation Using Pseudorandom Functions," NIST Special Publication 800-108, October 2009.

[NSA-Suite-B] "NSA Suite B Cryptography," NSA Information Assurance Directorate, NSA Suite B Cryptography.

[NSA-Suite-B-Guide-56A] "Suite B Implementer's Guide to NIST SP 800-56A," Suite B Implementer's Guide to NIST SP 800-56A, 28 July 2009.

[TwoFish] Schneier, B., Kelsey, J., Whiting, D., Hall, C., and N. Ferguson, "Twofish: A 128-Bit Block Cipher," June 1998.

[Skein] Ferguson, N., Lucks, S., Schneier, B., Whiting, D., Bellare, M., Kohno, T., Callas, J., and J. Walker, "The Skein Hash Function Family, Version 1.3 - 1 Oct 2010."

[pgpwordlist] "PGP Word List," December 2010.

17.2. Informative References

TOC

[RFC1191] Mogul, J. and S. Deering, "Path MTU discovery," RFC 1191, November 1990 (TXT).

[RFC1981] McCann, J., Deering, S., and J. Mogul, "Path MTU Discovery for IP version 6," RFC 1981, August 1996 (TXT).

[RFC3261] Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M., and E. Schooler, "SIP: Session Initiation Protocol," RFC 3261, June 2002 (TXT).

[RFC3514] Bellovin, S., "The Security Flag in the IPv4 Header," RFC 3514, April 1 2003 (TXT).

[RFC3824] Peterson, J., Liu, H., Yu, J., and B. Campbell, "Using E.164 numbers with the Session Initiation Protocol (SIP)," RFC 3824, June 2004 (TXT).

[RFC4086] Eastlake, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security," BCP 106, RFC 4086, June 2005 (TXT).

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

[RFC4474] Peterson, J. and C. Jennings, "Enhancements for Authenticated Identity Management in the Session Initiation Protocol (SIP)," RFC 4474, August 2006 (TXT).

[RFC4475] Sparks, R., Hawrylyshen, A., Johnston, A., Rosenberg, J., and H. Schulzrinne, "Session Initiation Protocol (SIP) Torture Test Messages," RFC 4475, May 2006 (TXT).

[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 (**TXT**).

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

[RFC4579] Johnston, A. and O. Levin, "**Session Initiation Protocol (SIP) Call Control - Conferencing for User Agents**," BCP 119, RFC 4579, August 2006 (**TXT**).

[RFC5117] Westerlund, M. and S. Wenger, "**RTP Topologies**," RFC 5117, January 2008 (**TXT**).

[RFC5245] Rosenberg, J., "**Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols**," RFC 5245, April 2010 (**TXT**).

[RFC5764] McGrew, D. and E. Rescorla, "**Datagram Transport Layer Security (DTLS) Extension to Establish Keys for the Secure Real-time Transport Protocol (SRTP)**," RFC 5764, May 2010 (**TXT**).

[RFC5869] Krawczyk, H. and P. Eronen, "**HMAC-based Extract-and-Expand Key Derivation Function (HKDF)**," RFC 5869, May 2010 (**TXT**).

[RFC6090] McGrew, D., Igoe, K., and M. Salter, "**Fundamental Elliptic Curve Cryptography Algorithms**," RFC 6090, February 2011 (**TXT**).

[SRTP-AES-GCM] McGrew, D., "AES-GCM and AES-CCM Authenticated Encryption in Secure RTP (SRTP)," Work in Progress, January 2011.

[ECC-OpenPGP] Jivsov, A., "ECC in OpenPGP," Work in Progress, March 2011.

[VBR-AUDIO] Perkins, C. and J. Valin, "Guidelines for the use of Variable Bit Rate Audio with Secure RTP," Work in Progress, December 2010.

[SIP-IDENTITY] Wing, D. and H. Kaplan, "SIP Identity using Media Path," Work in Progress, February 2008.

[NIST-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.

[NIST-SP800-131A] Barker, E. and A. Roginsky, "**Recommendation for the Transitioning of Cryptographic Algorithms and Key Lengths**," NIST Special Publication 800-131A January 2011.

[SHA-3] "**Cryptographic Hash Algorithm Competition**," NIST Computer Security Resource Center Cryptographic Hash Project.

[Skein1] "**The Skein Hash Function Family - Web site**."

[XEP-0262] **Saint-Andre, P.**, "**Use of ZRTP in Jingle RTP Sessions**," XSF XEP 0262, August 2010.

[Ferguson] Ferguson, N. and B. Schneier, "Practical Cryptography," Wiley Publishing, 2003.

[Juola1] Juola, P. and P. Zimmermann, "**Whole-Word Phonetic Distances and the PGPfone Alphabet**," Proceedings of the International Conference of Spoken Language Processing (ICSLP-96), 1996.

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

[PGPfone] Zimmermann, P., "**PGPfone**," July 1996.

[Zfone] Zimmermann, P., "**Zfone Project**," 2006.

[Byzantine] "**The Two Generals' Problem**," March 2011.

[TESLA] Perrig, A., Canetti, R., Tygar, J., and D. Song, "**The TESLA Broadcast Authentication Protocol**," October 2002.

[comsec] Blossom, E., "**The VP1 Protocol for Voice Privacy Devices Version 1.2**."

[Wright1] Wright, C., Ballard, L., Coull, S., Monrose, F., and G. Masson, "**Spot me if you can: Uncovering spoken phrases in encrypted VoIP conversations**," Proceedings of the 2008 IEEE Symposium on Security and Privacy 2008.

[Sunshine] Sunshine, J., Egelman, S., Almuhimedi, H., Atri, N., and L. Cranor, "**Crying Wolf: An Empirical Study of SSL Warning Effectiveness**," USENIX Security Symposium 2009.

[dsa-1571] "**Debian Security Advisory - OpenSSL predictable random number generator**," May 2008.

Authors' Addresses

TOC

Philip Zimmermann
Zfone Project
Santa Cruz, California
EMail: prz@mit.edu
URI: http://philzimmermann.com

Alan Johnston (editor)
Avaya
St. Louis, MO 63124
EMail: alan.b.johnston@gmail.com

Jon Callas
Apple, Inc.
EMail: jon@callas.org