Network Working Group
Internet-Draft
Expires: April 25, 2005

R. Moskowitz
ICSAlabs, a Division of TruSecure Corporation
P. Nikander
P. Jokela (editor)
Ericsson Research NomadicLab
T. Henderson
The Boeing Company

October 25, 2004

Host Identity Protocol
draft-ietf-hip-base-01

Status of this Memo

Copyright Notice

Abstract

This memo specifies the details of the Host Identity Protocol (HIP). The overall description of protocol and the underlying architectural thinking is available in the separate HIP architecture specification. The Host Identity Protocol is used to establish a rapid authentication between two hosts and to provide continuity of

communications between those hosts independent of the networking layer.

The various forms of the Host Identity, Host Identity Tag (HIT) and Local Scope Identifier (LSI), are covered in detail. It is described how they are used to support authentication and the establishment of keying material, which is then used by IPsec Encapsulated Security payload (ESP) to establish a two-way secured communication channel between the hosts. The basic state machine for HIP provides a HIP compliant host with the resiliency to avoid many denial-of-service (DoS)attacks. The basic HIP exchange for two public hosts shows the actual packet flow. Other HIP exchanges, including those that work across NATs are covered elsewhere.

Table of Contents

1. Introduction

The Host Identity Protocol (HIP) provides a rapid exchange of Host Identities between two hosts. The exchange also establishes a pair IPsec Security Associations (SA), to be used with IPsec Encapsulated Security Payload (ESP) [19]. The HIP protocol is designed to be resistant to Denial-of-Service (DoS) and Man-in-the-middle (MitM) attacks, and when used to enable ESP, provides DoS and MitM protection for upper layer protocols, such as TCP and UDP.

1.1 A new name space and identifiers

The Host Identity Protocol introduces a new namespace, the Host Identity. The effects of this change are explained in the companion document, the HIP architecture [21] specification.

There are two main representations of the Host Identity, the full Host Identifier (HI) and the Host Identity Tag (HIT). The HI is a public key and directly represents the Identity. Since there are different public key algorithms that can be used with different key lengths, the HI is not good for using as a packet identifier, or as a index into the various operational tables needed to support HIP. Consequently, a hash of the HI, the Host Identity Tag (HIT), becomes the operational representation. It is 128 bits long and is used in the HIP payloads and to index the corresponding state in the end hosts.

1.2 The HIP protocol

The base HIP exchange consists of four packets. The four-packet design helps to make HIP DoS resilient. The protocol exchanges Diffie-Hellman keys in the 2nd and 3rd packets, and authenticates the parties in the 3rd and 4th packets. Additionally, it starts the cookie exchange in the 2nd packet, completing it in the 3rd packet.

The exchange uses the Diffie-Hellman exchange to hide the Host Identity of the Initiator in packet 3. The Responder's Host Identity is not protected. It should be noted, however, that both the Initiator's and the Responder's HITs are transported as such (in cleartext) in the packets, allowing an eavesdropper with a priori knowledge about the parties to verify their identities.

Data packets start after the 4th packet. The 3rd and 4th HIP packets may carry a data payload in the future. However, the details of this are to be defined later as more implementation experience is gained.

Finally, HIP is designed as an end-to-end authentication and key establishment protocol. It lacks much of the fine-grained policy

control found in Internet Key Exchange IKE RFC2409 [8] that allows IKE to support complex gateway policies. Thus, HIP is not a complete replacement for IKE.

2. Conventions used in this document

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC2119 [5].

3. Host Identifier (HI) and its representations

A public key of an asymmetric key pair is used as the Host Identifier (HI). Correspondingly, the host itself is the entity that holds the private key from the key pair. See the HIP architecture specification [21] for more details about the difference between an identity and the corresponding identifier.

HIP implementations MUST support the Rivest Shamir Adelman (RSA) [14] public key algorithm, and SHOULD support the Digital Signature Algorithm (DSA) [13] algorithm; other algorithms MAY be supported.

A hash of the HI, the Host Identity Tag (HIT), is used in protocols to represent the Host Identity. The HIT is 128 bits long and has the following three key properties: i) it is the same length as an IPv6 address and can be used in address-sized fields in APIs and protocols, ii) it is self-certifying (i.e., given a HIT, it is computationally hard to find a Host Identity key that matches the HIT), and iii) the probability of HIT collision between two hosts is very low.

In many environments, 128 bits is still considered large. For example, currently used IPv4 based applications are constrained with 32-bit address fields. Another problem is that the cohabitation of IPv6 and HIP might require some applications to differentiate an IPv6 address from a HIT. Thus, a third representation, the Local Scope Identifier (LSI), may be needed. There are two types of such LSIs: 32 bits long IPv4-compatible one and 128 bits long IPv6-compatible one. The LSI provides a compression of the HIT with only a local scope so that it can be carried efficiently in any application level packet and used in API calls. LSIs do not have the same properties as HITs (i.e., they are not self-certifying nor are they as unlikely to collide -- hence their local scope), and consequently they must be used more carefully.

Finally, HIs, HITs, and LSIs are not carried explicitly in the headers of user data packets. Instead, the IPsec Security Parameter Index (SPI) is used in data packets to index the right host context. The SPIs are selected during the HIP exchange. For user data packets, then, the combination of IPsec SPIs and IP addresses are used indirectly to identify the host context, thereby avoiding an additional explicit protocol header.

3.1 Host Identity Tag (HIT)

The Host Identity Tag is a 128 bit value -- a hash of the Host Identifier. There are two advantages of using a hash over the actual Identity in protocols. Firstly, its fixed length makes for easier

protocol coding and also better manages the packet size cost of this technology. Secondly, it presents a consistent format to the protocol whatever underlying identity technology is used.

There are two types of HITs. HITs of the first type, called *type 1 HIT*, consist of 128 bits of the SHA-1 hash of the public key. HITs of the second type consist of a Host Assigning Authority Field (HAA), and only the last 64 bits come from a SHA-1 hash of the Host Identity. This latter format for HIT is recommended for 'well known' systems. It is possible to support a resolution mechanism for these names in hierarchical directories, like the DNS. Another use of HAA is in policy controls, see Section 12.

As the type of a HIT cannot be determined by inspecting its contents, the HIT type must be communicated by some external means.

When comparing HITs for equality, it is RECOMMENDED that conforming implementations ignore the TBD top most bits. This is to allow better compatibility for legacy IPv6 applications; see Appendix A. However, independent of how many bits are actually used for HIT comparison, it is also RECOMMENDED that the final equality decision is based on the public key and not the HIT, if possible. See also Section 3.2 for related discussion.

This document fully specifies only type 1 HITs. HITs that consists of the HAA field and the hash are specified in [24].

Any conforming implementation MUST be able to deal with Type 1 HITs. When handling other than type 1 HITs, the implementation is RECOMMENDED to explicitly learn and record the binding between the Host Identifier and the HIT, as it may not be able to generate such HITs from the Host Identifiers.

3.1.1 Generating a HIT from a HI

The 128 or 64 hash bits in a HIT MUST be generated by taking the least significant 128 or 64 bits of the SHA-1 [22] hash of the Host Identifier as it is represented in the Host Identity field in a HIP payload packet.

For Identities that are either RSA or DSA public keys, the HIT is formed as follows:

1. The public key is encoded as specified in the corresponding DNSSEC document, taking the algorithm specific portion of the RDATA part of the KEY RR. There is currently only two defined public key algorithms: RSA and DSA. Hence, either of the following applies:

The RSA public key is encoded as defined in RFC3110 [14] Section 2, taking the exponent length (e_len), exponent (e) and modulus (n) fields concatenated. The length of the modulus (n) can be determined from the total HI length (hi_len) and the preceding HI fields including the exponent (e). Thus, the data to be hashed has the same length than the HI (hi_len). The fields MUST be encoded in network byte order, as defined in RFC3110 [14].
The DSA public key is encoded as defined in RFC2536 [13] Section 2, taking the fields T, Q, P, G, and Y, concatenated. Thus, the data to be hashed is 1 + 20 + 3 * 64 + 3 * 8 * T octets long, where T is the size parameter as defined in RFC2536 [13]. The size parameter T, affecting the field lengths, MUST be selected as the minimum value that is long enough to accommodate P, G, and Y. The fields MUST be encoded in network byte order, as defined in RFC2536 [13].

2. A SHA-1 hash [22] is calculated over the encoded key.
3. The least significant 128 or 64 bits of the hash result are used to create the HIT, as defined above.

The following pseudo-codes illustrates the process for both RSA and DSA. The symbol := denotes assignment; the symbol += denotes appending. The pseudo-function encode_in_network_byte_order takes two parameters, an integer (bignum) and a length in bytes, and returns the integer encoded into a byte string of the given length.

```
switch ( HI.algorithm )
{

case RSA:
 buffer := encode_in_network_byte_order ( HI.RSA.e_len,
           ( HI.RSA.e_len > 255 ) ? 3 : 1 )
 buffer += encode_in_network_byte_order ( HI.RSA.e, HI.RSA.e_len )
 buffer += encode_in_network_byte_order ( HI.RSA.n, HI.hi_len )
 break;

case DSA:
 buffer := encode_in_network_byte_order ( HI.DSA.T , 1 )
 buffer += encode_in_network_byte_order ( HI.DSA.Q , 20 )
 buffer += encode_in_network_byte_order ( HI.DSA.P , 64 + 8 * HI.DSA.T )
 buffer += encode_in_network_byte_order ( HI.DSA.G , 64 + 8 * HI.DSA.T )
 buffer += encode_in_network_byte_order ( HI.DSA.Y , 64 + 8 * HI.DSA.T )
 break;

}

digest := SHA-1 ( buffer )
```

```
hit_128 := low_order_bits ( digest, 128 )
hit_haa := concatenate ( HAA, low_order_bits ( digest,  64 ) )
```

3.2 Local Scope Identifier (LSI)

LSIs are 32 or 128 bits long localized representations of a Host Identity. The purpose of an LSI is to facilitate using Host Identities in existing IPv4 or IPv6 based protocols and APIs. The LSI can be used anywhere in system processes where IP addresses have traditionally been used, such as IPv4 and IPv6 API calls and FTP PORT commands.

The IPv4-compatible LSIs MUST be allocated from the TBD subnet and the IPv6-compatible LSIs MUST be allocated from the TBD subnet. That makes it easier to differentiate between LSIs and IP addresses at the API level. By default, the low order 24 bits of an IPv4-compatible LSI are equal to the low order 24 bits of the corresponding HIT, while the low order TBD bits of an IPv6-compatible LSI are equal to the low order TBD bits of the corresponding HIT.

A host performing a HIP handshake may discover that the LSI formed from the peer's HIT collides with another LSI in use locally (i.e., the lower 24 or TBD bits of two different HITs are the same). In that case, the host MUST handle the LSI collision locally such that application calls can be disambiguated. One possible means of doing so is to perform a Host NAT function to locally convert a peer's LSI into a different LSI value. This would require the host to ensure that LSI bits on the wire (i.e., in the application data stream) are converted back to match that host's LSI. Other alternatives for resolving LSI collisions may be added in the future.

3.3 Security Parameter Index (SPI)

SPIs are used in ESP to find the right security association for received packets. The ESP SPIs have added significance when used with HIP; they are a compressed representation of the HITs in every packet. Thus, SPIs MAY be used by intermediary systems in providing services like address mapping. Note that since the SPI has significance at the receiver, only the < DST, SPI >, where DST is a destination IP address, uniquely identifies the receiver HIT at every given point of time. The same SPI value may be used by several hosts. A single < DST, SPI > value may denote different hosts at different points of time, depending on which host is currently reachable at the DST.

Each host selects for itself the SPI it wants to see in packets received from its peer. This allows it to select different SPIs for

different peers. The SPI selection SHOULD be random; the rules of Section 2.1 of the ESP specification [19] must be followed. A different SPI SHOULD be used for each HIP exchange with a particular host; this is to avoid a replay attack. Additionally, when a host rekeys, the SPI MUST be changed. Furthermore, if a host changes over to use a different IP address, it MAY change the SPI.

One method for SPI creation that meets these criteria would be to concatenate the HIT with a 32-bit random or sequential number, hash this (using SHA1), and then use the high order 32 bits as the SPI.

The selected SPI is communicated to the peer in the third (I2) and fourth (R2) packets of the base HIP exchange. Changes in SPI are signaled with NES parameters.

4. Host Identity Protocol

The Host Identity Protocol is IP protocol TBD (number will be assigned by IANA). The HIP payload could be carried in every datagram. However, since HIP datagrams are relatively large (at least 40 bytes), and ESP already has all of the functionality to maintain and protect state, the HIP payload is 'compressed' into an ESP payload after the HIP exchange. Thus in practice, HIP packets only occur in datagrams to establish or change HIP state.

For testing purposes, the protocol number 99 is currently used.

4.1 HIP base exchange

The base HIP exchange serves to manage the establishment of state between an Initiator and a Responder. During the exchange, an IPsec Security Association is created between the hosts. The last three packets of the exchange, R1, I2, and R2, constitute a standard authenticated Diffie-Hellman key exchange for session key generation.

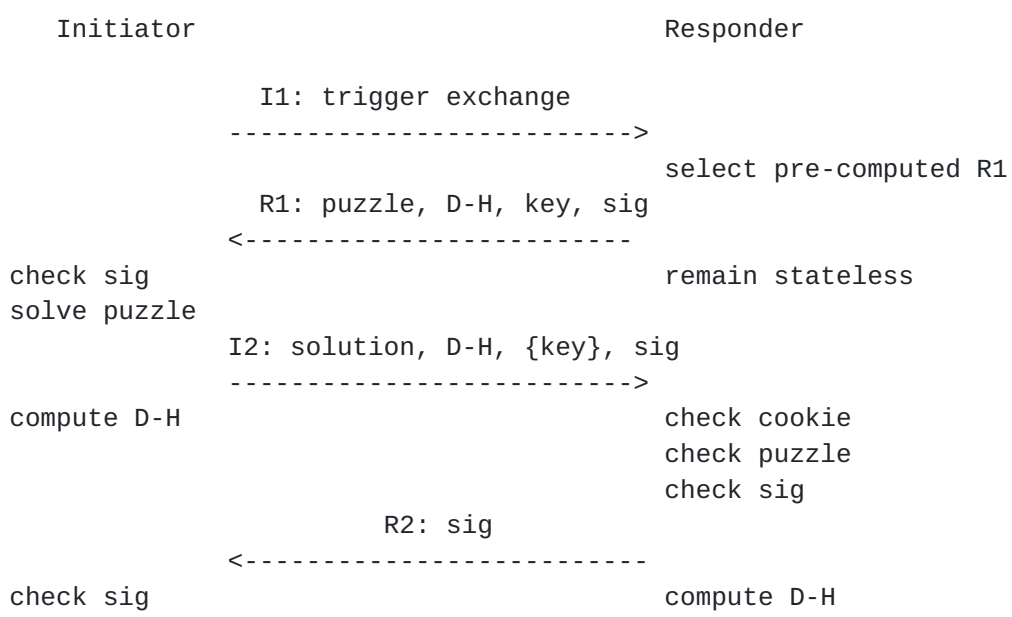
The Initiator first sends a trigger packet, I1, to the Responder. The packet contains only the HIT of the Initiator and possibly the HIT of the Responder, if it is known.

The second packet, R1, starts the actual exchange. It contains a puzzle, that is, a cryptographic challenge that the Initiator must solve before continuing the exchange. In addition, it contains the initial Diffie-Hellman parameters and a signature, covering part of the message. Some fields are left outside the signature to support pre-created R1s.

In the I2 packet, the Initiator must display the solution to the received puzzle. Without a correct solution, the I2 message is discarded. The I2 also contains a Diffie-Hellman parameter that carries needed information for the Responder. The packet is signed by the sender.

The R2 packet finalizes the 4-way handshake, containing the SPI value of the Responder. The packet is signed.

The base exchange is illustrated below. During this D-H procedure, the hosts create an IPsec session key.



In R1, the signature covers the packet, after setting the Initiator HIT, header checksum, and the PUZZLE parameter's Opaque and Random #I fields temporarily to zero, and excluding any TLVs that follow the signature.

In I2, the signature covers the whole packet, excluding any TLVs that follow the signature.

In R2, the signature and the HMAC cover the whole envelope.

In this section we cover the overall design of the base exchange. The details are the subject of the rest of this memo.

4.1.1 HIP Cookie Mechanism

The purpose of the HIP cookie mechanism is to protect the Responder from a number of denial-of-service threats. It allows the Responder to delay state creation until receiving I2. Furthermore, the puzzle included in the cookie allows the Responder to use a fairly cheap calculation to check that the Initiator is "sincere" in the sense that it has churned CPU cycles in solving the puzzle.

The Cookie mechanism has been explicitly designed to give space for various implementation options. It allows a responder implementation to completely delay session specific state creation until a valid I2 is received. In such a case a validly formatted I2 can be rejected earliest only once the Responder has checked its validity by computing one hash function. On the other hand, the design also allows a responder implementation to keep state about received I1s,

and match the received I2s against the state, thereby allowing the implementation to avoid the computational cost of the hash function. The drawback of this latter approach is the requirement of creating state. Finally, it also allows an implementation to use any combination of the space-saving and computation-saving mechanisms.

One possible way for a Responder to remain stateless but drop most spoofed I2s is to base the selection of the cookie on some function over the Initiator's Host Identity. The idea is that the Responder has a (perhaps varying) number of pre-calculated R1 packets, and it selects one of these based on the information carried in I1. When the Responder then later receives I2, it checks that the cookie in the I2 matches with the cookie sent in the R1, thereby making it impractical for the attacker to first exchange one I1/R1, and then generate a large number of spoofed I2s that seemingly come from different IP addresses or use different HITs. The method does not protect from an attacker that uses fixed IP addresses and HITs, though. Against such an attacker it is probably best to create a piece of local state, and remember that the puzzle check has previously failed. See Appendix D for one possible implementation. Note, however, that the implementations MUST NOT use the exact implementation given in the appendix, and SHOULD include sufficient randomness to the algorithm so that algorithm complexity attacks become impossible [26].

The Responder can set the difficulty for Initiator, based on its concern of trust of the Initiator. The Responder SHOULD use heuristics to determine when it is under a denial-of-service attack, and set the difficulty value K appropriately.

The Responder starts the cookie exchange when it receives an I1. The Responder supplies a random number I, and requires the Initiator to find a number J. To select a proper J, the Initiator must create the concatenation of I, the HITs of the parties, and J, and take a SHA-1 hash over this concatenation. The lowest order K bits of the result MUST be zeros.

To generate a proper number J, the Initiator will have to generate a number of Js until one produces the hash target of zero. The Initiator SHOULD give up after exceeding the puzzle lifetime received in PUZZLE TLV. The Responder needs to re-create the concatenation of I, the HITs, and the provided J, and compute the hash once to prove that the Initiator did its assigned task.

To prevent pre-computation attacks, the Responder MUST select the number I in such a way that the Initiator cannot guess it. Furthermore, the construction MUST allow the Responder to verify that the value was indeed selected by it and not by the Initiator. See

Appendix D for an example on how to implement this.

Using the Opaque data field in the ECHO_REQUEST, the Responder can include some data in R1 that the Initiator must copy unmodified in the corresponding I2 packet. The Responder can generate the Opaque data e.g. using the sent I, some secret and possibly other related data. Using this same secret, received I in I2 packet and possible other data, the Receiver can verify that it has itself sent the I to the Initiator. The Responder must change the secret periodically.

It is RECOMMENDED that the Responder generates a new cookie and a new R1 once every few minutes. Furthermore, it is RECOMMENDED that the Responder remembers an old cookie at least 2*lifetime seconds after it has been deprecated. These time values allow a slower Initiator to solve the cookie puzzle while limiting the usability that an old, solved cookie has to an attacker.

NOTE: The protocol developers explicitly considered whether R1 should include a timestamp in order to protect the Initiator from replay attacks. The decision was NOT to include a timestamp.

In R1, the values I and K are sent in network byte order. Similarly, in I2 the values I and J are sent in network byte order. The SHA-1 hash is created by concatenating, in network byte order, the following data, in the following order:

- 64-bit random value I, in network byte order, as appearing in R1 and I2.
- 128-bit initiator HIT, in network byte order, as appearing in the HIP Payload in R1 and I2.
- 128-bit responder HIT, in network byte order, as appearing in the HIP Payload in R1 and I2.
- 64-bit random value J, in network byte order, as appearing in I2.

In order to be a valid response cookie, the K low-order bits of the resulting SHA-1 digest must be zero.

Notes:

- The length of the data to be hashed is 48 bytes.
- All the data in the hash input MUST be in network byte order.
- The order of the initiator and responder HITs are different in the R1 and I2 packets, see Section 6.1. Care must be taken to copy the values in right order to the hash input.

Precomputation by the Responder

- Sets up the challenge difficulty K.
- Creates a signed R1 and caches it.

Responder

Selects a suitable cached R1.
Generates a random number I.
Sends I and K in a HIP Cookie in the R1.
Saves I and K for a Delta time.

Initiator

Generates repeated attempts to solve the challenge until a matching J is found:

Ltrunc(SHA-1(I | HIT-I | HIT-R | J), K) == 0
Sends I and J in HIP Cookie in a I2.

Responder

Verifies that the received I is a saved one.
Finds the right K based on I.
Computes V := Ltrunc(SHA-1(I | HIT-I | HIT-R | J), K)
Rejects if V != 0
Accept if V == 0

The Ltrunc (SHA-1(), K) denotes the lowest order K bits of the SHA-1 result.

4.1.2 Authenticated Diffie-Hellman protocol

The packets R1, I2, and R2 implement a standard authenticated Diffie-Hellman exchange. The Responder sends its public Diffie-Hellman key and its public authentication key, i.e., its host identity, in R1. The signature in R1 allows the Initiator to verify that the R1 has been once generated by the Responder. However, since it is precomputed and therefore does not cover all of the packet, it does not protect from replay attacks.

When the Initiator receives an R1, it computes the Diffie-Hellman session key. It creates a HIP security association using keying material from the session key (see Section 9), and uses the security association to encrypt its public authentication key, i.e., host identity. The resulting I2 contains the Initiator's Diffie-Hellman key and its the encrypted public authentication key. The signature in I2 covers all of the packet.

The Responder extracts the Initiator Diffie-Hellman public key from the I2, computes the Diffie-Hellman session key, creates a corresponding HIP security association, and decrypts the Initiator's public authentication key. It can then verify the signature using the authentication key.

The final message, R2, is needed to protect the Initiator from replay attacks.

4.1.3 HIP replay protection

The HIP protocol includes the following mechanisms to protect against malicious replays. Responders are protected against replays of I1 packets by virtue of the stateless response to I1s with presigned R1 messages. Initiators are protected against R1 replays by a monotonically increasing "R1 generation counter" included in the R1. Responders are protected against replays or false I2s by the cookie mechanism (Section 4.1.1 above), and optional use of opaque data. Hosts are protected against replays to R2s and UPDATEs by use of a less expensive HMAC verification preceding HIP signature verification.

The R1 generation counter is a monotonically increasing 64-bit counter that may be initialized to any value. The scope of the counter MAY be system-wide but SHOULD be per host identity, if there is more than one local host identity. The value of this counter SHOULD be kept across system reboots and invocations of the HIP signaling process. This counter indicates the current generation of cookie puzzles. Implementations MUST accept puzzles from the current generation and MAY accept puzzles from earlier generations. A system's local counter MUST be incremented at least as often as every time old R1s cease to be valid, and SHOULD never be decremented, lest the host expose its peers to the replay of previously generated, higher numbered R1s. Also, the R1 generation counter MUST NOT roll over; if the counter is about to become exhausted, the corresponding HI must be abandoned and replaced with a new one.

A host may receive more than one R1, either due to sending multiple I1s (Section 8.4.1) or due to a replay of an old R1. When sending multiple I1s, an initiator SHOULD wait for a small amount of time after the first R1 reception to allow possibly multiple R1s to arrive, and it SHOULD respond to an R1 among the set with the largest R1 generation counter. If an initiator is processing an R1 or has already sent an I2 (still waiting for R2) and it receives another R1 with a larger R1 generation counter, it MAY elect to restart R1 processing with the fresher R1, as if it were the first R1 to arrive.

Upon conclusion of an active HIP association with another host, the R1 generation counter associated with the peer host SHOULD be flushed. A local policy MAY override the default flushing of R1 counters on a per-HIT basis. The reason for recommending the flushing of this counter is that there may be hosts where the R1 generation counter (occasionally) decreases; e.g., due to hardware failure.

4.2 TCP and UDP pseudo-header computation

When computing TCP and UDP checksums on sockets bound to HITs or LSIs, the IPv6 pseudo-header format [11] MUST be used. Additionally, the HITs MUST be used in the place of the IPv6 addresses in the IPv6 pseudo-header. Note that the pseudo-header for actual HIP payloads is computed differently; see Section 6.1.2.

4.3 Updating a HIP association

A HIP association between two hosts may need to be updated over time. Examples include the need to rekey expiring security associations, add new security associations, or change IP addresses associated with hosts. This document only specifies how UPDATE is used for rekeying; other uses are deferred to other drafts.

HIP provides a general purpose UPDATE packet, which can carry multiple HIP parameters, for updating the HIP state between two peers. The UPDATE mechanism has the following properties:

- UPDATE messages carry a monotonically increasing sequence number and are explicitly acknowledged by the peer. Lost UPDATEs or acknowledgments may be recovered via retransmission. Multiple UPDATE messages may be outstanding.
- UPDATE is protected by both HMAC and HIP_SIGNATURE parameters, since processing UPDATE signatures alone is a potential DoS attack against intermediate systems.

The UPDATE packet is defined in Section 7.6.

4.4 Error processing

HIP error processing behaviour depends on whether there exists an active HIP association or not. In general, if an HIP security association exists between the sender and receiver of a packet causing an error condition, the receiver SHOULD respond with a NOTIFY packet. On the other hand, if there are no existing HIP security associations between the sender and receiver, or the receiver cannot reasonably determine the identity of the sender, the receiver MAY respond with a suitable ICMP message; see Section 6.3 for more details.

4.5 Certificate distribution

HIP does not define how to use certificates. However, it does define a simple certificate transport mechanisms that MAY be used to implement certificate based security policies. The certificate payload is defined in Section 6.2.11, and the certificate packet in Section 7.5.

4.6 Sending data on HIP packets

A future version of this document may define how to include ESP protected data on various HIP packets. However, currently the HIP header is a terminal header, and not followed by any other headers.

5. HIP protocol overview

The following material is an overview of the HIP protocol operation. Section 8 describes the packet processing steps in more detail.

A typical HIP packet flow is shown below, between an Initiator (I) and a Responder (R). It illustrates the exchange of four HIP packets (I1, R1, I2, and R2).

```
I --> Directory: lookup R
I <-- Directory: return R's addresses, and HI and/or HIT
I1      I --> R (Hi. Here is my I1, let's talk HIP)
R1      I <-- R (OK. Here is my R1, handle this HIP cookie)
I2      I --> R (Compute, compute, here is my counter I2)
R2      I <-- R (OK.  Let's finish HIP with my R2)
I --> R (ESP protected data)
I <-- R (ESP protected data)
```

By definition, the system initiating a HIP exchange is the Initiator, and the peer is the Responder. This distinction is forgotten once the base exchange completes, and either party can become the initiator in future communications.

5.1 HIP Scenarios

The HIP protocol and state machine is designed to recover from one of the parties crashing and losing its state. The following scenarios describe the main use cases covered by the design.

- No prior state between the two systems.
  - The system with data to send is the Initiator. The process follows the standard four packet base exchange, establishing the SAs.
- The system with data to send has no state with the receiver, but the receiver has a residual SA.
  - The system with data to send is the Initiator. The Initiator acts as in no prior state, sending I1 and getting R1. When the Responder receives a valid I2, the old SAs are 'discovered' and deleted, and the new SAs are established.
- The system with data to send has an SA, but the receiver does not.
  - The system sends data on the outbound SA. The receiver 'detects' the situation when it receives an ESP packet that contains an unknown SPI. The receiving host MUST discard this packet, in accordance with IPsec architecture. Optionally, the receiving host MAY send an ICMP packet with the Parameter Problem type to inform about invalid SPI (see Section 6.3, and it MAY initiate a new HIP negotiation. However, responding with these optional mechanisms is implementation or policy dependent.

A system determines that it needs to reset ESP Sequence Number, or rekey.

The system sends a HIP UPDATE packet. The peer responds with a HIP UPDATE response. The UPDATE exchanges can refresh or establish new SAs for peers.

5.2 Refusing a HIP exchange

A HIP aware host may choose not to accept a HIP exchange. If the host's policy is to only be an Initiator, it should begin its own HIP exchange. A host MAY choose to have such a policy since only the Initiator HI is protected in the exchange. There is a risk of a race condition if each host's policy is to only be an Initiator, at which point the HIP exchange will fail.

If the host's policy does not permit it to enter into a HIP exchange with the Initiator, it should send an ICMP 'Destination Unreachable, Administratively Prohibited' message. A more complex HIP packet is not used here as it actually opens up more potential DoS attacks than a simple ICMP message.

5.3 Reboot and SA timeout restart of HIP

Simulating a loss of state is a potential DoS attack. The following process has been crafted to manage state recovery without presenting a DoS opportunity.

If a host reboots or times out, it has lost its HIP state. If the system that lost state has a datagram to deliver to its peer, it simply restarts the HIP exchange. The peer replies with an R1 HIP packet, but does not reset its state until it receives the I2 HIP packet. The I2 packet MUST have a valid solution to the puzzle and, if inserted in R1, a valid Opaque data as well as a valid signature. Note that either the original Initiator or the Responder could end up restarting the exchange, becoming the new Initiator.

If a system receives an ESP packet for an unknown SPI, it is possible that it has lost the state and its peer has not. It MAY send an ICMP packet with the Parameter Problem type, the Pointer pointing to the SPI value within the ESP header. Reacting to ESP traffic with unknown SPI depends on the implementation and the environment where the implementation is used.

The initiating host cannot know, if the SA indicated by the received ESP packet is either a HIP SA or and IKE SA. If the old SA was not a HIP SA, the peer may not respond to the I1 packet.

After sending the I1, the HIP negotiation proceeds as normally and,

when successful, the SA is created at the initiating end. The peer end removes the OLD SA and replaces it with the new one.

5.4 HIP State Machine

The HIP protocol itself has very little state. In the HIP base exchange, there is an Initiator and a Responder. Once the SAs are established, this distinction is lost. If the HIP state needs to be re-established, the controlling parameters are which peer still has state and which has a datagram to send to its peer. The following state machine attempts to capture these processes.

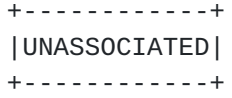
The state machine is presented in a single system view, representing either an Initiator or a Responder. There is not a complete overlap of processing logic here and in the packet definitions. Both are needed to completely implement HIP.

Implementors must understand that the state machine, as described here, is informational. Specific implementations are free to implement the actual functions differently. Section 8 describes the packet processing rules in more detail. This state machine focuses on the HIP I1, R1, I2, R2, and UPDATE packets only, and specifically, the state induced by an UPDATE that triggers a rekeying event. Other states may be introduced by mechanisms in other drafts (such as mobility and multihoming).

5.4.1 HIP States

UNASSOCIATED State machine start
I1-SENT Initiating HIP
I2-SENT Waiting to finish HIP
R2-SENT Waiting to finish HIP
ESTABLISHED HIP SA established
REKEYING HIP SA established, but UPDATE is outstanding for rekeying
CLOSING HIP SA closing, no data (ESP) can be sent
CLOSED HIP SA closed, no data (ESP) can be sent
E-FAILED HIP exchange failed

5.4.2 HIP State Processes



Datagram to send requiring a new SA, send I1 and go to I1-SENT
Receive I1, send R1 and stay at UNASSOCIATED
Receive I2, process
if successful, send R2 and go to R2-SENT

```
        if fail, stay at UNASSOCIATED

   Receive ESP for unknown SA, optionally send ICMP as defined
        in
   Section 6.3
    and stay at UNASSOCIATED


   Receive CLOSE, or UPDATE, optionally send ICMP Parameter
   Problem and stay in UNASSOCIATED.

   Receive ANYOTHER, drop and stay at UNASSOCIATED

   +---------+
   | I1-SENT |  Initiating HIP
   +---------+

   Receive I1, send R1 and stay at I1-SENT
   Receive I2, process
        if successful, send R2 and go to R2-SENT
        if fail, stay at I1-SENT
   Receive R1, process
        if successful, send I2 and go to I2-SENT
        if fail, go to E-FAILED

   Receive ANYOTHER, drop and stay at I1-SENT
   Timeout, increment timeout counter
        If counter is less than I1_RETRIES_MAX, send I1 and stay at I1-SENT
        If counter is greater than I1_RETRIES_MAX, go to E-FAILED

   +---------+
   | I2-SENT | Waiting to finish HIP
   +---------+

   Receive I1, send R1 and stay at I2-SENT
   Receive R1, process
        if successful, send I2 and cycle at I2-SENT
        if fail, stay at I2-SENT
   Receive I2, process
        if successful, send R2 and go to R2-SENT
        if fail, stay at I2-SENT
   Receive R2, process
        if successful, go to ESTABLISHED
        if fail, go to E-FAILED

   Receive ANYOTHER, drop and stay at I2-SENT
   Timeout, increment timeout counter
        If counter is less than I2_RETRIES_MAX, send I2 and stay at I2-SENT
```

If counter is greater than I2_RETRIES_MAX, go to E-FAILED

```
+---------+
| R2-SENT | Waiting to finish HIP
+---------+

Receive I1, send R1 and stay at R2-SENT
Receive I2, process,
   if successful, send R2, and cycle at R2-SENT
   if failed, stay at R2-SENT
Receive R1, drop and stay at R2-SENT
Receive R2, drop and stay at R2-SENT

Receive ESP for SA, process and go to ESTABLISHED
Receive UPDATE, go to ESTABLISHED and process from ESTABLISHED state

Move to ESTABLISHED after an implementation specific time.

+------------+
|ESTABLISHED | HIP SA established
+------------+

Receive I1, send R1 and stay at ESTABLISHED
Receive I2, process with cookie and possible Opaque data verification
     if successful, send R2, drop old SAs, establish new SA, go to
     R2-SENT
     if fail, stay at ESTABLISHED
Receive R1, drop and stay at ESTABLISHED
Receive R2, drop and stay at ESTABLISHED

Receive ESP for SA, process and stay at ESTABLISHED
Receive UPDATE, process
     if successful, send UPDATE in reply and go to REKEYING
     if failed, stay at ESTABLISHED
Need rekey,
     send UPDATE and go to REKEYING
No packet sent/received during UAL minutes, send CLOSE and go to
  CLOSING.
Receive CLOSE, process
     if successful, send CLOSE_ACK and go to CLOSED
     if failed, stay at ESTABLISHED


+---------+
| CLOSING | HIP association has not been used for UAL (Unused
+---------+ Association Lifetime) minutes.

Datagram to send, requires the creation of another incarnation
```

```
       of the HIP association, started by sending an I1,
       and stay at CLOSING

   Receive I1, send R1 and stay at CLOSING
   Receive I2, process
       if successful, send R2 and go to R2-SENT
       if fail, stay at CLOSING

   Receive R1, process
       if successful, send I2 and go to I2-SENT
       if fail, stay at CLOSING

   Receive CLOSE, process
       if successful, send CLOSE_ACK, discard state and go to CLOSED
       if failed, stay at CLOSING
   Receive CLOSE_ACK, process
       if successful, discard state and go to UNASSOCIATED
       if failed, stay at CLOSING

   Receive ANYOTHER, drop and stay at CLOSING

   Timeout, increment timeout sum, reset timer
       if timeout sum is less than UAL+MSL minutes, retransmit CLOSE
         and stay at CLOSING
       if timeout sum is greater than UAL+MSL minutes, go to
          UNASSOCIATED

   +--------+
   | CLOSED | CLOSE_ACK sent, resending CLOSE_ACK if necessary
   +--------+

   Datagram to send, requires the creation of another incarnation
       of the HIP association, started by sending an I1,
       and stay at CLOSED

   Receive I1, send R1 and stay at CLOSED
   Receive I2, process
       if successful, send R2 and go to R2-SENT
       if fail, stay at CLOSED

   Receive R1, process
       if successful, send I2 and go to I2-SENT
       if fail, stay at CLOSED

   Receive CLOSE, process
       if successful, send CLOSE_ACK, stay at CLOSED
       if failed, stay at CLOSED
```

Receive CLOSE_ACK, process
if successful, discard state and go to UNASSOCIATED
if failed, stay at CLOSED

Receive ANYOTHER, drop and stay at CLOSED

Timeout (UAL + 2MSL), discard state and go to UNASSOCIATED

+----------+
| REKEYING | HIP SA established, rekey pending
+----------+

Receive I1, send R1 and stay at REKEYING
Receive I2, process with cookie and possible Opaque data verification
if successful, send R2, drop old SA and go to R2-SENT
if fail, stay at REKEYING
Receive R1, drop and stay at REKEYING
Receive R2, drop and stay at REKEYING

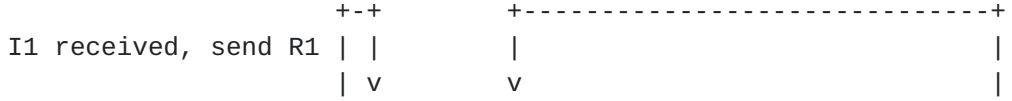
Receive ESP for SA, process and stay at REKEYING
Receive UPDATE, process
if successful completion of rekey, go to ESTABLISHED
if failed, stay at REKEYING
Timeout, increment timeout counter
If counter is less than UPDATE_RETRIES_MAX, send UPDATE and stay at REKEYING
If counter is greater than UPDATE_RETRIES_MAX, go to E-FAILED

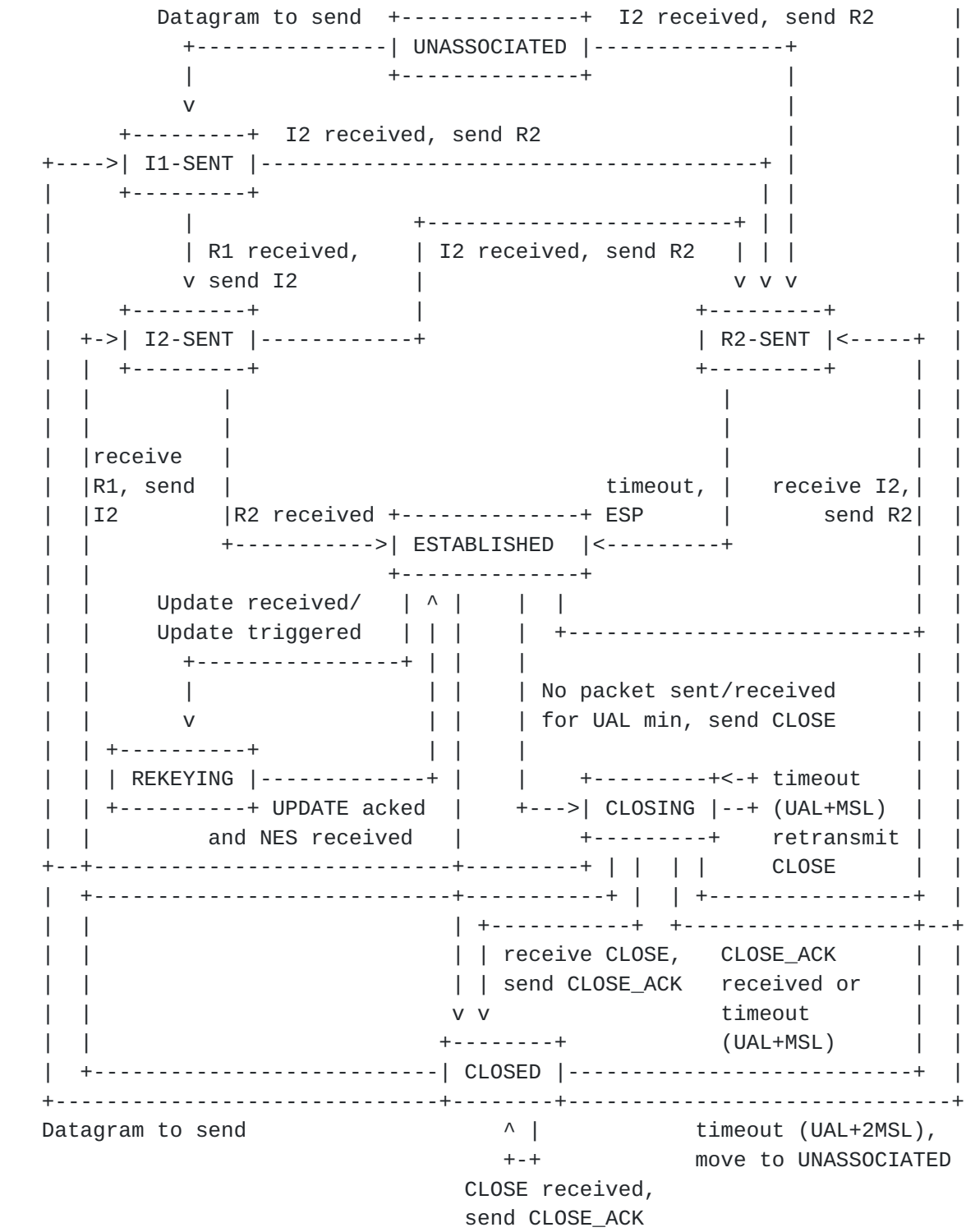
+----------+
| E-FAILED | HIP failed to establish association with peer
+----------+

Move to UNASSOCIATED after an implementation specific time. Re-negotiation is possible after moving to UNASSOCIATED state.

5.4.3 Simplified HIP State Diagram

The following diagram shows the major state transitions. Transitions based on received packets implicitly assume that the packets are successfully authenticated or processed. The diagram assumes that UPDATE messages are being used for rekeying.

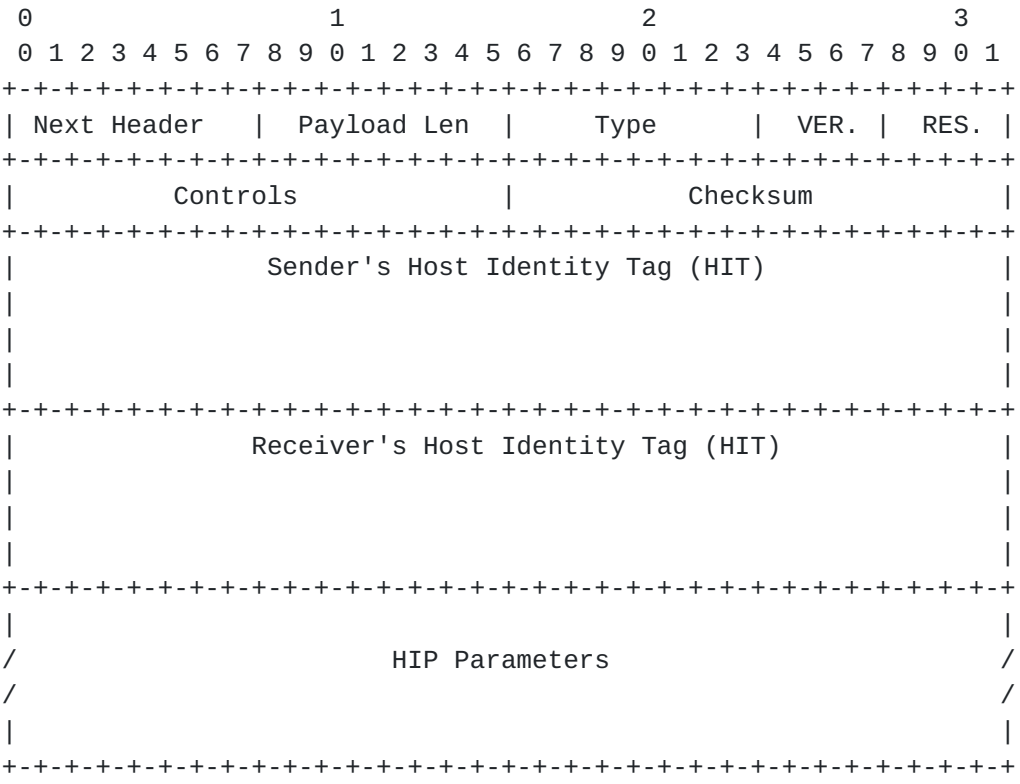



Datagram to send
UNASSOCIATED
I2 received, send R2
I1-SENT
I2 received, send R2
R1 received,
send I2
I2 received, send R2
I2-SENT
R2-SENT
receive
R1, send
I2
R2 received
ESTABLISHED
timeout, ESP
receive I2,
send R2
Update received/
Update triggered
No packet sent/received
for UAL min, send CLOSE
REKEYING
UPDATE acked
and NES received
CLOSING
timeout
(UAL+MSL)
retransmit
CLOSE
receive CLOSE,
send CLOSE_ACK
CLOSE_ACK
received or
timeout
(UAL+MSL)
CLOSED
Datagram to send
CLOSE received,
send CLOSE_ACK
timeout (UAL+2MSL),
move to UNASSOCIATED

6. Packet formats

6.1 Payload format

All HIP packets start with a fixed header.



The HIP header is logically an IPv6 extension header. However, this document does not describe processing for Next Header values other than decimal 59, IPPROTO_NONE, the IPV6 no next header value. Future documents MAY do so. However, implementations MUST ignore trailing data if a Next Header value is received that is not implemented.

The Header Length field contains the length of the HIP Header and the length of HIP parameters in 8 bytes units, excluding the first 8 bytes. Since all HIP headers MUST contain the sender's and receiver's HIT fields, the minimum value for this field is 4, and conversely, the maximum length of the HIP Parameters field is (255*8)-32 = 2008 bytes. Note: this sets an additional limit for sizes of TLVs included in the Parameters field, independent of the individual TLV parameter maximum lengths.

The Packet Type indicates the HIP packet type. The individual packet types are defined in the relevant sections. If a HIP host receives a

HIP packet that contains an unknown packet type, it MUST drop the packet.

The HIP Version is four bits. The current version is 1. The version number is expected to be incremented only if there are incompatible changes to the protocol. Most extensions can be handled by defining new packet types, new parameter types, or new controls.

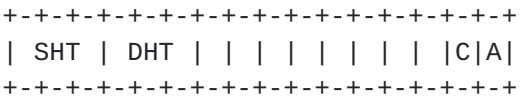
The following four bits are reserved for future use. They MUST be zero when sent, and they SHOULD be ignored when handling a received packet.

The HIT fields are always 128 bits (16 bytes) long.

6.1.1 HIP Controls

The HIP control section transfers information about the structure of the packet and capabilities of the host.

The following fields have been defined:



C - Certificate One or more certificate packets (CER) follows this HIP packet (see Section 7.5).

A - Anonymous If this is set, the sender's HI in this packet is anonymous, i.e., one not listed in a directory. Anonymous HIs SHOULD NOT be stored. This control is set in packets R1 and/or I2. The peer receiving an anonymous HI may choose to refuse it by silently dropping the exchange.

SHT - Sender's HIT Type Currently the following values are specified:

0 RESERVED
1 Type 1 HIT
2 Type 2 HIT
3-6 UNASSIGNED
7 RESERVED

DHT - Destination's HIT Type Using the same values as SHT.

The rest of the fields are reserved for future use and MUST be set to zero on sent packets and ignored on received packets.

6.1.2 Checksum

The checksum field is located at the same location within the header as the checksum field in UDP packets, enabling hardware assisted checksum generation and verification. Note that since the checksum covers the source and destination addresses in the IP header, it must

be recomputed on HIP based NAT boxes.

If IPv6 is used to carry the HIP packet, the pseudo-header [11] contains the source and destination IPv6 addresses, HIP packet length in the pseudo-header length field, a zero field, and the HIP protocol number (TBD, see Section 4) in the Next Header field. The length field is in bytes and can be calculated from the HIP header length field: (HIP Header Length + 1) * 8.

In case of using IPv4, the IPv4 UDP pseudo header format [1] is used. In the pseudo header, the source and destination addresses are those used in the IP header, the zero field is obviously zero, the protocol is the HIP protocol number (TBD, see Section 4), and the length is calculated as in the IPv6 case.

6.2 HIP parameters

The HIP Parameters are used to carry the public key associated with the sender's HIT, together with other related security information. The HIP Parameters consists of ordered parameters, encoded in TLV format.

The following parameter types are currently defined.

TLV	Type	Length	Data
SPI	1	4	Remote's SPI.
R1_COUNTER	2	12	System Boot Counter
PUZZLE	5	12	K and Random #I
SOLUTION	7	20	K, Random #I and puzzle solution
NES	9	12	Old SPI, New SPI and other info needed for UPDATE
SEQ	11	4	Update packet ID number
ACK	13	variable	Update packet ID number
DIFFIE_HELLMAN	15	variable	public key
HIP_TRANSFORM	17	variable	HIP Encryption and Integrity Transform
ESP_TRANSFORM	19	variable	ESP Encryption and Authentication Transform

ENCRYPTED	21	variable	Encrypted part of I2 or CER packets
HOST_ID	35	variable	Host Identity with Fully Qualified Domain Name
CERT	64	variable	HI certificate
NOTIFY	256	variable	Informational data
ECHO_REQUEST	1022	variable	Opaque data to be echoed back; under signature
ECHO_RESPONSE	1024	variable	Opaque data echoed back; under signature
HMAC	65245	20	HMAC based message authentication code, with key material from HIP_TRANSFORM
HMAC_2	65247	20	HMAC based message authentication code, with key material from HIP_TRANSFORM
HIP_SIGNATURE_2	65277	variable	Signature of the R1 packet
HIP_SIGNATURE	65279	variable	Signature of the packet
ECHO_REQUEST	65281	variable	Opaque data to be echoed back
ECHO_RESPONSE	65283	variable	Opaque data echoed back; after signature

6.2.1 TLV format

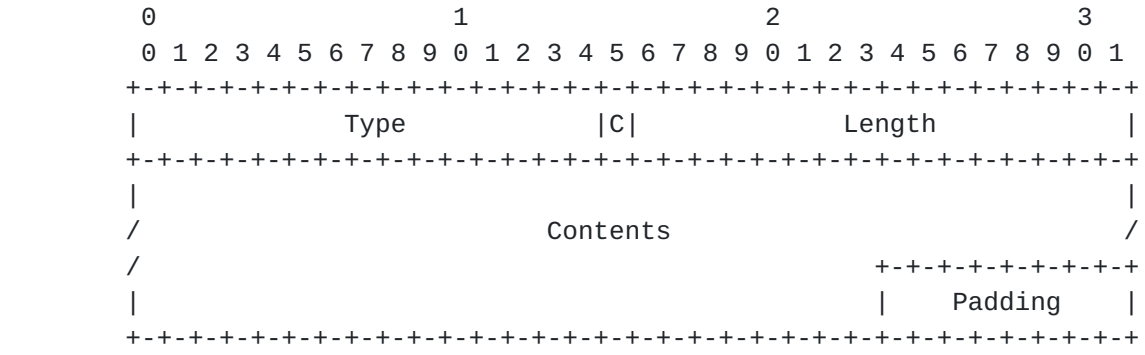
The TLV encoded parameters are described in the following subsections. The type-field value also describes the order of these fields in the packet. The parameters MUST be included into the packet so that the types form an increasing order. If the order does not follow this rule, the packet is considered to be malformed and it MUST be discarded.

All the TLV parameters have a length (including Type and Length fields) which is a multiple of 8 bytes. When needed, padding MUST be added to the end of the parameter so that the total length becomes a multiple of 8 bytes. This rule ensures proper alignment of data. If

padding is added, the Length field MUST NOT include the padding. Any added padding bytes MUST be set zero by the sender, but their content SHOULD NOT be checked on the receiving end.

Consequently, the Length field indicates the length of the Contents field (in bytes). The total length of the TLV parameter (including Type, Length, Contents, and Padding) is related to the Length field according to the following formula:

Total Length = 11 + Length - (Length + 3) % 8;



Type	Type code for the parameter
C	Critical. One if this parameter is critical, and MUST be recognized by the recipient, zero otherwise. The C bit is considered to be a part of the Type field. Consequently, critical parameters are always odd and non-critical ones have an even value.
Length	Length of the Contents, in bytes.
Contents	Parameter specific, defined by Type
Padding	Padding, 0-7 bytes, added if needed

Critical parameters MUST be recognized by the recipient. If a recipient encounters a critical parameter that it does not recognize, it MUST NOT process the packet any further.

Non-critical parameters MAY be safely ignored. If a recipient encounters a non-critical parameter that it does not recognize, it SHOULD proceed as if the parameter was not present in the received packet.

6.2.2 Defining new parameters

Future specifications may define new parameters as needed. When defining new parameters, care must be taken to ensure that the parameter type values are appropriate and leave suitable space for other future extensions. One must remember that the parameters MUST always be arranged in the increasing order by the type code, thereby

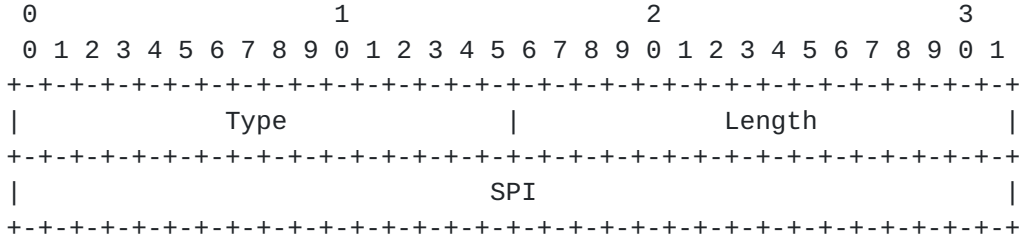
limiting the order of parameters.

The following rules must be followed when defining new parameters.

1. The low order bit C of the Type code is used to distinguish between critical and non-critical parameters.
2. A new parameter may be critical only if an old recipient ignoring it would cause security problems. In general, new parameters SHOULD be defined as non-critical, and expect a reply from the recipient.
3. If a system implements a new critical parameter, it MUST provide the ability to configure the associated feature off, such that the critical parameter is not sent at all. The configuration option must be well documented. By default, sending of such a new critical parameter SHOULD be off. In other words, the management interface MUST allow vanilla standards only mode as a default configuration setting, and MAY allow new critical payloads to be configured on (and off).
4. The following type codes are reserved for future base protocol extensions, and may be assigned only through an appropriate WG or RFC action.
   0 - 511
   65024 - 65535
5. The following type codes are reserved for experimentation and private use. Types SHOULD be selected in a random fashion from this range, thereby reducing the probability of collisions. A method employing genuine randomness (such as flipping a coin) SHOULD be used.
   32768 - 49141
6. All other parameter type codes MUST be registered by the IANA. See Section 14.

6.2.3 SPI

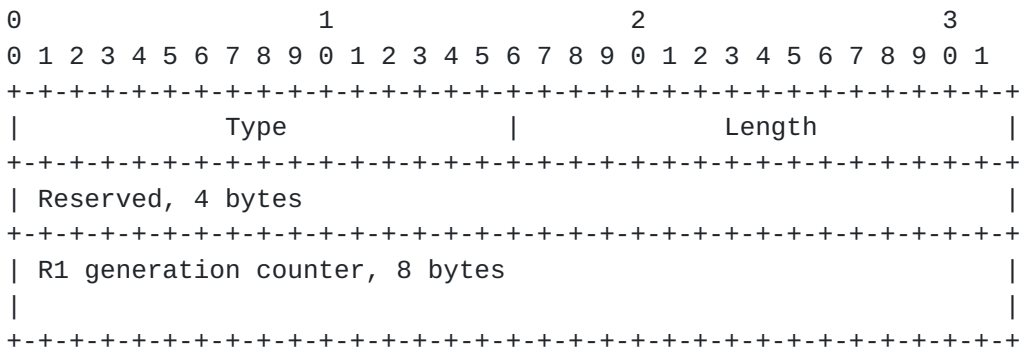
The SPI parameter contains the SPI that the receiving host must use when sending data to the sending host. It may be possible, in future extensions of this protocol, for multiple SPIs to exist in a host-host communications context.



Type 1
Length 4

SPI Security Parameter Index

6.2.4 R1_COUNTER

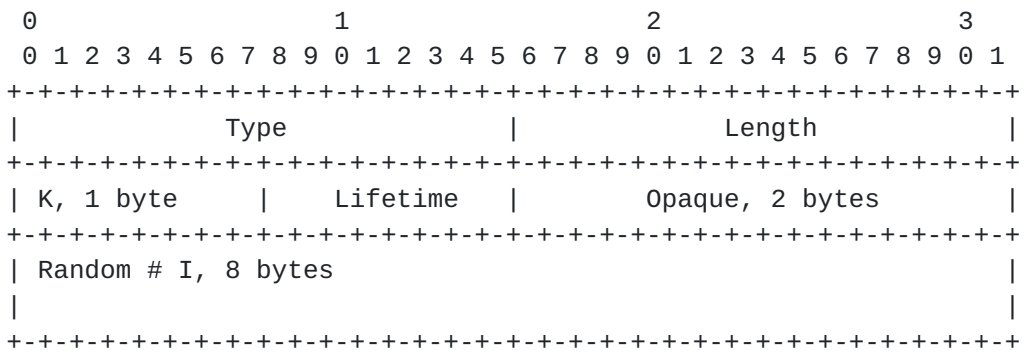


Type 2
Length 12
R1 generation counter The current generation of valid puzzles

The R1_COUNTER parameter contains an 64-bit unsigned integer in network byte order, indicating the current generation of valid puzzles. The sender is supposed to increment this counter periodically. It is RECOMMENDED that the counter value is incremented at least as often as old PUZZLE values are deprecated so that SOLUTIONs to them are no longer accepted.

The R1_COUNTER parameter is optional. It SHOULD be included in the R1 (in which case it is covered by the signature), and if present in the R1, it MAY be echoed (including the Reserved field) by the Initiator in the I2.

6.2.5 PUZZLE



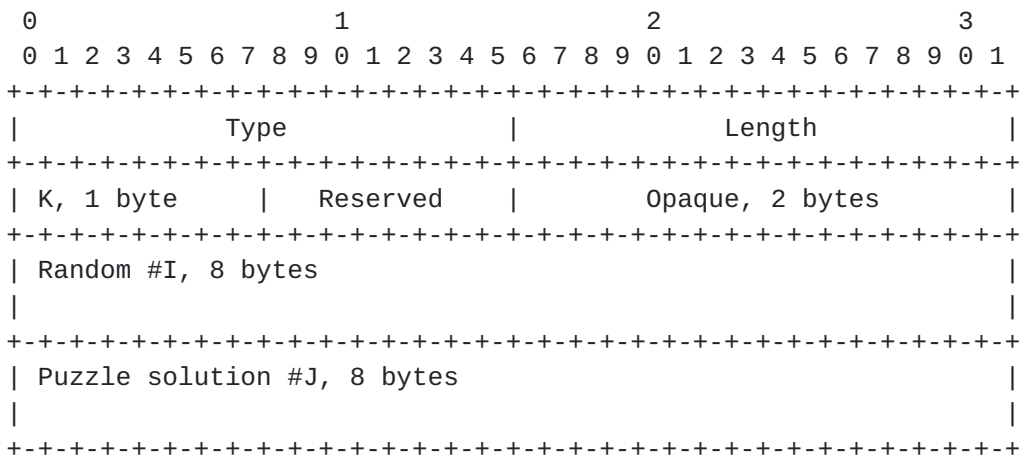
Type	5
Length	12
K	K is the number of verified bits
Lifetime	Puzzle lifetime 2^(value-32) seconds
Opaque	Data set by the Responder, indexing the puzzle
Random #I	random number

Random #I is represented as 64-bit integer, K and Lifetime as 8-bit integer, all in network byte order.

The PUZZLE parameter contains the puzzle difficulty K and an 64-bit puzzle random integer #I. Puzzle Lifetime indicates the time during which the puzzle solution is valid and sets a time limit for initiator which it should not exceed while trying to solve the puzzle. The lifetime is indicated as power of 2 using formula 2^(Lifetime-32) seconds. A puzzle MAY be augmented by including an ECHO_REQUEST parameter to an R1. The contents of the ECHO_REQUEST are then echoed back in ECHO_RESPONSE, allowing the Responder to use the included information as a part of puzzle processing.

The Opaque and Random #I field are not covered by the HIP_SIGNATURE_2 parameter.

6.2.6 SOLUTION

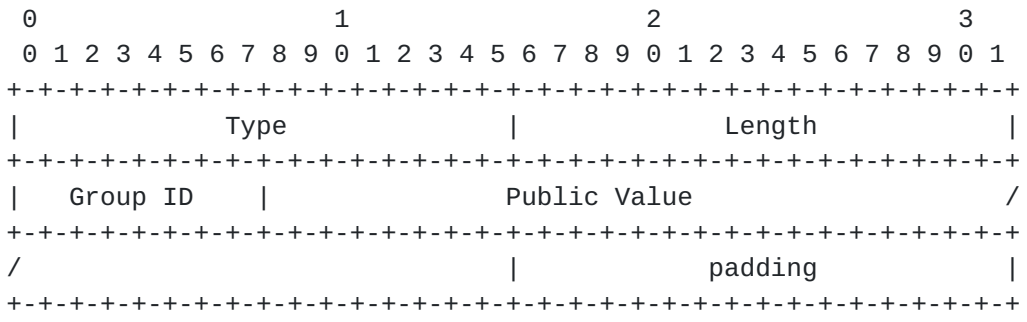


Type	7
Length	20
K	K is the number of verified bits
Reserved	zero when sent, ignored when received
Opaque	Copied unmodified from the received PUZZLE TLV
Random #I	random number
Puzzle solution #J	random number

Random #I, and Random #J are represented as 64-bit integers, K as 8-bit integer, all in network byte order.

The SOLUTION parameter contains a solution to a puzzle. It also echoes back the random difficulty K, the Opaque field, and the puzzle integer #I.

6.2.7 DIFFIE_HELLMAN



Type	15
Length	length in octets, excluding Type, Length, and padding
Group ID	defines values for p and g
Public Value	the sender's public Diffie-Hellman key

The following Group IDs have been defined:

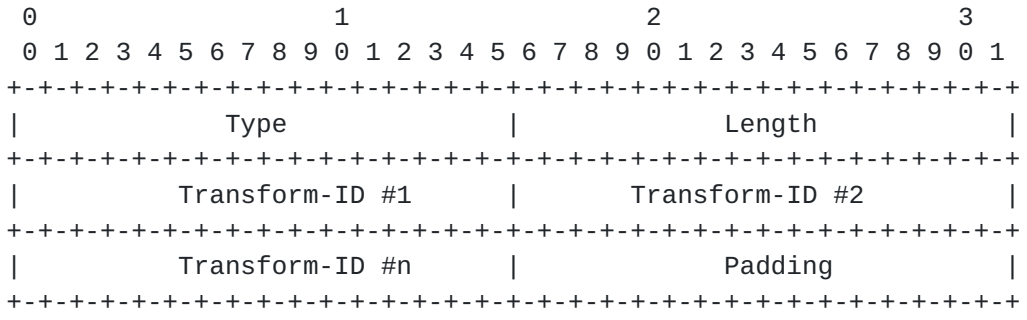
Group	Value
Reserved	0
384-bit group	1
OAKLEY well known group 1	2
1536-bit MODP group	3
3072-bit MODP group	4
6144-bit MODP group	5
8192-bit MODP group	6

The MODP Diffie-Hellman groups are defined in [18]. The OAKLEY group is defined in [9]. The OAKLEY well known group 5 is the same as the 1536-bit MODP group.

A HIP implementation MUST support Group IDs 1 and 3. The 384-bit group can be used when lower security is enough (e.g. web surfing) and when the equipment is not powerful enough (e.g. some PDAs). Equipment powerful enough SHOULD implement also group ID 5. The 384-bit group is defined in Appendix G.

To avoid unnecessary failures during the 4-way handshake, the rest of the groups SHOULD be implemented in hosts where resources are adequate.

6.2.8 HIP_TRANSFORM



Type 17
Length length in octets, excluding Type, Length, and padding
Transform-ID Defines the HIP Suite to be used

The Suite-IDs are identical to those defined in Section 6.2.9.

There MUST NOT be more than six (6) HIP Suite-IDs in one HIP transform TLV. The limited number of transforms sets the maximum size of HIP_TRANSFORM TLV. The HIP_TRANSFORM TLV MUST contain at least one of the mandatory Suite-IDs.

Mandatory implementations: ENCR-AES-CBC with HMAC-SHA1 and ENCR-NULL with HMAC-SHA1.

6.2.9 ESP_TRANSFORM

Type 19
Length length in octets, excluding Type, Length, and padding
E One if the ESP transform requires 64-bit sequence numbers
(see
Section 11.6
)
Reserved zero when sent, ignored when received

Suite-ID defines the ESP Suite to be used

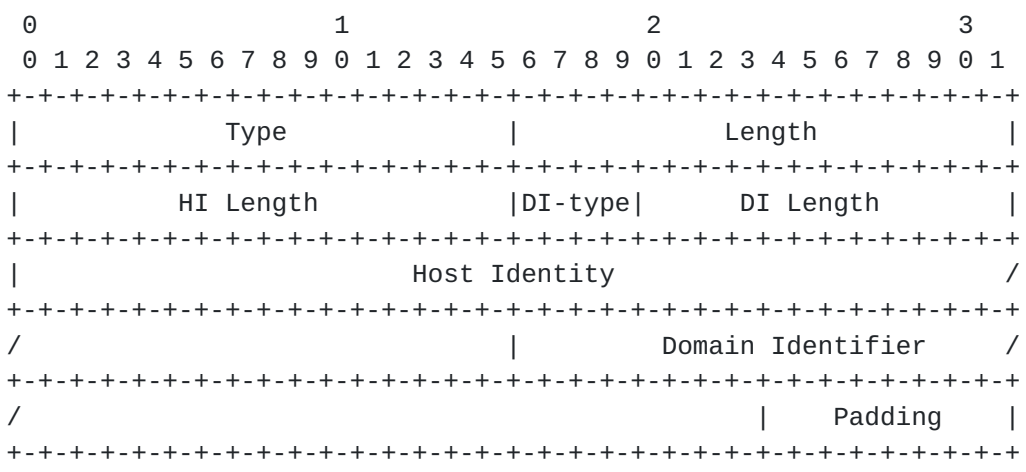
The following Suite-IDs are defined ([20],[23]):

Suite-ID	Value
RESERVED	0
ESP-AES-CBC with HMAC-SHA1	1
ESP-3DES-CBC with HMAC-SHA1	2
ESP-3DES-CBC with HMAC-MD5	3
ESP-BLOWFISH-CBC with HMAC-SHA1	4
ESP-NULL with HMAC-SHA1	5
ESP-NULL with HMAC-MD5	6

There MUST NOT be more than six (6) ESP Suite-IDs in one ESP_TRANSFORM TLV. The limited number of Suite-IDs sets the maximum size of ESP_TRANSFORM TLV. The ESP_TRANSFORM MUST contain at least one of the mandatory Suite-IDs.

Mandatory implementations: ESP-AES-CBC with HMAC-SHA1 and ESP-NULL with HMAC-SHA1.

6.2.10 HOST_ID



Type	35
Length	length in octets, excluding Type, Length, and Padding
DI-type	type of the following Domain Identifier field
DI Length	length of the FQDN or NAI in octets
N	if set, the following FQDN/NAI field contains a NAI
Host Identity	actual host identity
Domain Identifier	the identifier of the sender

The Host Identity is represented in RFC2535 [12] format. The algorithms used in RDATA format are the following:

Algorithms	Values
RESERVED	0
DSA	3 [RFC2536] (RECOMMENDED)
RSA	5 [RFC3110] (REQUIRED)

The following DI-types have been defined:

Type	Value
none included	0
FQDN	1
NAI	2

FQDN	Fully Qualified Domain Name, in binary format.
NAI	Network Access Identifier, in binary format. The format of the NAI is login@FQDN.

The format for the FQDN is defined in RFC1035 [3] Section 3.1.

If there is no Domain Identifier, i.e. the DI-type field is zero, also the DI Length field is set to zero.

6.2.11 CERT

Type	64
Length	length in octets, excluding Type, Length, and padding
Cert count	total count of certificates that are sent, possibly in several consecutive CER packets
Cert ID	the order number for this certificate
Cert Type	describes the type of the certificate

The receiver must know the total number (Cert count) of certificates

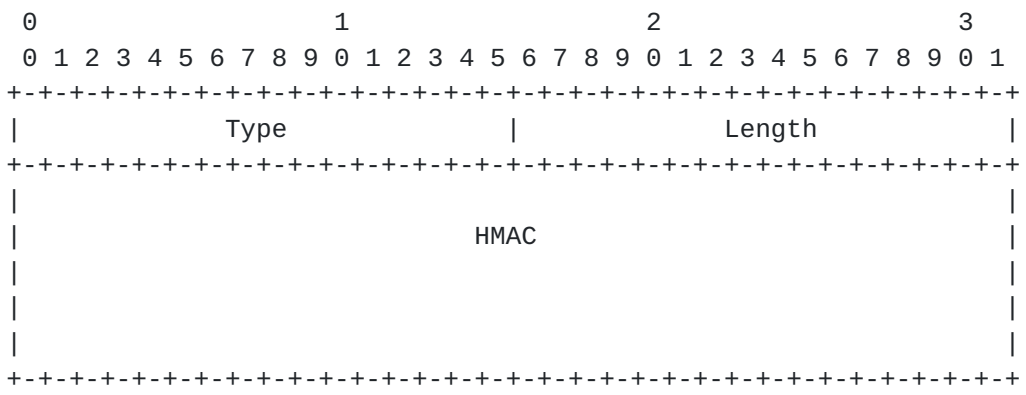
that it will receive from the sender, related to the R1 or I2. The Cert ID identifies the particular certificate and its order in the certificate chain. The numbering in Cert ID MUST go from 1 to Cert count.

The following certificate types are defined:

Cert format	Type number
X.509 v3	1

The encoding format for X.509v3 certificate is defined in [15].

6.2.12 HMAC



Type	65245
Length	20
HMAC	160 low order bits of the HMAC computed over the HIP packet, excluding the HMAC parameter and any following HIP_SIGNATURE or HIP_SIGNATURE_2 parameters. The checksum field MUST be set to zero and the HIP header length in the HIP common header MUST be calculated not to cover any excluded parameters when the HMAC is calculated.

The HMAC calculation and verification process is presented in Section 8.3.1

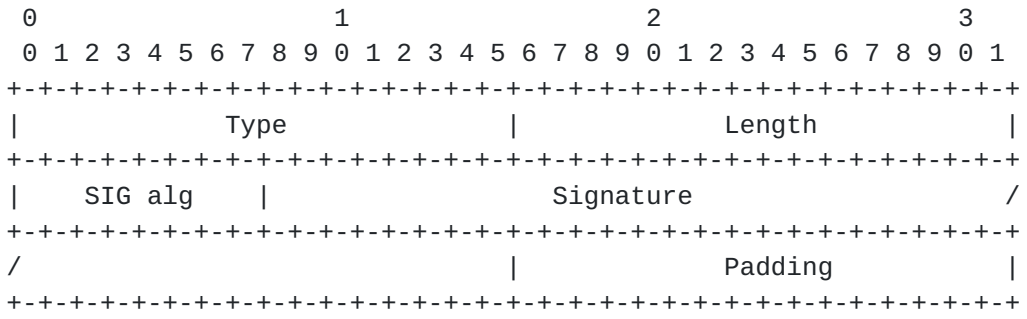
6.2.13 HMAC_2

The TLV structure is the same as in Section 6.2.12. The fields are:

Type	65247
Length	20
HMAC	160 low order bits of the HMAC computed over the HIP packet, excluding the HMAC parameter and any following HIP_SIGNATURE or HIP_SIGNATURE_2 parameters and including an additional sender's HOST_ID TLV during the HMAC calculation. The checksum field MUST be set to zero and the HIP header length in the HIP common header MUST be calculated not to cover any excluded parameters when the HMAC is calculated.

The HMAC calculation and verification process is presented in Section 8.3.1

6.2.14 HIP_SIGNATURE



Type	65279 (2^16-2^8-1)
Length	length in octets, excluding Type, Length, and Padding
SIG alg	Signature algorithm
Signature	the signature is calculated over the HIP packet, excluding the HIP_SIGNATURE TLV field and any TLVs that follow the HIP_SIGNATURE TLV. The checksum field MUST be set to zero, and the HIP header length in the HIP common header MUST be calculated only to the beginning of the HIP_SIGNATURE TLV when the signature is calculated.

The signature algorithms are defined in Section 6.2.10. The signature in the Signature field is encoded using the proper method depending on the signature algorithm (e.g. according to [14] in case of RSA, or according to [13] in case of DSA).

The HIP_SIGNATURE calculation and verification process is presented in Section 8.3.2

6.2.15 HIP_SIGNATURE_2

The TLV structure is the same as in Section 6.2.14. The fields are:

Type	65277 (2^16-2^8-3)
Length	length in octets, excluding Type, Length, and Padding
SIG alg	Signature algorithm
Signature	the signature is calculated over the HIP R1 packet, excluding the HIP_SIGNATURE_2 TLV field and any TLVs that follow the HIP_SIGNATURE_2 TLV. Initiator's HIT, checksum field, and the Opaque and Random #I fields in the PUZZLE TLV MUST be set to zero while computing the HIP_SIGNATURE_2 signature. Further, the HIP packet length in the HIP header MUST be calculated to the beginning of the HIP_SIGNATURE_2 TLV when the signature is calculated.

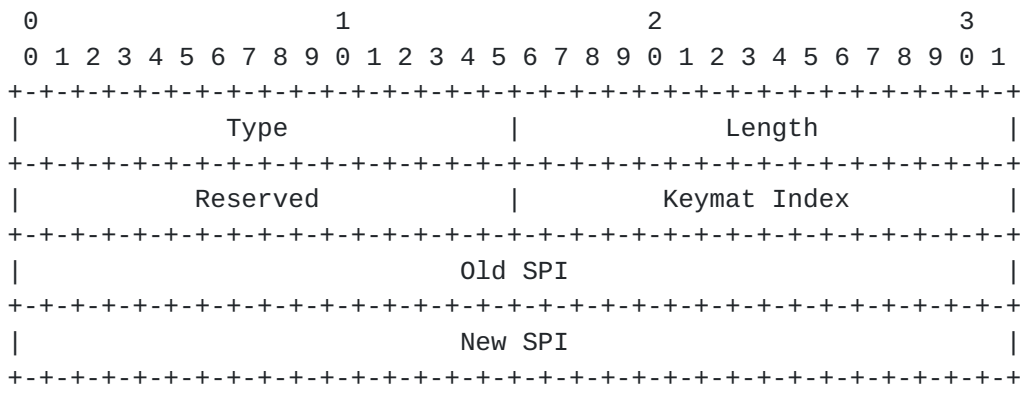
Zeroing the Initiator's HIT makes it possible to create R1 packets beforehand to minimize the effects of possible DoS attacks. Zeroing the I and Opaque fields allows these fields to be populated dynamically on precomputed R1s.

Signature calculation and verification follows the process in Section 8.3.2.

6.2.16 NES

During the life of an SA established by HIP, one of the hosts may need to reset the Sequence Number to one (to prevent wrapping) and rekey. The reason for rekeying might be an approaching sequence number wrap in ESP, or a local policy on use of a key. Rekeying ends the current SAs and starts new ones on both peers.

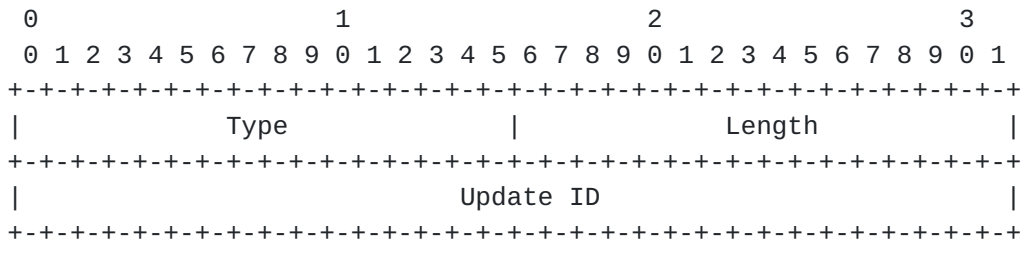
The NES parameter is carried in the HIP UPDATE packet. It is used to reset Security Associations. It introduces a new SPI to be used when sending data to the sender of the UPDATE packet. The keys for the new Security Association will be drawn from KEYMAT. If the packet contains a Diffie-Hellman parameter, the KEYMAT is first recomputed before drawing the new keys.



Type	9
Length	12
Keymat Index	Index, in bytes, where to continue to draw ESP keys from KEYMAT. If the packet includes a new Diffie-Hellman key, the field MUST be zero. Note that the length of this field limits the amount of keying material that can be drawn from KEYMAT. If that amount is exceeded, the NES packet MUST contain a new Diffie-Hellman key.
Old SPI	Old SPI for data sent to the source address of this packet
New SPI	New SPI for data sent to the source address of this packet

A host that receives an NES must reply shortly thereafter with an NES. Any middleboxes between the communicating hosts will learn the mappings from the pair of UPDATE messages.

6.2.17 SEQ

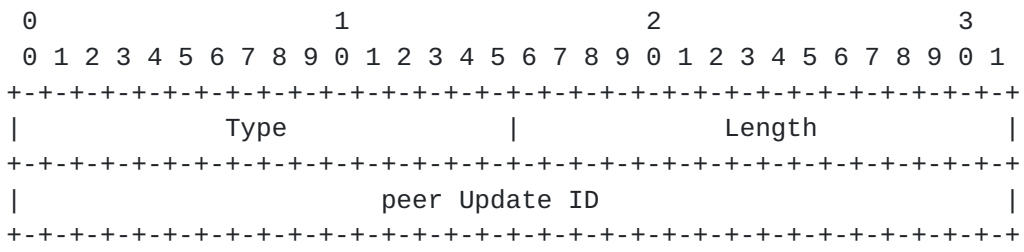


Type	11
Length	4
Update ID	32-bit sequence number

The Update ID is an unsigned quantity, initialized by a host to zero upon moving to ESTABLISHED state. The Update ID has scope within a single HIP association, and not across multiple associations or multiple hosts. The Update ID is incremented by one before each new

UPDATE that is sent by the host (i.e., the first UPDATE packet originated by a host has an Update ID of 1).

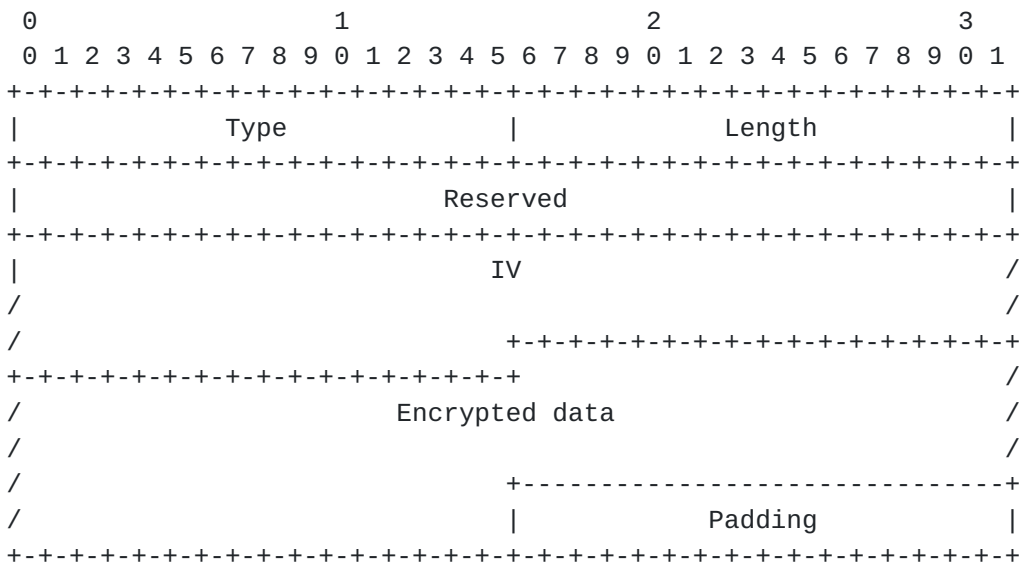
6.2.18 ACK



Type	13
Length	variable (multiple of 4)
peer Update ID	32-bit sequence number corresponding to the Update ID being acked.

The ACK parameter includes one or more Update IDs that have been received from the peer. The Length field identifies the number of peer Update IDs that are present in the parameter.

6.2.19 ENCRYPTED



Type	21
Length	length in octets, excluding Type, Length, and Padding
Reserved	zero when sent, ignored when received
IV	Initialization vector, if needed, otherwise nonexistent. The length of the IV is inferred from the HIP transform.
Encrypted data	The data is encrypted using an encryption algorithm as defined in HIP transform.
Padding	Any Padding, if necessary, to make the TLV a multiple of 8 bytes.

The encrypted data is in TLV format itself. Consequently, the first fields in the contents are Type and Length, allowing the contents to be easily parsed after decryption. Each of the TLVs to be encrypted, must be padded according to rules in Section 6.2.1 before encryption.

If the encryption algorithm requires the length of the data to be encrypted to be a multiple of the cipher algorithm block size, thereby necessitating padding, and if the encryption algorithm does not specify the padding contents, then an implementation MUST append the TLV parameter that is to be encrypted with an additional padding, so that the length of the resulting cleartext is a multiple of the cipher block size length. Such a padding MUST be constructed as specified in [19] Section 2.4. On the other hand, if the data to be encrypted is already a multiple of the block size, or if the encryption algorithm does specify padding as per [19] Section 2.4, then such additional padding SHOULD NOT be added.

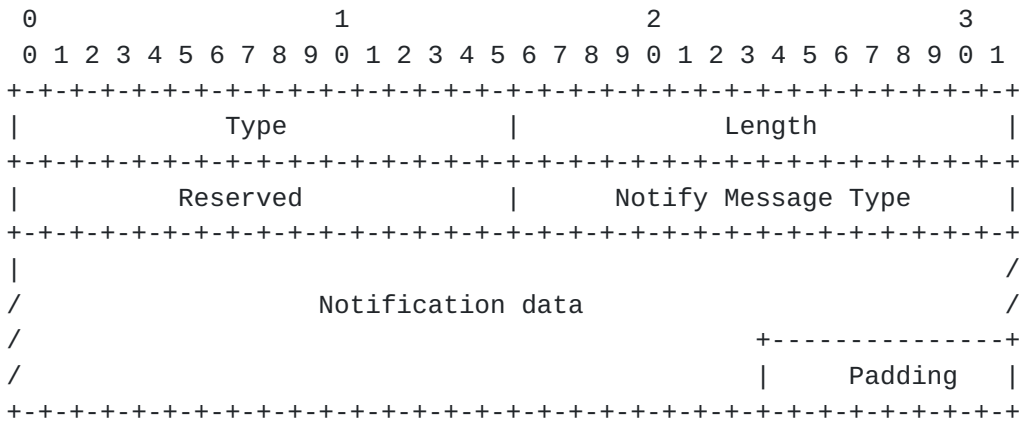
The Length field in the inside, to be encrypted TLV does not include the padding. The Length field in the outside ENCRYPTED TLV is the

length of the data after encryption (including the Reserved field, the IV field, and the output from the encryption process specified for that suite, but not any additional external padding). Note that the length of the cipher suite output may be smaller or larger than the length of the data to be encrypted, since the encryption process may compress the data or add additional padding to the data.

The ENCRYPTED payload may contain additional external padding, if the result of encryption, the TLV header and the IV is not a multiple of 8 bytes. The contents of this external padding MUST follow the rules given in Section 6.2.1.

6.2.20 NOTIFY

The NOTIFY parameter is used to transmit informational data, such as error conditions and state transitions, to a HIP peer. A NOTIFY parameter may appear in the NOTIFY packet type. The use of the NOTIFY parameter in other packet types is for further study.



Type	256
Length	length in octets, excluding Type, Length, and Padding
Reserved	zero when sent, ignored when received
Notify Message Type	Specifies the type of notification
Notification Data	Informational or error data transmitted in addition to the Notify Message Type. Values for this field are type specific (see below).
Padding	Any Padding, if necessary, to make the TLV a multiple of 8 bytes.

Notification information can be error messages specifying why an SA could not be established. It can also be status data that a process managing an SA database wishes to communicate with a peer process. The table below lists the Notification messages and their corresponding values.

To avoid certain types of attacks, a Responder SHOULD avoid sending a NOTIFY to any host with which it has not successfully verified a puzzle solution.

Types in the range 0 - 16383 are intended for reporting errors. An implementation that receives a NOTIFY error parameter in response to a request packet (e.g., I1, I2, UPDATE), SHOULD assume that the corresponding request has failed entirely. Unrecognized error types MUST be ignored except that they SHOULD be logged.

Notify payloads with status types MUST be ignored if not recognized.

NOTIFY PARAMETER - ERROR TYPES	Value
UNSUPPORTED_CRITICAL_PARAMETER_TYPE	1

Sent if the parameter type has the "critical" bit set and the parameter type is not recognized. Notification Data contains the two octet parameter type.

INVALID_SYNTAX	7

Indicates that the HIP message received was invalid because some type, length, or value was out of range or because the request was rejected for policy reasons. To avoid a denial of service attack using forged messages, this status may only be returned for and in an encrypted packet if the message ID and cryptographic checksum were valid. To avoid leaking information to someone probing a node, this status MUST be sent in response to any error not covered by one of the other status types. To aid debugging, more detailed error information SHOULD be written to a console or log.

NO_DH_PROPOSAL_CHOSEN	14

None of the proposed group IDs was acceptable.

INVALID_DH_CHOSEN	15

The D-H Group ID field does not correspond to one offered by the responder.

NO_HIP_PROPOSAL_CHOSEN	16

None of the proposed HIP Transform crypto suites was acceptable.

INVALID_HIP_TRANSFORM_CHOSEN 17

The HIP Transform crypto suite does not correspond to one offered by the responder.

NO_ESP_PROPOSAL_CHOSEN 18

None of the proposed ESP Transform crypto suites was acceptable.

INVALID_ESP_TRANSFORM_CHOSEN 19

The ESP Transform crypto suite does not correspond to one offered by the responder.

AUTHENTICATION_FAILED 24

Sent in response to a HIP signature failure.

CHECKSUM_FAILED 26

Sent in response to a HIP checksum failure.

HMAC_FAILED 28

Sent in response to a HIP HMAC failure.

ENCRYPTION_FAILED 32

The responder could not successfully decrypt the ENCRYPTED TLV.

INVALID_HIT 40

Sent in response to a failure to validate the peer's HIT from the corresponding HI.

BLOCKED_BY_POLICY 42

The responder is unwilling to set up an association for some policy reason (e.g. received HIT is NULL and policy does not allow opportunistic mode).

SERVER_BUSY_PLEASE_RETRY 44

The responder is unwilling to set up an association as it is suffering under some kind of overload and has chosen to shed load by rejecting your request. You may retry if you wish, however you MUST find another (different) puzzle solution for any such retries. Note that you may need to obtain a new puzzle with a new I1/R1 exchange.

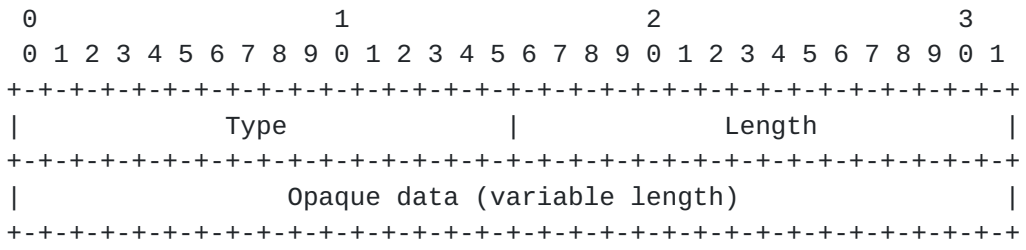
I2_ACKNOWLEDGEMENT	46

The responder has received your I2 but had to queue the I2 for processing. The puzzle was correctly solved and the responder is willing to set up an association but has currently a number of I2s in processing queue. R2 will be sent after the I2 has been processed.

NOTIFY MESSAGES - STATUS TYPES	Value

(None defined at present)

6.2.21 ECHO_REQUEST

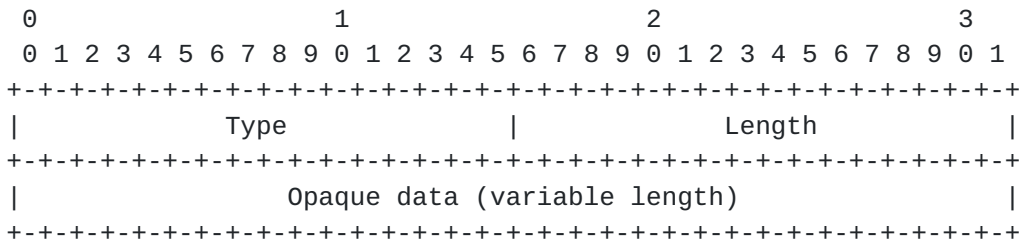


Type	65281 or 1022
Length	variable
Opaque data	Opaque data, supposed to be meaningful only to the node that sends ECHO_REQUEST and receives a corresponding ECHO_RESPONSE.

The ECHO_REQUEST parameter contains an opaque blob of data that the sender wants to get echoed back in the corresponding reply packet.

The ECHO_REQUEST and ECHO_RESPONSE parameters MAY be used for any purpose where a node wants to carry some state in a request packet and get it back in a response packet. The ECHO_REQUEST MAY be covered by the HMAC and SIGNATURE. This is dictated by the Type field selected for the parameter; Type 1022 ECHO_REQUEST is covered and Type 65281 is not.

6.2.22 ECHO_RESPONSE



Type	65283 or 1024
Length	variable
Opaque data	Opaque data, copied unmodified from the ECHO_REQUEST parameter that triggered this response.

The ECHO_RESPONSE parameter contains an opaque blob of data that the sender of the ECHO_REQUEST wants to get echoed back. The opaque data is copied unmodified from the ECHO_REQUEST parameter.

The ECHO_REQUEST and ECHO_RESPONSE parameters MAY be used for any purpose where a node wants to carry some state in a request packet and get it back in a response packet. The ECHO_RESPONSE MAY be covered by the HMAC and SIGNATURE. This is dictated by the Type field selected for the parameter; Type 1024 ECHO_RESPONSE is covered and Type 65283 is not.

6.3 ICMP messages

When a HIP implementation detects a problem with an incoming packet, and it either cannot determine the identity of the sender of the packet or does not have any existing HIP security association with the sender of the packet, it MAY respond with an ICMP packet. Any such replies MUST be rate limited as described in [4]. In most cases, the ICMP packet will have the Parameter Problem type (12 for ICMPv4, 4 for ICMPv6), with the Pointer field pointing to the field that caused the ICMP message to be generated.

XXX: Should we say something more about rate limitation here?

6.3.1 Invalid Version

If a HIP implementation receives a HIP packet that has an unrecognized HIP version number, it SHOULD respond, rate limited, with an ICMP packet with type Parameter Problem, the Pointer pointing to the VER./RES. byte in the HIP header.

6.3.2 Other problems with the HIP header and packet structure

If a HIP implementation receives a HIP packet that has other unrecoverable problems in the header or packet format, it MAY respond, rate limited, with an ICMP packet with type Parameter Problem, the Pointer pointing to the field that failed to pass the format checks. However, an implementation MUST NOT send an ICMP message if the Checksum fails; instead, it MUST silently drop the packet.

6.3.3 Unknown SPI

If a HIP implementation receives an ESP packet that has an unrecognized SPI number, it MAY responder, rate limited, with an ICMP packet with type Parameter Problem, the Pointer pointing to the the beginning of SPI field in the ESP header.

6.3.4 Invalid Cookie Solution

If a HIP implementation receives an I2 packet that has an invalid cookie solution, the behaviour depends on the underlying version of IP. If IPv6 is used, the implementation SHOULD respond with an ICMP packet with type Parameter Problem, the Pointer pointing to the beginning of the Puzzle solution #J field in the SOLUTION payload in the HIP message.

If IPv4 is used, the implementation MAY respond with an ICMP packet with the type Parameter Problem, copying enough of bytes form the I2 message so that the SOLUTION parameter fits in to the ICMP message, the Pointer pointing to the beginning of the Puzzle solution #J field, as in the IPv6 case. Note, however, that the resulting ICMPv4 message exceeds the typical ICMPv4 message size as defined in [2].

6.3.5 Non-existing HIP association

If a HIP implementation receives a CLOSE, or UPDATE packet, or any other packet whose handling requires an existing association, that has either a Receiver or Sender HIT that does not match with any existing HIP association, the implementation MAY respond, rate limited, with an ICMP packet with the type Parameter Problem, the Pointer pointing to the the beginning of the first HIT that does not match.

A host MUST NOT reply with such an ICMP if it receives any of the following messages: I1, R2, I2, R2, CER, and NOTIFY. When introducing new packet types, a specification SHOULD define the appropriate rules for sending or not sending this kind of ICMP replies.

7. HIP Packets

There are nine basic HIP packets. Four are for the base HIP exchange, one is for updating, one is a broadcast for use when there is no IP addressing (e.g., before DHCP exchange), one is used to send certificates, one for sending notifications, and one is for sending unencrypted data.

Packets consist of the fixed header as described in Section 6.1, followed by the parameters. The parameter part, in turn, consists of zero or more TLV coded parameters.

In addition to the base packets, other packets types will be defined later in separate specifications. For example, support for mobility and multi-homing is not included in this specification.

Packet representation uses the following operations:

```
()      parameter
x{y}    operation x on content y
<x>i    x exists i times
[]      optional parameter
x | y   x or y
```

In the future, an OPTIONAL upper layer payload MAY follow the HIP header. The payload proto field in the header indicates if there is additional data following the HIP header. The HIP packet, however, MUST NOT be fragmented. This limits the size of the possible additional data in the packet.

7.1 I1 - the HIP initiator packet

The HIP header values for the I1 packet:

```
Header:
  Packet Type = 1
  SRC HIT = Initiator's HIT
  DST HIT = Responder's HIT, or NULL

IP ( HIP () )
```

The I1 packet contains only the fixed HIP header.

Valid control bits: none

The Initiator gets the Responder's HIT either from a DNS lookup of the Responder's FQDN, from some other repository, or from a local table. If the Initiator does not know the Responder's HIT, it may

attempt opportunistic mode by using NULL (all zeros) as the Responder's HIT. If the Initiator send a NULL as the Responder's HIT, it MUST be able to handle all MUST and SHOULD algorithms from Section 3, which are currently RSA and DSA.

Since this packet is so easy to spoof even if it were signed, no attempt is made to add to its generation or processing cost.

Implementation MUST be able to handle a storm of received I1 packets, discarding those with common content that arrive within a small time delta.

7.2 R1 - the HIP responder packet

The HIP header values for the R1 packet:

```
Header:
  Packet Type = 2
  SRC HIT = Responder's HIT
  DST HIT = Initiator's HIT

IP ( HIP ( [ R1_COUNTER, ]
           PUZZLE,
           DIFFIE_HELLMAN,
           HIP_TRANSFORM,
           ESP_TRANSFORM,
           HOST_ID,
           [ ECHO_REQUEST, ]
           HIP_SIGNATURE_2 )
           [, ECHO_REQUEST ])
```

Valid control bits: C, A

The R1 packet may be followed by one or more CER packets. In this case, the C-bit in the control field MUST be set.

If the responder HI is an anonymous one, the A control MUST be set.

The initiator HIT MUST match the one received in I1. If the Responder has multiple HIs, the responder HIT used MUST match Initiator's request. If the Initiator used opportunistic mode, the Responder may select freely among its HIs.

The R1 generation counter is used to determine the currently valid generation of puzzles. The value is increased periodically, and it is RECOMMENDED that it is increased at least as often as solutions to old puzzles are not accepted any longer.

The Puzzle contains a random #I and the difficulty K. The difficulty K is the number of bits that the Initiator must get zero in the puzzle. The random #I is not covered by the signature and must be zeroed during the signature calculation, allowing the sender to select and set the #I into a pre-computed R1 just prior sending it to the peer.

The Diffie-Hellman value is ephemeral, but can be reused over a number of connections. In fact, as a defense against I1 storms, an implementation MAY use the same Diffie-Hellman value for a period of time, for example, 15 minutes. By using a small number of different Cookies for a given Diffie-Hellman value, the R1 packets can be pre-computed and delivered as quickly as I1 packets arrive. A scavenger process should clean up unused DHs and Cookies.

The HIP_TRANSFORM contains the encryption and integrity algorithms supported by the Responder to protect the HI exchange, in the order of preference. All implementations MUST support the AES [10] with HMAC-SHA-1-96 [6].

The ESP_TRANSFORM contains the ESP modes supported by the Responder, in the order of preference. All implementations MUST support AES [10] with HMAC-SHA-1-96 [6].

The ECHO_REQUEST contains data that the sender wants to receive unmodified in the corresponding response packet in the ECHO_RESPONSE parameter. The ECHO_REQUEST can be either covered by the signature, or it can be left out from it. In the first case, the ECHO_REQUEST gets Type number 1022 and in the latter case 65281.

The signature is calculated over the whole HIP envelope, after setting the initiator HIT, header checksum as well as the Opaque field and the Random #I in the PUZZLE parameter temporarily to zero, and excluding any TLVs that follow the signature, as described in Section 6.2.15. This allows the Responder to use precomputed R1s. The Initiator SHOULD validate this signature. It SHOULD check that the responder HI received matches with the one expected, if any.

7.3 I2 - the second HIP initiator packet

The HIP header values for the I2 packet:

```
Header:
  Type = 3
  SRC HIT = Initiator's HIT
  DST HIT = Responder's HIT

IP ( HIP ( SPI,
           [R1_COUNTER,]
           SOLUTION,
           DIFFIE_HELLMAN,
           HIP_TRANSFORM,
           ESP_TRANSFORM,
           ENCRYPTED { HOST_ID },
           [ ECHO_RESPONSE ,]
           HMAC,
           HIP_SIGNATURE
           [, ECHO_RESPONSE] ) )
```

Valid control bits: C, A

The HITs used MUST match the ones used previously.

If the initiator HI is an anonymous one, the A control MUST be set.

The Initiator MAY include an unmodified copy of the R1_COUNTER parameter received in the corresponding R1 packet into the I2 packet.

The Solution contains the random # I from R1 and the computed # J. The low order K bits of the SHA-1(I | ... | J) MUST be zero.

The Diffie-Hellman value is ephemeral. If precomputed, a scavenger process should clean up unused DHs.

The HIP_TRANSFORM contains the encryption and integrity used to protect the HI exchange selected by the Initiator. All implementations MUST support the AES transform [10].

The Initiator's HI is encrypted using the HIP_TRANSFORM encryption algorithm. The keying material is derived from the Diffie-Hellman exchanged as defined in Section 9.

The ESP_TRANSFORM contains the ESP mode selected by the Initiator. All implementations MUST support AES [10] with HMAC-SHA-1-96 [6].

The ECHO_RESPONSE contains the the unmodified Opaque data copied from the corresponding ECHO_REQUEST TLV. The ECHO_RESPONSE can be either covered by the signature, or it can be left out from it. In the first case, the ECHO_RESPONSE gets Type number 1024 and in the latter case 65283.

The HMAC is calculated over whole HIP envelope, excluding any TLVs after the HMAC, as described in Section 8.3.1. The Responder MUST validate the HMAC.

The signature is calculated over whole HIP envelope, excluding any TLVs after the HIP_SIGNATURE, as described in Section 6.2.14. The Responder MUST validate this signature. It MAY use either the HI in the packet or the HI acquired by some other means.

7.4 R2 - the second HIP responder packet

The HIP header values for the R2 packet:

```
Header:
  Packet Type = 4
  SRC HIT = Responder's HIT
  DST HIT = Initiator's HIT

IP ( HIP ( SPI, HMAC_2, HIP_SIGNATURE ) )
```

Valid control bits: none

The HMAC_2 is calculated over whole HIP envelope, with Responder's HOST_ID TLV concatenated with the HIP envelope. The HOST_ID TLV is removed after the HMAC calculation. The procedure is described in 8.3.1.

The signature is calculated over whole HIP envelope.

The Initiator MUST validate both the HMAC and the signature.

7.5 CER - the HIP Certificate Packet

The CER packet is OPTIONAL.

The Optional CER packets over the Announcer's HI by a higher level authority known to the Recipient is an alternative method for the Recipient to trust the Announcer's HI (over DNSSEC or PKI).

The HIP header values for CER packet:

```
Header:
  Packet Type = 5
  SRC HIT = Announcer's HIT
  DST HIT = Recipient's HIT

IP ( HIP ( <CERT>i , HIP_SIGNATURE ) )  or
```

IP (HIP (ENCRYPTED { <CERT>i }, HIP_SIGNATURE))

Valid control bits: None

Certificates in the CER packet MAY be encrypted. The encryption algorithm is provided in the HIP transform of the previous (R1 or I2) packet.

7.6 UPDATE - the HIP Update Packet

Support for the UPDATE packet is MANDATORY.

The HIP header values for the UPDATE packet:

Header:
Packet Type = 6
SRC HIT = Sender's HIT
DST HIT = Recipient's HIT

IP (HIP ([NES, SEQ, ACK, DIFFIE_HELLMAN,] HMAC, HIP_SIGNATURE))

Valid control bits: None

The UPDATE packet contains mandatory HMAC and HIP_SIGNATURE parameters, and other optional parameters.

The UPDATE packet contains zero or one SEQ parameter. The presence of a SEQ parameter indicates that the receiver MUST ack the UPDATE. An UPDATE that does not contain a SEQ parameter is simply an ACK of a previous UPDATE and itself MUST not be acked.

An UPDATE packet contains zero or one ACK parameters. The ACK parameter echoes the SEQ sequence number of the UPDATE packet being acked. A host MAY choose to ack more than one UPDATE packet at a time; e.g., the ACK may contain the last two SEQ values received, for robustness to ack loss. ACK values are not cumulative; each received unique SEQ value requires at least one corresponding ACK value in reply. Received ACKs that are redundant are ignored.

The UPDATE packet may contain both a SEQ and an ACK parameter. In this case, the ACK is being piggybacked on an outgoing UPDATE. In general, UPDATEs carrying SEQ SHOULD be acked upon completion of the processing of the UPDATE. A host MAY choose to hold the UPDATE carrying ACK for a short period of time to allow for the possibility of piggybacking the ACK parameter, in a manner similar to TCP delayed acknowledgments.

A sender MAY choose to forego reliable transmission of a particular

UPDATE (e.g., it becomes overcome by events). The semantics are such that the receiver MUST acknowledge the UPDATE but the sender MAY choose to not care about receiving the ACK.

UPDATEs MAY be retransmitting without incrementing SEQ. If the same subset of parameters is included in multiple UPDATEs with different SEQs, the host MUST ensure that receiver processing of the parameters multiple times will not result in a protocol error.

In the case of rekeying (Section 8.10), the UPDATE packet MUST carry NES and MAY carry DIFFIE_HELLMAN parameter, unless the UPDATE is a bare ack.

Intermediate systems that use the SPI will have to inspect HIP packets for a UPDATE packet. The packet is signed for the benefit of the intermediate systems. Since intermediate systems may need the new SPI values, the contents of this packet cannot be encrypted.

7.7 NOTIFY - the HIP Notify Packet

The NOTIFY packet is OPTIONAL. The NOTIFY packet MAY be used to provide information to a peer. Typically, NOTIFY is used to indicate some type of protocol error or negotiation failure.

The HIP header values for the NOTIFY packet:

```
Header:
  Packet Type = 7
  SRC HIT = Sender's HIT
  DST HIT = Recipient's HIT, or zero if unknown

IP ( HIP (<NOTIFY>i, [HOST_ID, ] HIP_SIGNATURE) )
```

Valid control bits: None

The NOTIFY packet is used to carry one or more NOTIFY parameters.

7.8 CLOSE - the HIP association closing packet

The HIP header values for the CLOSE packet:

```
Header:
  Packet Type = 8
  SRC HIT = Sender's HIT
  DST HIT = Recipient's HIT

IP ( HIP ( ECHO_REQUEST, HMAC, HIP_SIGNATURE ) )
```

Valid control bits: none

The sender MUST include an ECHO_REPLY used to validate CLOSE_ACK received in response, and both an HMAC and a signature (calculated over the whole HIP envelope).

The receiver peer MUST validate both the HMAC and the signature if it has a HIP association state, and MUST reply with a CLOSE_ACK containing an ECHO_REPLY corresponding to the received ECHO_REQUEST.

7.9 CLOSE_ACK - the HIP closing acknowledgment packet

The HIP header values for the CLOSE_ACK packet:

```
Header:
  Packet Type = 9
  SRC HIT = Sender's HIT
  DST HIT = Recipient's HIT

IP ( HIP ( ECHO_REPLY, HMAC, HIP_SIGNATURE ) )
```

Valid control bits: none

The sender MUST include both an HMAC and signature (calculated over the whole HIP envelope).

The receiver peer MUST validate both the HMAC and the signature.

8. Packet processing

Each host is assumed to have a single HIP protocol implementation that manages the host's HIP associations and handles requests for new ones. Each HIP association is governed by a conceptual state machine, with states defined above in Section 5.4. The HIP implementation can simultaneously maintain HIP associations with more than one host. Furthermore, the HIP implementation may have more than one active HIP association with another host; in this case, HIP associations are distinguished by their respective HITs and IPsec SPIs. It is not possible to have more than one HIP associations between any given pair of HITs. Consequently, the only way for two hosts to have more than one parallel association is to use different HITs, at least at one end.

The processing of packets depends on the state of the HIP association(s) with respect to the authenticated or apparent originator of the packet. A HIP implementation determines whether it has an active association with the originator of the packet based on the HITs or the SPI of the packet.

8.1 Processing outgoing application data

In a HIP host, an application can send application level data using HITs or LSIs as source and destination identifiers. The HITs and LSIs may be specified via a backwards compatible API (see Appendix A) or a completely new API. However, whenever there is such outgoing data, the stack has to protect the data with ESP, and send the resulting datagram using appropriate source and destination IP addresses. Here, we specify the processing rules only for the base case where both hosts have only single usable IP addresses; the multi-address multi-homing case will be specified separately.

If the IPv4 or IPv6 backward compatible APIs and therefore LSIs are supported, it is assumed that the LSIs will be converted into proper HITs somewhere in the stack. The exact location of the conversion is an implementation specific issue and not discussed here. The following conceptual algorithm discusses only HITs, with the assumption that the LSI-to-HIT conversion takes place somewhere.

The following steps define the conceptual processing rules for outgoing datagrams destined to a HIT.

1. If the datagram has a specified source address, it MUST be a HIT. If it is not, the implementation MAY replace the source address with a HIT. Otherwise it MUST drop the packet.
2. If the datagram has an unspecified source address, the implementation must choose a suitable source HIT for the datagram. In selecting a proper local HIT, the implementation

SHOULD consult the table of currently active HIP sessions, and preferably select a HIT that already has an active session with the target HIT.

3. If there no active HIP session with the given < source, destination > HIT pair, one must be created by running the base exchange. The implementation SHOULD queue at least one packet per HIP session to be formed, and it MAY queue more than one.
4. Once there is an active HIP session for the given < source, destination > HIT pair, the outgoing datagram is protected using the associated ESP security association. In a typical implementation, this will result in an transport mode ESP datagram that still has HITs in the place of IP addresses.
5. The HITs in the datagram are replaced with suitable IP addresses. For IPv6, the rules defined in [16] SHOULD be followed. Note that this HIT-to-IP-address conversion step MAY also be performed at some other point in the stack, e.g., before ESP processing. However, care must be taken to make sure that the right ESP SA is employed.

8.2 Processing incoming application data

Incoming HIP datagrams arrive as ESP protected packets. In the usual case the receiving host has a corresponding ESP security association, identified by the SPI and destination IP address in the packet. However, if the host has crashed or otherwise lost its HIP state, it may not have such an SA.

The following steps define the conceptual processing rules for incoming ESP protected datagrams targeted to an ESP security association created with HIP.

1. Detect the proper IPsec SA using the SPI. If the resulting SA is a non-HIP ESP SA, process the packet according to standard IPsec rules. If there are no SAs identified with the SPI, the host MAY send an ICMP packet as defined in Section 6.3.3. How to handle lost state is an implementation issue.
2. If a proper HIP ESP SA is found, the packet is processed normally by ESP, as if the packet were a transport mode packet. The packet may be dropped by ESP, as usual. In a typical implementation, the result of successful ESP decryption and verification is a datagram with the original IP addresses as source and destination.
3. The IP addresses in the datagram are replaced with the HITs associated with the ESP SA. Note that this IP-address-to-HIT conversion step MAY also be performed at some other point in the stack, e.g., before ESP processing.
4. The datagram is delivered to the upper layer. Demultiplexing the datagram the right upper layer socket is based on the HITs (or LSIs).

8.3 HMAC and SIGNATURE calculation and verification

The following subsections define the actions for processing HMAC, HIP_SIGNATURE and HIP_SIGNATURE_2 TLVs.

8.3.1 HMAC calculation

The following process applies both to the HMAC and HMAC_2 TLVs. When processing HMAC_2, the difference is that the HMAC calculation includes pseudo HOST_ID field containing the Responder's information as sent in the R1 packet earlier.

The HMAC TLV is defined in Section 6.2.12 and HMAC_2 TLV in Section 6.2.13. HMAC calculation and verification process:

Packet sender:
1. Create the HIP packet, without the HMAC or any possible HIP_SIGNATURE or HIP_SIGNATURE_2 TLVs.
2. In case of HMAC_2 calculation, add a HOST_ID (Responder) TLV to the packet.
3. Calculate the Length field in the HIP header.
4. Compute the HMAC.
5. In case of HMAC_2, remove the HOST_ID TLV from the packet.
6. Add the HMAC TLV to the packet and any HIP_SIGNATURE or HIP_SIGNATURE_2 TLVs that may follow.
7. Recalculate the Length field in the HIP header.

Packet receiver:
1. Verify the HIP header Length field.
2. Remove the HMAC or HMAC_2 TLV, and if the packet contains any HIP_SIGNATURE or HIP_SIGNATURE_2 fields, remove them too, saving the contents if they will be needed later.
3. In case of HMAC_2, build and add a HOST_ID TLV (with Responder information) to the packet.
4. Recalculate the HIP packet length in the HIP header and clear the Checksum field (set it to all zeros).
5. Compute the HMAC and verify it against the received HMAC.
6. In case of HMAC_2, remove the HOST_ID TLV from the packet before further processing.

8.3.2 Signature calculation

The following process applies both to the HIP_SIGNATURE and HIP_SIGNATURE_2 TLVs. When processing HIP_SIGNATURE_2, the only difference is that instead of HIP_SIGNATURE TLV, the HIP_SIGNATURE_2 TLV is used, and the Initiator's HIT and PUZZLE Opaque and Random #I fields are cleared (set to all zeros) before computing the signature. The HIP_SIGNATURE TLV is defined in Section 6.2.14 and the

HIP_SIGNATURE_2 TLV in Section 6.2.15.

Signature calculation and verification process:

Packet sender:
1. Create the HIP packet without the HIP_SIGNATURE TLV or any TLVs that follow the HIP_SIGNATURE TLV.
2. Calculate the Length field in the HIP header.
3. Compute the signature.
4. Add the HIP_SIGNATURE TLV to the packet.
5. Add any TLVs that follow the HIP_SIGNATURE TLV.
6. Recalculate the Length field in the HIP header.

Packet receiver:
1. Verify the HIP header Length field.
2. Save the contents of the HIP_SIGNATURE TLV and any TLVs following the HIP_SIGNATURE TLV and remove them from the packet.
3. Recalculate the HIP packet Length in the HIP header and clear the Checksum field (set it to all zeros).
4. Compute the signature and verify it against the received signature.

The verification can use either the HI received from a HIP packet, the HI from a DNS query, if the FQDN has been received either in the HOST_ID or in the CER packet, or one received by some other means.

8.4 Initiation of a HIP exchange

An implementation may originate a HIP exchange to another host based on a local policy decision, usually triggered by an application datagram, in much the same way that an IPsec IKE key exchange can dynamically create a Security Association. Alternatively, a system may initiate a HIP exchange if it has rebooted or timed out, or otherwise lost its HIP state, as described in Section 5.3.

The implementation prepares an I1 packet and sends it to the IP address that corresponds to the peer host. The IP address of the peer host may be obtained via conventional mechanisms, such as DNS lookup. The I1 contents are specified in Section 7.1. The selection of which host identity to use, if a host has more than one to choose from, is typically a policy decision.

The following steps define the conceptual processing rules for initiating a HIP exchange:
1. The Initiator gets the Responder's HIT and one or more addresses either from a DNS lookup of the responder's FQDN, from some other repository, or from a local table. If the initiator does not know the responder's HIT, it may attempt opportunistic mode by using

NULL (all zeros) as the responder's HIT.
2. The Initiator sends an I1 to one of the Responder's addresses. The selection of which address to use is a local policy decision.
3. Upon sending an I1, the sender shall transition to state I1-SENT, start a timer whose timeout value should be larger than the worst-case anticipated RTT, and shall increment a timeout counter associated with the I1.
4. Upon timeout, the sender SHOULD retransmit the I1 and restart the timer, up to a maximum of I1_RETRIES_MAX tries.

8.4.1 Sending multiple I1s in parallel

For the sake of minimizing the session establishment latency, an implementation MAY send the same I1 to more than one of the Responder's addresses. However, it MUST NOT send to more than three (3) addresses in parallel. Furthermore, upon timeout, the implementation MUST refrain from sending the same I1 packet to multiple addresses. These limitations are placed order to avoid congestion of the network, and potential DoS attacks that might happen, e.g., because someone claims to have hundreds or thousands of addresses.

As the Responder is not guaranteed to distinguish the duplicate I1's it receives at several of its addresses (because it avoids to store states when it answers back an R1), the Initiator may receive several duplicate R1's.

The Initiator SHOULD then select the initial preferred destination address using the source address of the selected received R1, and use the preferred address as a source address for the I2. Processing rules for received R1s are discussed in Section 8.6.

8.4.2 Processing incoming ICMP Protocol Unreachable messages

A host may receive an ICMP Destination Protocol Unreachable message as a response to sending an HIP I1 packet. Such a packet may be an indication that the peer does not support HIP, or it may be an attempt to launch an attack by making the Initiator believe that the Responder does not support HIP.

When a system receives an ICMP Destination Protocol Unreachable message while it is waiting for an R1, it MUST NOT terminate the wait. It MAY continue as if it had not received the ICMP message, and send a few more I1s. Alternatively, it MAY take the ICMP message as a hint that the peer most probably does not support HIP, and return to state UNASSOCIATED earlier than otherwise. However, at minimum, it MUST continue waiting for an R1 for a reasonable time before returning to UNASSOCIATED.

8.5 Processing incoming I1 packets

An implementation SHOULD reply to an I1 with an R1 packet, unless the implementation is unable or unwilling to setup a HIP association. If the implementation is unable to setup a HIP association, the host SHOULD send an ICMP Destination Protocol Unreachable, Administratively Prohibited, message to the I1 source address. If the implementation is unwilling to setup a HIP association, the host MAY ignore the I1. This latter case may occur during a DoS attack such as an I1 flood.

The implementation MUST be able to handle a storm of received I1 packets, discarding those with common content that arrive within a small time delta.

A spoofed I1 can result in an R1 attack on a system. An R1 sender MUST have a mechanism to rate limit R1s to an address.

Under no circumstances does the HIP state machine transition upon sending an R1.

The following steps define the conceptual processing rules for responding to an I1 packet:

1. The responder MUST check that the responder HIT in the received I1 is either one of its own HITs, or NULL.
2. If the responder is in ESTABLISHED state, the responder MAY respond to this with an R1 packet, prepare to drop existing SAs and stay at ESTABLISHED state.
3. If the implementation chooses to respond to the I1 with and R1 packet, it creates a new R1 or selects a precomputed R1 according to the format described in Section 7.2.
4. The R1 MUST contain the received responder HIT, unless the received HIT is NULL, in which case the Responder SHOULD select a HIT that is constructed with the MUST algorithm in Section 3, which is currently RSA. Other than that, selecting the HIT is a local policy matter.
5. The responder sends the R1 to the source IP address of the I1 packet.

8.5.1 R1 Management

All compliant implementations MUST produce R1 packets. An R1 packet MAY be precomputed. An R1 packet MAY be reused for time Delta T, which is implementation dependent. R1 information MUST not be discarded until Delta S after T. Time S is the delay needed for the last I2 to arrive back to the responder.

An implementation MAY keep state about received I1s and match the

received I2s against the state, as discussed in Section 4.1.1.

8.5.2 Handling malformed messages

If an implementation receives a malformed I1 message, it SHOULD NOT respond with a NOTIFY message, as such practice could open up a potential denial-of-service danger. Instead, it MAY respond with an ICMP packet, as defined in Section 6.3.

8.6 Processing incoming R1 packets

A system receiving an R1 MUST first check to see if it has sent an I1 to the originator of the R1 (i.e., it is in state I1-SENT). If so, it SHOULD process the R1 as described below, send an I2, and go to state I2-SENT, setting a timer to protect the I2. If the system is in state I2-SENT, it MAY respond to an R1 if the R1 has a larger R1 generation counter; if so, it should drop its state due to processing the previous R1 and start over from state I1-SENT. If the system is in any other state with respect to that host, it SHOULD silently drop the R1.

When sending multiple I1s, an initiator SHOULD wait for a small amount of time after the first R1 reception to allow possibly multiple R1s to arrive, and it SHOULD respond to an R1 among the set with the largest R1 generation counter.

The following steps define the conceptual processing rules for responding to an R1 packet:

1. A system receiving an R1 MUST first check to see if it has sent an I1 to the originator of the R1 (i.e., it has a HIP association that is in state I1-SENT and that is associated with the HITs in the R1). If so, it should process the R1 as described below.
2. Otherwise, if the system is in any other state than I1-SENT or I2-SENT with respect to the HITs included in the R1, it SHOULD silently drop the R1 and remain in the current state.
3. If the HIP association state is I1-SENT or I2-SENT, the received Initiator's HIT MUST correspond to the HIT used in the original, I1 and the Responder's HIT MUST correspond to the one used, unless the I1 contained a NULL HIT.
4. The system SHOULD validate the R1 signature before applying further packet processing, according to Section 6.2.15.
5. If the HIP association state is I1-SENT, and multiple valid R1s are present, the system SHOULD select from among the R1s with the largest R1 generation counter.
6. If the HIP association state is I2-SENT, the system MAY reenter state I1-SENT and process the received R1 if it has a larger R1 generation counter than the R1 responded to previously.

7. The R1 packet may have the C bit set -- in this case, the system should anticipate the receipt of HIP CER packets that contain the host identity corresponding to the responder's HIT.
8. The R1 packet may have the A bit set -- in this case, the system MAY choose to refuse it by dropping the R1 and returning to state UNASSOCIATED. The system SHOULD consider dropping the R1 only if it used a NULL HIT in I1. If the A bit is set, the Responder's HIT is anonymous and should not be stored.
9. The system SHOULD attempt to validate the HIT against the received Host Identity.
10. The system MUST store the received R1 generation counter for future reference.
11. The system attempts to solve the cookie puzzle in R1. The system MUST terminate the search after exceeding the remaining lifetime of the puzzle. If the cookie puzzle is not successfully solved, the implementation may either resend I1 within the retry bounds or abandon the HIP exchange.
12. The system computes standard Diffie-Hellman keying material according to the public value and Group ID provided in the DIFFIE_HELLMAN parameter. The Diffie-Hellman keying material Kij is used for key extraction as specified in Section 9. If the received Diffie-Hellman Group ID is not supported, the implementation may either resend I1 within the retry bounds or abandon the HIP exchange.
13. The system selects the HIP transform and ESP transform from the choices presented in the R1 packet and uses the selected values subsequently when generating and using encryption keys, and when sending the I2. If the proposed alternatives are not acceptable to the system, it may either resend I1 within the retry bounds or abandon the HIP exchange.
14. The system prepares and creates an incoming IPsec ESP security association. It may also prepare a security association for outgoing traffic, but since it does not have the correct SPI value yet, it cannot activate it.
15. The system initialized the remaining variables in the associated state, including Update ID counters.
16. The system prepares and sends an I2, as described in Section 7.3.
17. The system SHOULD start a timer whose timeout value should be larger than the worst-case anticipated RTT, and MUST increment a timeout counter associated with the I2. The sender SHOULD retransmit the I2 upon a timeout and restart the timer, up to a maximum of I2_RETRIES_MAX tries.
18. If the system is in state I1-SENT, it shall transition to state I2-SENT. If the system is in any other state, it remains in the current state.

8.6.1 Handling malformed messages

If an implementation receives a malformed R1 message, it MUST silently drop the packet. Sending a NOTIFY or ICMP would not help, as the sender of the R1 typically doesn't have any state. An implementation SHOULD wait for some more time for a possible good R1, after which it MAY try again by sending a new I1 packet.

8.7 Processing incoming I2 packets

Upon receipt of an I2, the system MAY perform initial checks to determine whether the I2 corresponds to a recent R1 that has been sent out, if the Responder keeps such state. For example, the sender could check whether the I2 is from an address or HIT that has recently received an R1 from it. The R1 may have had Opaque data included that was echoed back in the I2. If the I2 is considered to be suspect, it MAY be silently discarded by the system.

Otherwise, the HIP implementation SHOULD process the I2. This includes validation of the cookie puzzle solution, generating the Diffie-Hellman key, decrypting the Initiator's Host Identity, verifying the signature, creating state, and finally sending an R2.

The following steps define the conceptual processing rules for responding to an I2 packet:

1. The system MAY perform checks to verify that the I2 corresponds to a recently sent R1. Such checks are implementation dependent. See Appendix D for a description of an example implementation.
2. The system MUST check that the Responder's HIT corresponds to one of its own HITs.
3. If the system is in the R2-SENT state, it MAY check if the newly received I2 is similar to the one that triggered moving to R2-SENT. If so, it MAY retransmit a previously sent R2, reset the R2-SENT timer, and stay in R2-SENT.
4. If the system is in any other state, it SHOULD check that the echoed R1 generation counter in I2 is within the acceptable range. Implementations MUST accept puzzles from the current generation and MAY accept puzzles from earlier generations. If the newly received I2 is outside the accepted range, the I2 is stale (perhaps replayed) and SHOULD be dropped.
5. The system MUST validate the solution to the cookie puzzle by computing the SHA-1 hash described in Section 7.3.
6. The I2 MUST have a single value in each of the HIP_TRANSFORM and ESP_TRANSFORM parameters, which MUST each match one of the values offered to the Initiator in the R1 packet.
7. The system must derive Diffie-Hellman keying material Kij based on the public value and Group ID in the DIFFIE_HELLMAN

parameter. This key is used to derive the HIP and ESP association keys, as described in Section 9. If the Diffie-Hellman Group ID is unsupported, the I2 packet is silently dropped.

8. The encrypted HOST_ID decrypted by the Initiator encryption key defined in Section 9. If the decrypted data is not an HOST_ID parameter, the I2 packet is silently dropped.
9. The implementation SHOULD also verify that the Initiator's HIT in the I2 corresponds to the Host Identity sent in the I2.
10. The system MUST verify the HMAC according to the procedures in Section 6.2.12.
11. The system MUST verify the HIP_SIGNATURE according to Section 6.2.14 and Section 7.3.
12. If the checks above are valid, then the system proceeds with further I2 processing; otherwise, it discards the I2 and remains in the same state.
13. The I2 packet may have the C bit set -- in this case, the system should anticipate the receipt of HIP CER packets that contain the host identity corresponding to the responder's HIT.
14. The I2 packet may have the A bit set -- in this case, the system MAY choose to refuse it by dropping the I2 and returning to state UNASSOCIATED. If the A bit is set, the Initiator's HIT is anonymous and should not be stored.
15. The SPI field is parsed to obtain the SPI that will be used for the Security Association outbound from the Responder and inbound to the Initiator.
16. The system prepares and creates both incoming and outgoing ESP security associations.
17. The system initialized the remaining variables in the associated state, including Update ID counters.
18. Upon successful processing of an I2 in states UNASSOCIATED, I1-SENT, I2-SENT, and R2-SENT, an R2 is sent and the state machine transitions to state ESTABLISHED.
19. Upon successful processing of an I2 in state ESTABLISHED/ REKEYING, the old Security Association is dropped and a new one is installed, an R2 is sent, and the state machine transitions to R2-SENT, dropping any possibly ongoing rekeying attempt.
20. Upon transitioning to R2-SENT, start a timer. Leave R2-SENT if either the timer expires (allowing for maximal retransmission of I2s), some data has been received on the incoming SA, or an UPDATE packet has been received (or some other packet that indicates that the peer has moved to ESTABLISHED).

8.7.1 Handling malformed messages

If an implementation receives a malformed I2 message, the behaviour SHOULD depend on how much checks the message has already passed. If the puzzle solution in the message has already been checked, the

implementation SHOULD report the error by responding with a NOTIFY packet. Otherwise the implementation MAY respond with an ICMP message as defined in Section 6.3.

8.8 Processing incoming R2 packets

An R2 received in states UNASSOCIATED, I1-SENT, ESTABLISHED, or REKEYING results in the R2 being dropped and the state machine staying in the same state. If an R2 is received in state I2-SENT, it SHOULD be processed.

The following steps define the conceptual processing rules for incoming R2 packet:

1. The system MUST verify that the HITs in use correspond to the HITs that were received in R1.
2. The system MUST verify the HMAC_2 according to the procedures in Section 6.2.13.
3. The system MUST verify the HIP signature according to the procedures in Section 6.2.14.
4. If any of the checks above fail, there is a high probability of an ongoing man-in-the-middle or other security attack. The system SHOULD act accordingly, based on its local policy.
5. If the system is in any other state than I2-SENT, the R2 is silently dropped.
6. The SPI field is parsed to obtain the SPI that will be used for the ESP Security Association inbound to the Responder. The system uses this SPI to create or activate the outgoing ESP security association used to send packets to the peer.
7. Upon successful processing of the R2, the state machine moves to state ESTABLISHED.

8.9 Dropping HIP associations

A HIP implementation is free to drop a HIP association at any time, based on its own policy. If a HIP host decides to drop an HIP association, it deletes the IPsec SAs related to that association and the corresponding HIP state, including the keying material. The implementation MUST also drop the peer's R1 generation counter value, unless a local policy explicitly defines that the value of that particular host is stored. An implementation MUST NOT store R1 generation counters by default, but storing R1 generation counter values, if done, MUST be configured by explicit HITs.

8.10 Initiating rekeying

A system may initiate the rekey procedure at any time. It MUST initiate a rekey if its incoming ESP sequence counter is about to overflow. The system MUST NOT replace its keying material until the

rekeying packet exchange successfully completes. Optionally, depending on policy, a system may include a new Diffie-Hellman key for use in new KEYMAT generation. New KEYMAT generation occurs prior to drawing the new keys.

In the conceptual state machine, a system rekeys when it sends a NES parameter to the peer and receives both an ACK of the relevant UPDATE message and its peer's NES parameter. To leave REKEYING state, both parameters must be received. It may be that the ACK and the NES arrive in different UPDATE messages. This is always true if a system does not initiate rekeying but responds to a rekeying request from the peer, but may also occur if two systems initiate a rekey nearly simultaneously. In such a case, if the system is in state REKEYING, it saves the one parameter and waits for the other before leaving state REKEYING. This implies that the REKEYING state may have conceptual substates.

The following steps define the processing rules for initiating a rekey:

1. The system decides whether to continue to use the existing KEYMAT or to generate new KEYMAT. In the latter case, the system MUST generate a new Diffie-Hellman public key.
2. The system increments its outgoing Update ID by one.
3. The system creates a UPDATE packet, which contains an SEQ parameter (with the current value of Update ID), NES parameter and an optional DIFFIE_HELLMAN parameter. If the UPDATE packet contains the DIFFIE_HELLMAN parameter, the Keymat Index in the NES parameter MUST be zero. If the UPDATE does not contain DIFFIE_HELLMAN, the NES Keymat Index MUST be larger or equal to the index of the next byte to be drawn from the current KEYMAT.
4. The system sends the UPDATE packet and transitions to state REKEYING.
5. The system SHOULD start a timer whose timeout value should be larger than the worst-case anticipated RTT, and MUST increment a timeout counter associated with UPDATE. The sender SHOULD retransmit the UPDATE upon a timeout and restart the timer, up to a maximum of UPDATE_RETRIES_MAX tries.
6. The system MUST NOT delete its existing SAs, but continue using them if its policy still allows. The UPDATE procedure SHOULD be initiated early enough to make sure that the SA replay counters do not overflow.
7. In case a protocol error occurs and the peer system acknowledges the UPDATE but does not itself send a NES, the system may hang in state REKEYING. To guard against this, a system MAY re-initiate the rekeying procedure after some time waiting for the peer to respond, or it MAY decide to abort the HIP association after waiting for an implementation-dependent time. The system MUST NOT hang in state REKEYING for an indefinite time.

To simplify the state machine, a host MUST NOT generate new UPDATEs (with higher Update IDs) while in state REKEYING, unless it is restarting the rekeying process.

8.11 Processing UPDATE packets

When a system receives an UPDATE packet, its processing depends on the state of the HIP association and the presence of and values of the SEQ and ACK parameters. An UPDATE MUST be processed if the following conditions hold (note: UPDATEs may also be processed when additional conditions hold, as specified in other drafts):

1. If there is no corresponding HIP association, the implementation MAY reply with an ICMP Parameter Problem, as specified in Section 6.3.5.
2. The state of the HIP association is ESTABLISHED or REKEYING, and both the SEQ and NES parameters are present in the UPDATE. This is the case for which the peer host is in the process of rekeying.
3. The state of the HIP association is REKEYING and an ACK (of outstanding Update ID) is in the UPDATE. This case usually corresponds to the peer completing the rekeying process first.

If the above conditions hold, the following steps define the conceptual processing rules for handling a received UPDATE packet:

1. If the SEQ parameter is present, and the Update ID in the received SEQ is smaller than the stored Update ID for the host, the packet MUST BE dropped.
2. If the SEQ parameter is present, and the Update ID in the received SEQ is equal to the stored Update ID for the host, the packet is treated as a retransmission. However, the HMAC verification (next step) MUST NOT be skipped. (A byte-by-byte comparison of the received and a store packet would be OK, though.) It is recommended that a host cache the last packet that was acked to avoid the cost of generating a new ACK packet to respond to a replayed UPDATE. Systems MUST again acknowledge such apparent UPDATE message retransmissions but SHOULD also consider rate-limiting such retransmission responses to guard against replay attacks.
3. The system MUST verify the HMAC in the UPDATE packet. If the verification fails, the packet MUST be dropped.
4. If the received UPDATE contains a DIFFIE_HELLMAN parameter, the received Keymat Index MUST be zero. If this test fails, the packet SHOULD be dropped and the system SHOULD log an error message.
5. The system MAY verify the SIGNATURE in the UPDATE packet. If the verification fails, the packet SHOULD be dropped and an error message logged.

6. If a new SEQ parameter is being processed, the system MUST record the Update ID in the received SEQ parameter, for replay protection.
7. If the system is in state ESTABLISHED, and the UPDATE has the NES and SEQ parameters, the packet processing continues as specified in Section 8.11.1.
8. If the system is in state REKEYING, the packet processing continues as specified in Section 8.11.2.

8.11.1 Processing an UPDATE packet in state ESTABLISHED

The following steps define the conceptual processing rules responding handling a received initial UPDATE packet:

1. The system consults its policy to see if it needs to generate a new Diffie-Hellman key, and generates a new key if needed. The system records any newly generated or received Diffie-Hellman keys, for use in KEYMAT generation upon leaving the REKEYING state.
2. If the system generated new Diffie-Hellman key in the previous step, or it received a DIFFIE_HELLMAN parameter, it sets NES Keymat Index to zero. Otherwise, the NES Keymat Index MUST be larger or equal to the index of the next byte to be drawn from the current KEYMAT. In this case, it is RECOMMENDED that the host use the Keymat Index requested by the peer in the received NES.
3. The system increments its outgoing Update ID by one.
4. The system creates a UPDATE packet, which contains an SEQ parameter (with the current value of Update ID), NES parameter and the optional DIFFIE_HELLMAN parameter. The UPDATE packet also includes the ACK of the Update ID found in the received UPDATE SEQ parameter.
5. The system sends the UPDATE packet and transitions to state REKEYING. The system stores any received NES and DIFFIE_HELLMAN parameters. At this point, it only needs to receive an ACK of its current Update ID to finish rekeying.

8.11.2 Processing an UPDATE packet in state REKEYING

The following steps define the conceptual processing rules responding handling a received reply UPDATE packet:

1. If the packet contains a SEQ and NES parameters, then the system sends a new UPDATE packet with an ACK of the peer's Update ID as received in the SEQ parameter. Additionally, if the UPDATE packet contained an ACK of the outstanding Update ID, or if the ACK of the UPDATE packet that contained the NES has already been received, the system stores the received NES and (optional) DIFFIE_HELLMAN parameters and finishes the rekeying procedure as described in Section 8.11.3. If the ACK of the outstanding Update

ID has not been received, stay in state REKEYING after storing the received NES and (optional) DIFFIE_HELLMAN.

2. If the packet contains an ACK parameter that ACKs the outstanding Update ID, and the system has previously received a NES from the peer, the system finishes the rekeying procedure as described in Section 8.11.3. If the system is still waiting for the peer's NES parameter (to arrive in subsequent UPDATE message), the system stays in state REKEYING.

8.11.3 Leaving REKEYING state

A system leaves REKEYING state when it has received both a NES from its peer and the ACK of the Update ID that was sent in its own NES to the peer. The following steps are taken:

1. If either the received UPDATE contains a new Diffie-Hellman key, the system has a new Diffie-Hellman key from initiating rekey, or both, the system generates new KEYMAT. If there is only one new Diffie-Hellman key, the old key is used as the other key.
2. If the system generated new KEYMAT in the previous step, it sets Keymat Index to zero, independent on whether the received UPDATE included a Diffie-Hellman key or not. If the system did not generate new KEYMAT, it uses the lowest Keymat Index of the two NES parameters.
3. The system draws keys for new incoming and outgoing ESP SAs, starting from the Keymat Index, and prepares new incoming and outgoing ESP SAs. The SPI for the outgoing SA is the new SPI value from the UPDATE. The SPI for the incoming SA was generated when NES was sent. The order of the keys retrieved from the KEYMAT during rekeying process is similar to that described in Section 9. Note, that only IPsec ESP keys are retrieved during rekeying process, not the HIP keys.
4. The system cancels any timers protecting the UPDATE and transitions to ESTABLISHED.
5. The system starts to send to the new outgoing SA and prepares to start receiving data on the new incoming SA.

8.12 Processing CER packets

Processing CER packets is OPTIONAL, and currently undefined.

8.13 Processing NOTIFY packets

Processing NOTIFY packets is OPTIONAL. If processed, any errors noted by the NOTIFY parameter SHOULD be taken into account by the HIP state machine (e.g., by terminating a HIP handshake), and the error SHOULD be logged.

8.14 Processing CLOSE packets

When the host receives a CLOSE message it responds with a CLOSE_ACK message and moves to CLOSED state. (The authenticity of the CLOSE message is verified using both HMAC and SIGNATURE). This processing applies whether or not the HIP association state is CLOSING in order to handle CLOSE messages from both ends crossing in flight.

The HIP association is not discarded before the host moves from the UNASSOCIATED state.

Once the closing process has started, any need to send data packets will trigger creating and establishing of a new HIP association, starting with sending an I1.

If there is no corresponding HIP association, the implementation MAY reply to a CLOSE with an ICMP Parameter Problem, as specified in Section 6.3.5.

8.15 Processing CLOSE_ACK packets

When a host receives a CLOSE_ACK message it verifies that it is in CLOSING or CLOSED state and that the CLOSE_ACK was in response to the CLOSE (using the included ECHO_REPLY in response to the sent ECHO_REQUEST).

The CLOSE_ACK uses HMAC and SIGNATURE for verification. The state is discarded when the state changes to UNASSOCIATED and, after that, NOTIFY is sent as a response to a CLOSE message.

9. HIP KEYMAT

HIP keying material is derived from the Diffie-Hellman Kij produced during the base HIP exchange. The Initiator has Kij during the creation of the I2 packet, and the Responder has Kij once it receives the I2 packet. This is why I2 can already contain encrypted information.

The KEYMAT is derived by feeding Kij and the HITs into the following operation; the | operation denotes concatenation.

```
KEYMAT = K1 | K2 | K3 | ...
      where

K1   = SHA-1( Kij | sort(HIT-I | HIT-R) | 0x01 )
K2   = SHA-1( Kij | K1 | 0x02 )
K3   = SHA-1( Kij | K2 | 0x03 )
...
K255 = SHA-1( Kij | K254 | 0xff )
K256 = SHA-1( Kij | K255 | 0x00 )
etc.
```

Sort(HIT-I | HIT-R) is defined as the network byte order concatenation of the two HITs, with the smaller HIT preceding the larger HIT, resulting from the numeric comparison of the two HITs interpreted as positive (unsigned) 128-bit integers in network byte order.

The initial keys are drawn sequentially in the order that is determined by the numeric comparison of the two HITs, with comparison method described in the previous paragraph. HOST_g denotes the host with the greater HIT value, and HOST_l the host with the lower HIT value.

The drawing order for initial keys:

- HIP-gl encryption key for HOST_g's outgoing HIP packets
- HIP-gl integrity (HMAC) key for HOST_g's outgoing HIP packets
- HIP-lg encryption key (currently unused) for HOST_l's outgoing HIP packets
- HIP-lg integrity (HMAC) key for HOST_l's outgoing HIP packets
- SA-gl ESP encryption key for HOST_g's outgoing traffic
- SA-gl ESP authentication key for HOST_g's outgoing traffic
- SA-lg ESP encryption key for HOST_l's outgoing traffic
- SA-lg ESP authentication key for HOST_l's outgoing traffic

The number of bits drawn for a given algorithm is the "natural" size of the keys. For the mandatory algorithms, the following sizes apply:

AES 128 bits
SHA-1 160 bits
NULL 0 bits

The four HIP keys are only drawn from KEYMAT during a HIP I1->R2 exchange. Subsequent rekeys using UPDATE will only draw the four ESP keys from KEYMAT. Section 8.11 describes the rules for reusing or regenerating KEYMAT based on the UPDATE exchange.

10. HIP Fragmentation Support

A HIP implementation must support IP fragmentation / reassembly. Fragment reassembly MUST be implemented in both IPv4 and IPv6, but fragment generation MUST be implemented only in IPv4 (IPv4 stacks and networks will usually do this by default) and SHOULD be implemented in IPv6. In the IPv6 world, the minimum MTU is larger, 1280 bytes, than in the IPv4 world. The larger MTU size is usually sufficient for most HIP packets, and therefore fragment generation may not be needed. If a host expects to send HIP packets that are larger than the minimum IPv6 MTU, it MUST implement fragment generation even for IPv6.

In the IPv4 world, HIP packets may encounter low MTUs along their routed path. Since HIP does not provide a mechanism to use multiple IP datagrams for a single HIP packet, support of path MTU discovery does not bring any value to HIP in the IPv4 world. HIP aware NAT systems MUST perform any IPv4 reassembly/fragmentation.

All HIP implementations MUST employ a reassembly algorithm that is sufficiently resistant against DoS attacks.

11. ESP with HIP

HIP is designed to be used in end-to-end fashion. The IPsec mode used with HIP is the BEET mode (A Bound End-to-End mode for ESP) [27]. The BEET mode provides some features from both IPsec tunnel and transport modes. The HIP uses HITs and LSIs as the "inner" addresses and IP addresses as "outer" addresses like IP addresses are used in the tunnel mode. Instead of tunneling packets between hosts, a conversion between inner and outer addresses is made at end-hosts and the inner address is never sent in the wire after the initial HIP negotiation. BEET provides IPsec transport mode syntax (no inner headers) with limited tunnel mode semantics (fixed logical inner addresses - the HITs - and changeable outer IP addresses).

Since HIP does not negotiate any lifetimes, all lifetimes are local policy. The only lifetimes a HIP implementation MUST support are sequence number rollover (for replay protection), and SA timeout. An SA times out if no packets are received using that SA. The default timeout value is 15 minutes. Implementations MAY support lifetimes for the various ESP transforms.

11.1 ESP Security Associations

Each HIP association is linked with two ESP SAs, one incoming and one outgoing. The Initiator's incoming SA corresponds with the Responder's outgoing one. The initiator defines the SPI for this association, as defined in Section 3.3. This SA is called SA-RI, and the corresponding SPI is called SPI-RI. Respectively, the Responder's incoming SA corresponds with the Initiator's outgoing SA and is called SA-IR, with the SPI-IR.

The Initiator creates SA-RI as a part of R1 processing, before sending out the I2, as explained in Section 8.6. The keys are derived from KEYMAT, as defined in Section 9. The Responder creates SA-RI as a part of I2 processing, see Section 8.7.

The Responder creates SA-IR as a part of I2 processing, before sending out R2, see Step 17 in Section 8.7. The Initiator creates SA-IR when processing R2, see Step 7 in Section 8.8.

11.2 Updating ESP SAs during rekeying

After the initial 4-way handshake and SA establishment, both hosts are in state ESTABLISHED. There are no longer Initiator and Responder roles and the association is symmetric. In this subsection, the initiating party of the rekey procedure is denoted with I' and the peer with R'.

The I' initiates the rekeying process when needed (see Section 8.10). It creates an UPDATE packet with required information and sends it to the peer node. The old SAs are still in use.

The R', after receiving and processing the UPDATE (see Section 8.11), generates new SAs: SA-I'R' and SA-R'I'. It does not take the new outgoing SA into use, but uses still the old one, so there exists two SA pairs towards the same peer host. For the new outgoing SA, the SPI-R'I' value is picked from the received UPDATE packet. The R' generates the new SPI value for the incoming SA, SPI-I'R', and includes it in the response UPDATE packet.

When the I' receives a response UPDATE from the R', it generates new SAs, as described in Section 8.11: SA-I'R' and SA-R'I'. It starts using the new outgoing SA immediately.

The R' starts using the new outgoing SA when it receives traffic from the new incoming SA. After this, the R' can remove old SAs. Similarly, when the I' receives traffic from the new incoming SA, it can safely remove old SAs.

11.3 Security Association Management

An SA pair is indexed by the 2 SPIs and 2 HITs (both HITs since a system can have more than one HIT). An inactivity timer is recommended for all SAs. If the state dictates the deletion of an SA, a timer is set to allow for any late arriving packets.

11.4 Security Parameter Index (SPI)

The SPIs in ESP provide a simple compression of the HIP data from all packets after the HIP exchange. This does require a per HIT- pair Security Association (and SPI), and a decrease of policy granularity over other Key Management Protocols like IKE.

When a host rekeys, it gets a new SPI from its partner.

11.5 Supported Transforms

All HIP implementations MUST support AES [10] and HMAC-SHA-1-96 [6]. If the Initiator does not support any of the transforms offered by the Responder in the R1 HIP packet, it MUST use AES and HMAC-SHA-1-96 and state so in the I2 HIP packet.

In addition to AES, all implementations MUST implement the ESP NULL encryption and authentication algorithms. These algorithms are provided mainly for debugging purposes, and SHOULD NOT be used in production environments. The default configuration in

implementations MUST be to reject NULL encryption or authentication.

11.6 Sequence Number

The Sequence Number field is MANDATORY in ESP. Anti-replay protection MUST be used in an ESP SA established with HIP.

This means that each host MUST rekey before its sequence number reaches 2^32, or if extended sequence numbers are used, 2^64. Note that in HIP rekeying, unlike IKE rekeying, only one Diffie-Hellman key can be changed, that of the rekeying host. However, if one host rekeys, the other host SHOULD rekey as well.

In some instances, a 32-bit sequence number is inadequate. In the ESP_TRANSFORM parameter, a peer MAY require that a 64 bit sequence number be used. In this case the higher 32 bits are NOT included in the ESP header, but are simply kept local to both peers. 64 bit sequence numbers must only be used for ciphers that will not be open to cryptanalysis as a result. AES is one such cipher.

12. HIP Policies

There are a number of variables that will influence the HIP exchanges that each host must support. All HIP implementations MUST support more than one simultaneous HIs, at least one of which SHOULD be reserved for anonymous usage. Although anonymous HIs will be rarely used as responder HIs, they will be common for Initiators. Support for more than two HIs is RECOMMENDED.

Many Initiators would want to use a different HI for different Responders. The implementations SHOULD provide for an ACL of initiator HIT to responder HIT. This ACL SHOULD also include preferred transform and local lifetimes. For HITs with HAAs, wildcarding SHOULD be supported. Thus if a Community of Interest, like Banking, gets an RAA, a single ACL could be used. A global wildcard would represent the general policy to be used. Policy selection would be from most specific to most general.

The value of K used in the HIP R1 packet can also vary by policy. K should never be greater than 20, but for trusted partners it could be as low as 0.

Responders would need a similar ACL, representing which hosts they accept HIP exchanges, and the preferred transform and local lifetimes. Wildcarding SHOULD be supported for this ACL also.

13. Security Considerations

HIP is designed to provide secure authentication of hosts and to provide a fast key exchange for IPsec ESP. HIP also attempts to limit the exposure of the host to various denial-of-service and man-in-the-middle attacks. In so doing, HIP itself is subject to its own DoS and MitM attacks that potentially could be more damaging to a host's ability to conduct business as usual.

HIP enabled ESP is IP address independent. This might seem to make it easier for an attacker, but ESP with replay protection is already as well protected as possible, and the removal of the IP address as a check should not increase the exposure of ESP to DoS attacks. Furthermore, this is in line with the forthcoming revision of ESP.

Denial-of-service attacks take advantage of the cost of start of state for a protocol on the Responder compared to the 'cheapness' on the Initiator. HIP makes no attempt to increase the cost of the start of state on the Initiator, but makes an effort to reduce the cost to the Responder. This is done by having the Responder start the 3-way exchange instead of the Initiator, making the HIP protocol 4 packets long. In doing this, packet 2 becomes a 'stock' packet that the Responder MAY use many times. The duration of use is a paranoia versus throughput concern. Using the same Diffie- Hellman values and random puzzle I has some risk. This risk needs to be balanced against a potential storm of HIP I1 packets.

This shifting of the start of state cost to the Initiator in creating the I2 HIP packet, presents another DoS attack. The attacker spoofs the I1 HIP packet and the Responder sends out the R1 HIP packet. This could conceivably tie up the 'initiator' with evaluating the R1 HIP packet, and creating the I2 HIP packet. The defense against this attack is to simply ignore any R1 packet where a corresponding I1 or ESP data was not sent.

A second form of DoS attack arrives in the I2 HIP packet. Once the attacking Initiator has solved the cookie challenge, it can send packets with spoofed IP source addresses with either invalid encrypted HIP payload component or a bad HIP signature. This would take resources in the Responder's part to reach the point to discover that the I2 packet cannot be completely processed. The defense against this attack is after N bad I2 packets, the Responder would discard any I2s that contain the given Initiator HIT. Thus will shut down the attack. The attacker would have to request another R1 and use that to launch a new attack. The Responder could up the value of K while under attack. On the downside, valid I2s might get dropped too.

A third form of DoS attack is emulating the restart of state after a reboot of one of the partners. A host restarting would send an I1 to a peer, which would respond with an R1 even if it were in state ESTABLISHED. If the I1 were spoofed, the resulting R1 would be received unexpectedly by the spoofed host and would be dropped, as in the first case above.

A fourth form of DoS attack is emulating the end of state. HIP relies on timers plus a CLOSE/CLOSE_ACK handshake to explicitly signals the end of a state. Because both CLOSE and CLOSE_ACK messages contain an HMAC, an outsider cannot close a connection. The presence of an additional SIGNATURE allows middle-boxes to inspect these messages and discard the associated state (for e.g., firewalling, SPI-based NATing, etc.). However, the optional behavior of replying to CLOSE with an ICMP Parameter Problem packet (as described in Section 6.3.5), might allow an IP spoofer sending CLOSE messages to launch reflection attacks.

A fifth form of DoS attack is replaying R1s to cause the initiator to solve stale puzzles and become out of synchronization with the responder. The R1 generation counter is a monotonically increasing counter designed to protect against this attack, as described in section Section 4.1.3.

Man-in-the-middle attacks are difficult to defend against, without third-party authentication. A skillful MitM could easily handle all parts of HIP; but HIP indirectly provides the following protection from a MitM attack. If the Responder's HI is retrieved from a signed DNS zone, a certificate, or through some other secure means, the Initiator can use this to validate the R1 HIP packet.

Likewise, if the Initiator's HI is in a secure DNS zone, a trusted certificate, or otherwise securely available, the Responder can retrieve it after it gets the I2 HIP packet and validate that. However, since an Initiator may choose to use an anonymous HI, it knowingly risks a MitM attack. The Responder may choose not to accept a HIP exchange with an anonymous Initiator.

If an initiator wants to use opportunistic mode, it is vulnerable to man-in-the-middle attacks. Furthermore, the available HI types are limited to the MUST implement algorithms, as per Section 3. Hence, if a future specification deprecates the current MUST implement algorithm(s) and replaces it (them) with some new one(s), backward compatibility cannot be preserved.

Since not all hosts will ever support HIP, ICMP 'Destination Protocol Unreachable' are to be expected and present a DoS attack. Against an Initiator, the attack would look like the Responder does not support

HIP, but shortly after receiving the ICMP message, the Initiator would receive a valid R1 HIP packet. Thus to protect from this attack, an Initiator should not react to an ICMP message until a reasonable delta time to get the real Responder's R1 HIP packet. A similar attack against the Responder is more involved. First an ICMP message is expected if the I1 was a DoS attack and the real owner of the spoofed IP address does not support HIP. The Responder SHOULD NOT act on this ICMP message to remove the minimal state from the R1 HIP packet (if it has one), but wait for either a valid I2 HIP packet or the natural timeout of the R1 HIP packet. This is to allow for a sophisticated attacker that is trying to break up the HIP exchange. Likewise, the Initiator should ignore any ICMP message while waiting for an R2 HIP packet, deleting state only after a natural timeout.

14. IANA Considerations

IANA has assigned IP Protocol number TBD to HIP.

15. Acknowledgments

The drive to create HIP came to being after attending the MALLOC meeting at IETF 43. Baiju Patel and Hilarie Orman really gave the original author, Bob Moskowitz, the assist to get HIP beyond 5 paragraphs of ideas. It has matured considerably since the early drafts thanks to extensive input from IETFers. Most importantly, its design goals are articulated and are different from other efforts in this direction. Particular mention goes to the members of the NameSpace Research Group of the IRTF. Noel Chiappa provided the framework for LSIs and Keith Moore the impetus to provide resolvability. Steve Deering provided encouragement to keep working, as a solid proposal can act as a proof of ideas for a research group.

Many others contributed; extensive security tips were provided by Steve Bellovin. Rob Austein kept the DNS parts on track. Paul Kocher taught Bob Moskowitz how to make the cookie exchange expensive for the Initiator to respond, but easy for the Responder to validate. Bill Sommerfeld supplied the Birthday concept to simplify reboot management. Rodney Thayer and Hugh Daniels provide extensive feedback. In the early times of this draft, John Gilmore kept Bob Moskowitz challenged to provide something of value.

During the later stages of this document, when the editing baton was transfered to Pekka Nikander, the input from the early implementors were invaluable. Without having actual implementations, this document would not be on the level it is now.

In the usual IETF fashion, a large number of people have contributed to the actual text or ideas. The list of these people include Jeff Ahrenholz, Francis Dupont, Derek Fawcus, George Gross, Andrew McGregor, Julien Laganier, Miika Komu, Mika Kousa, Jan Melen, Henrik Petander, Michael Richardson, Tim Shepard, Jorma Wall, and Jukka Ylitalo. Our apologies to anyone who's name is missing.

16. References

16.1 Normative references

[1] Postel, J., "User Datagram Protocol", STD 6, RFC 768, August 1980.

[2] Postel, J., "Internet Control Message Protocol", STD 5, RFC 792, September 1981.

[3] Mockapetris, P., "Domain names - implementation and specification", STD 13, RFC 1035, November 1987.

[4] Conta, A. and S. Deering, "Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6)", RFC 1885, December 1995.

[5] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[6] Madson, C. and R. Glenn, "The Use of HMAC-SHA-1-96 within ESP and AH", RFC 2404, November 1998.

[7] Maughan, D., Schneider, M. and M. Schertler, "Internet Security Association and Key Management Protocol (ISAKMP)", RFC 2408, November 1998.

[8] Harkins, D. and D. Carrel, "The Internet Key Exchange (IKE)", RFC 2409, November 1998.

[9] Orman, H., "The OAKLEY Key Determination Protocol", RFC 2412, November 1998.

[10] Pereira, R. and R. Adams, "The ESP CBC-Mode Cipher Algorithms", RFC 2451, November 1998.

[11] Deering, S. and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification", RFC 2460, December 1998.

[12] Eastlake, D., "Domain Name System Security Extensions", RFC 2535, March 1999.

[13] Eastlake, D., "DSA KEYs and SIGs in the Domain Name System (DNS)", RFC 2536, March 1999.

[14] Eastlake, D., "RSA/SHA-1 SIGs and RSA KEYs in the Domain Name System (DNS)", RFC 3110, May 2001.

[15] Housley, R., Polk, W., Ford, W. and D. Solo, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 3280, April 2002.

[16] Draves, R., "Default Address Selection for Internet Protocol version 6 (IPv6)", RFC 3484, February 2003.

[17] Hinden, R. and S. Deering, "Internet Protocol Version 6 (IPv6) Addressing Architecture", RFC 3513, April 2003.

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

[19] Kent, S., "IP Encapsulating Security Payload (ESP)", draft-ietf-ipsec-esp-v3-05 (work in progress), April 2003.

[20] Kaufman, C., "Internet Key Exchange (IKEv2) Protocol", draft-ietf-ipsec-ikev2-07 (work in progress), April 2003.

[21] Moskowitz, R., "Host Identity Protocol Architecture", draft-moskowitz-hip-arch-03 (work in progress), May 2003.

[22] NIST, "FIPS PUB 180-1: Secure Hash Standard", April 1995.

16.2 Informative references

[23] Bellovin, S. and W. Aiello, "Just Fast Keying (JFK)", draft-ietf-ipsec-jfk-04 (work in progress), July 2002.

[24] Moskowitz, R. and P. Nikander, "Using Domain Name System (DNS) with Host Identity Protocol (HIP)", draft-nikander-hip-dns-00 (to be issued) (work in progress), June 2003.

[25] Nikander, P., "SPI assisted NAT traversal (SPINAT) with Host Identity Protocol (HIP)", draft-nikander-hip-nat-00 (to be issued) (work in progress), June 2003.

[26] Crosby, SA. and DS. Wallach, "Denial of Service via Algorithmic Complexity Attacks", in Proceedings of Usenix Security Symposium 2003, Washington, DC., August 2003.

[27] Nikander, P., "A Bound End-to-End Tunnel (BEET) mode for ESP", draft-nikander-esp-beet-mode-00 (expired) (work in progress), Oct 2003.

Authors' Addresses

Robert Moskowitz
ICSAlabs, a Division of TruSecure Corporation
1000 Bent Creek Blvd, Suite 200
Mechanicsburg, PA
USA

EMail: rgm@icsalabs.com

Pekka Nikander
Ericsson Research NomadicLab

JORVAS FIN-02420
FINLAND

Phone: +358 9 299 1
EMail: pekka.nikander@nomadiclab.com

Petri Jokela
Ericsson Research NomadicLab

JORVAS FIN-02420
FINLAND

Phone: +358 9 299 1
EMail: petri.jokela@nomadiclab.com

Thomas R. Henderson
The Boeing Company
P.O. Box 3707
Seattle, WA
USA

EMail: thomas.r.henderson@boeing.com

Appendix A. API issues

The following text is informational and may be expanded upon or revised in a separate Informational document.

HIP may be used to support application data transfers in one of three ways:

- the application may be HIP-aware and may explicitly use a HIP-based API and/or resolver library;
- the application may not be HIP-aware but may be provided with HITs or LSIs in place of IP addresses as part of the address resolution process; and
- the application may or may not be HIP-aware and may present IP addresses to the system, but the system may decide to opportunistically invoke HIP or use a pre-existing HIP-based SA on its behalf.

The first case is the most straightforward. The HIP-based API is outside the scope of this document.

The second case is one way to provide HIP support to non-HIP-aware applications. HITs may be stored in the DNS or some other infrastructure, and the resolver library may choose to supply a querying application with a HIT or LSI in place of an IP address. Note that if the application truly needs IP addresses for a domain name for some reason (e.g., a diagnostic application, or for use in a referral scenario to a non-HIP-based host), blindly providing HITs or LSIs in place of actual IP addresses may cause some applications to break.

In both of the first two cases, the means whereby a system can resolve an LSI or HIT to an IP address, when such a mapping is not locally cached in the system, is outside the scope of this document.

In the third case, the system is explicitly invoking HIP to a particular destination IP address on the basis of a local policy decision. This approach resembles the way that opportunistic IPsec works. Effectively, this approach is implicitly associating IP addresses with host identities, and is prone to certain failures or ambiguity in an environment where IP addresses are dynamic (e.g., an application connects to an IP address, the peer host moves at some later time, then another host acquires the old IP address, and the system again receives a request to connect to that IP address, in which case it is ambiguous whether the application wants to connect to the host previously at that IP address or the new host at that address).

If HIP is used to support an application, the application data stream

may contain either IP addresses or LSIs or HITs in place of the IP addresses.

Historically, the first two bits of a HIT were used to differentiate between Type 1, Type 2, and IPv6 address formats. This was changed in October 2004, when the Working Group decided that all (currently defined) HITs are 128-bit long. Hence, a Type 1 HIT consists of 128 bits of the SHA-1 hash of the public key, and a Type 2 HIT consists of a 64-bits long HAA field, followed by a 64-bits of the SHA-1 hash. [The format of the HAA field is left undefined in this document.]

In this document, we additionally define an internal IPv6-compatible LSI representation format, to be used within the legacy IPv6-compatible API (e.g., socket over AF_INET6). The format of these IPv6-compatible LSIs is designed to avoid the most commonly occurring IPv6 addresses in RFC3596 [9]. An IPv6-compatible LSI representation of a HIT can be easily computed by replacing the first TBDth bits of the HIT by the TBD bits long prefix "0xTBD". Accordingly, this specification also RECOMMENDS that conforming implementations ignore the TBD prefix bits when comparing HITs for equality; see Section 3.1.

Appendix B. Probabilities of HIT collisions

The birthday paradox sets a bound for the expectation of collisions. It is based on the square root of the number of values. A 64-bit hash, then, would put the chances of a collision at 50-50 with 2^32 hosts (4 billion). A 1% chance of collision would occur in a population of 640M and a .001% collision chance in a 20M population. A 128 bit hash will have the same .001% collision chance in a 9x10^16 population.

Appendix C. Probabilities in the cookie calculation

A question: Is it guaranteed that the Initiator is able to solve the puzzle in this way when the K value is large?

Answer: No, it is not guaranteed. But it is not guaranteed even in the old mechanism, since the Initiator may start far away from J and arrive to J after far too many steps. If we wanted to make sure that the Initiator finds a value, we would need to give some hint of a suitable J, and I don't think we want to do that.

In general, if we model the hash function with a random function, the probability that one iteration gives are result with K zero bits is 2^-K. Thus, the probability that one iteration does *not* give K zero bits is (1 - 2^-K). Consequently, the probability that 2^K iterations does not give K zero bits is (1 - 2^-K)^(2^K).

Since my calculus starts to be rusty, I made a small experiment and found out that

```
lim     (1 - 2^-k)^(2^k)     = 0.36788
k->inf

lim     (1 - 2^-k)^(2^(k+1)) = 0.13534
k->inf

lim     (1 - 2^-k)^(2^(k+2)) = 0.01832
k->inf

lim     (1 - 2^-k)^(2^(k+3)) = 0.000335
k->inf
```

Thus, if hash functions were random functions, we would need about 2^(K+3) iterations to make sure that the probability of a failure is less than 1% (actually less than 0.04%). Now, since my perhaps flawed understanding of hash functions is that they are "flatter" than random functions, 2^(K+3) is probably an overkill. OTOH, the currently suggested 2^K is clearly too little.

Appendix D. Using responder cookies

As mentioned in Section 4.1.1, the Responder may delay state creation and still reject most spoofed I2s by using a number of pre-calculated R1s and a local selection function. This appendix defines one possible implementation in detail. The purpose of this appendix is to give the implementors an idea on how to implement the mechanism. The method described in this appendix SHOULD NOT be used in any real implementation. If the implementation is based on this appendix, it SHOULD contain some local modification that makes an attacker's task harder.

The basic idea is to create a cheap, varying local mapping function f:

```
f( IP-I, IP-R, HIT-I, HIT-R ) -> cookie-index
```

That is, given the Initiator's and Responder's IP addresses and HITs, the function returns an index to a cookie. When processing an I1, the cookie is embedded in an pre-computed R1, and the Responder simply sends that particular R1 to the Initiator. When processing an I2, the cookie may still be embedded in the R1, or the R1 may be deprecated (and replaced with a new one), but the cookie is still there. If the received cookie does not match with the R1 or saved cookie, the I2 is simply dropped. That prevents the Initiator from generating spoofed I2s with a probability that depends on the number of pre-computed R1s.

As a concrete example, let us assume that the Responder has an array of R1s. Each slot in the array contains a timestamp, an R1, and an old cookie that was sent in the previous R1 that occupied that particular slot. The Responder replaces one R1 in the array every few minutes, thereby replacing all the R1s gradually.

To create a varying mapping function, the Responder generates a random number every few minutes. The octets in the IP addresses and HITs are XORed together, and finally the result is XORed with the random number. Using pseudo-code, the function looks like the following.

```
Pre-computation:
    r1 := random number

Index computation:
    index := r1    XOR hit_r[0] XOR hit_r[1] XOR ... XOR hit_r[15]
    index := index XOR hit_i[0] XOR hit_i[1] XOR ... XOR hit_i[15]
    index := index XOR  ip_r[0] XOR  ip_r[1] XOR ... XOR  ip_r[15]
    index := index XOR  ip_i[0] XOR  ip_i[1] XOR ... XOR  ip_i[15]
```

The index gives the slot used in the array.

It is possible that an Initiator receives an I1, and while it is computing I2, the Responder deprecates an R1 and/or chooses a new random number for the mapping function. Therefore the Responder must remember the cookies used in deprecated R1s and the previous random number.

To check an received I2, the Responder can use a simple algorithm, expressed in pseudo-code as follows.

```
  If I2.hit_r does not match my_hits, drop the packet.

  index := compute_index(current_random_number, I2)
  If current_cookie[index]  == I2.cookie, go to cookie check.
  If previous_cookie[index] == I2.cookie, go to cookie check.

  index := compute_index(previous_random_number, I2)
  If current_cookie[index]  == I2.cookie, go to cookie check.
  If previous_cookie[index] == I2.cookie, go to cookie check.

  Drop packet.

cookie_check:
  V := Ltrunc( SHA-1( I2.I, I2.hit_i, I2.hit_r, I2.J ), K )
  if V != 0, drop the packet.
```

Whenever the Responder receives an I2 that fails on the index check, it can simply drop the packet on the floor and forget about it. New I2s with the same or other spoofed parameters will get dropped with a reasonable probability and minimal effort.

If a Responder receives an I2 that passes the index check but fails on the puzzle check, it should create a state indicating this. After two or three failures the Responder should cease checking the puzzle but drop the packets directly. This saves the Responder from the SHA-1 calculations. Such block should not last long, however, or there would be a danger that a legitimate Initiator could be blocked from getting connections.

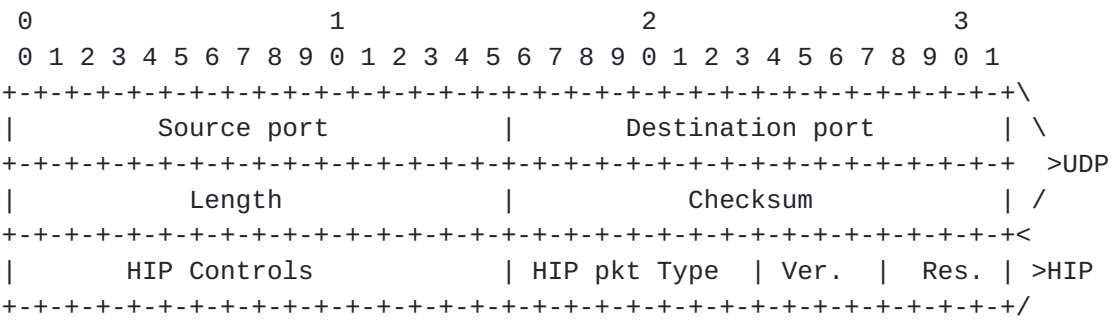
A key for the success of the defined scheme is that the mapping function must be considerably cheaper than computing SHA-1. It also must detect any changes in the IP addresses, and preferably most changes in the HITs. Checking the HITs is not that essential, though, since HITs are included in the cookie computation, too.

The effectivity of the method can be varied by varying the size of the array containing pre-computed R1s. If the array is large, the

probability that an I2 with a spoofed IP address or HIT happens to map to the same slot is fairly slow. However, a large array means that each R1 has a fairly long life time, thereby allowing an attacker to utilize one solved puzzle for a longer time.

Appendix E. Running HIP over IPv4 UDP

In the IPv4 world, with the deployed NAT devices, it may make sense to run HIP over UDP. When running HIP over UDP, the following packet structure is used. The structure is followed by the HITs, as usual. Both the Source and Destination port MUST be 272.



It is currently undefined how the actual data transfer, using ESP, is handled. Plain ESP may not go through all NAT devices.

It is currently FORBIDDEN to use this packet format with IPv6.

Appendix F. Example checksums for HIP packets

The HIP checksum for HIP packets is specified in Section 6.1.2. Checksums for TCP and UDP packets running over HIP-enabled security associations are specified in Section 3.5. The examples below use IP addresses of 192.168.0.1 and 192.168.0.2 (and their respective IPv4-compatible IPv6 formats), and type 1 HITs with the first two bits "01" followed by 124 zeroes followed by a decimal 1 or 2, respectively.

F.1 IPv6 HIP example (I1)

Source Address:	::c0a8:0001	
Destination Address:	::c0a8:0002	
Upper-Layer Packet Length:	40	0x28
Next Header:	99	0x63
Payload Protocol:	59	0x3b
Header Length:	4	0x04
Packet Type:	1	0x01
Version:	1	0x1
Reserved:	0	0x0
Control:	0	0x0000
Checksum:	49672	0xc208
Sender's HIT:	4000::0001	
Receiver's HIT:	4000::0002	

F.2 IPv4 HIP packet (I1)

The IPv4 checksum value for the same example I1 packet is the same as the IPv6 checksum (since the checksums due to the IPv4 and IPv6 pseudo-header components are the same).

F.3 TCP segment

Regardless of whether IPv6 or IPv4 is used, the TCP and UDP sockets use the IPv6 pseudo-header format [8], with the HITs used in place of the IPv6 addresses.

Sender's HIT:	4000::0001	
Receiver's HIT:	4000::0002	
Upper-Layer Packet Length:	20	0x14
Next Header:	6	0x06
Source port:	32769	0x8001
Destination port:	22	0x0016
Sequence number:	1	0x00000001
Acknowledgment number:	0	0x00000000
Header length:	20	0x14
Flags:	SYN	0x02
Window size:	5840	0x16d0
Checksum:	54519	0xd4f7
Urgent pointer:	0	0x0000

Appendix G. 384-bit group

This 384-bit group is defined only to be used with HIP. NOTE: The security level of this group is very low! The encryption may be broken in a very short time, even real-time. It should be used only when the host is not powerful enough (e.g. some PDAs) and when security requirements are low (e.g. during normal web surfing).

This prime is: 2^384 - 2^320 - 1 + 2^64 * { [2^254 pi] + 5857 }

Its hexadecimal value is:

```
FFFFFFFF FFFFFFFF C90FDAA2 2168C234 C4C6628B 80DC1CD1
29024E08 8A67CC74 020BBEA6 3B13B202 FFFFFFFF FFFFFFFF
```

The generator is: 2.

Intellectual Property Statement

Disclaimer of Validity

Copyright Statement

Acknowledgment

Funding for the RFC Editor function is currently provided by the Internet Society.