

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

R. Moskowitz  
ICSALabs, a Division of TruSecure  
Corporation  
P. Nikander  
P. Jokela (editor)  
Ericsson Research NomadicLab  
T. Henderson  
The Boeing Company  
February 21, 2005

**Host Identity Protocol  
draft-ietf-hip-base-02**

Status of this Memo

This document is an Internet-Draft and is subject to all provisions of [Section 3 of RFC 3667](#). By submitting this Internet-Draft, each author represents that any applicable patent or other IPR claims of which he or she is aware have been or will be disclosed, and any of which he or she become aware will be disclosed, in accordance with [RFC 3668](#).

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at <http://www.ietf.org/ietf/1id-abstracts.txt>.

The list of Internet-Draft Shadow Directories can be accessed at <http://www.ietf.org/shadow.html>.

This Internet-Draft will expire on August 25, 2005.

Copyright Notice

Copyright (C) The Internet Society (2005).

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 for protecting subsequent HIP messages, and which can be used for generating session keys for other security protocols, such as IPsec Encapsulaed Security Payload (ESP). 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

<a href="#">1.</a>	<a href="#">Introduction . . . . .</a>	<a href="#">6</a>
<a href="#">1.1</a>	<a href="#">A new name space and identifiers . . . . .</a>	<a href="#">6</a>
<a href="#">1.2</a>	<a href="#">The HIP base exchange . . . . .</a>	<a href="#">6</a>
<a href="#">2.</a>	<a href="#">Conventions used in this document . . . . .</a>	<a href="#">8</a>
<a href="#">3.</a>	<a href="#">Host Identifier (HI) and its representations . . . . .</a>	<a href="#">9</a>
<a href="#">3.1</a>	<a href="#">Host Identity Tag (HIT) . . . . .</a>	<a href="#">9</a>
<a href="#">3.1.1</a>	<a href="#">Restricting HIT values . . . . .</a>	<a href="#">10</a>
<a href="#">3.1.2</a>	<a href="#">Generating a HIT from a HI . . . . .</a>	<a href="#">11</a>
<a href="#">3.2</a>	<a href="#">Local Scope Identifier (LSI) . . . . .</a>	<a href="#">12</a>
<a href="#">4.</a>	<a href="#">Host Identity Protocol . . . . .</a>	<a href="#">14</a>
<a href="#">4.1</a>	<a href="#">HIP base exchange . . . . .</a>	<a href="#">14</a>
<a href="#">4.1.1</a>	<a href="#">HIP Cookie Mechanism . . . . .</a>	<a href="#">15</a>
<a href="#">4.1.2</a>	<a href="#">Authenticated Diffie-Hellman protocol . . . . .</a>	<a href="#">18</a>
<a href="#">4.1.3</a>	<a href="#">HIP replay protection . . . . .</a>	<a href="#">19</a>
<a href="#">4.2</a>	<a href="#">TCP and UDP pseudo-header computation for user data . . . . .</a>	<a href="#">20</a>
<a href="#">4.3</a>	<a href="#">Updating a HIP association . . . . .</a>	<a href="#">20</a>
<a href="#">4.4</a>	<a href="#">Error processing . . . . .</a>	<a href="#">20</a>
<a href="#">4.5</a>	<a href="#">Certificate distribution . . . . .</a>	<a href="#">21</a>
<a href="#">4.6</a>	<a href="#">Sending data on HIP packets . . . . .</a>	<a href="#">21</a>
<a href="#">4.7</a>	<a href="#">Transport Formats . . . . .</a>	<a href="#">21</a>
<a href="#">5.</a>	<a href="#">HIP protocol overview . . . . .</a>	<a href="#">22</a>
<a href="#">5.1</a>	<a href="#">HIP Scenarios . . . . .</a>	<a href="#">22</a>
<a href="#">5.2</a>	<a href="#">Refusing a HIP exchange . . . . .</a>	<a href="#">23</a>
<a href="#">5.3</a>	<a href="#">Reboot and SA timeout restart of HIP . . . . .</a>	<a href="#">23</a>
<a href="#">5.4</a>	<a href="#">HIP State Machine . . . . .</a>	<a href="#">23</a>
<a href="#">5.4.1</a>	<a href="#">HIP States . . . . .</a>	<a href="#">24</a>
<a href="#">5.4.2</a>	<a href="#">HIP State Processes . . . . .</a>	<a href="#">24</a>
<a href="#">5.4.3</a>	<a href="#">Simplified HIP State Diagram . . . . .</a>	<a href="#">28</a>
<a href="#">6.</a>	<a href="#">Packet formats . . . . .</a>	<a href="#">30</a>
<a href="#">6.1</a>	<a href="#">Payload format . . . . .</a>	<a href="#">30</a>
<a href="#">6.1.1</a>	<a href="#">HIP Controls . . . . .</a>	<a href="#">31</a>
<a href="#">6.1.2</a>	<a href="#">Checksum . . . . .</a>	<a href="#">31</a>
<a href="#">6.2</a>	<a href="#">HIP parameters . . . . .</a>	<a href="#">32</a>
<a href="#">6.2.1</a>	<a href="#">TLV format . . . . .</a>	<a href="#">33</a>
<a href="#">6.2.2</a>	<a href="#">Defining new parameters . . . . .</a>	<a href="#">35</a>
<a href="#">6.2.3</a>	<a href="#">R1_COUNTER . . . . .</a>	<a href="#">36</a>
<a href="#">6.2.4</a>	<a href="#">PUZZLE . . . . .</a>	<a href="#">37</a>
<a href="#">6.2.5</a>	<a href="#">SOLUTION . . . . .</a>	<a href="#">38</a>
<a href="#">6.2.6</a>	<a href="#">DIFFIE_HELLMAN . . . . .</a>	<a href="#">39</a>
<a href="#">6.2.7</a>	<a href="#">HIP_TRANSFORM . . . . .</a>	<a href="#">40</a>
<a href="#">6.2.8</a>	<a href="#">HOST_ID . . . . .</a>	<a href="#">41</a>
<a href="#">6.2.9</a>	<a href="#">CERT . . . . .</a>	<a href="#">42</a>
<a href="#">6.2.10</a>	<a href="#">HMAC . . . . .</a>	<a href="#">43</a>
<a href="#">6.2.11</a>	<a href="#">HMAC_2 . . . . .</a>	<a href="#">43</a>
<a href="#">6.2.12</a>	<a href="#">HIP_SIGNATURE . . . . .</a>	<a href="#">44</a>
<a href="#">6.2.13</a>	<a href="#">HIP_SIGNATURE_2 . . . . .</a>	<a href="#">44</a>



6.2.14	SEQ . . . . .	45
6.2.15	ACK . . . . .	45
6.2.16	ENCRYPTED . . . . .	46
6.2.17	NOTIFY . . . . .	47
6.2.18	ECHO_REQUEST . . . . .	50
6.2.19	ECHO_RESPONSE . . . . .	51
6.3	ICMP messages . . . . .	51
6.3.1	Invalid Version . . . . .	51
6.3.2	Other problems with the HIP header and packet structure . . . . .	51
6.3.3	Invalid Cookie Solution . . . . .	52
6.3.4	Non-existing HIP association . . . . .	52
7.	HIP Packets . . . . .	53
7.1	I1 - the HIP initiator packet . . . . .	53
7.2	R1 - the HIP responder packet . . . . .	54
7.3	I2 - the second HIP initiator packet . . . . .	55
7.4	R2 - the second HIP responder packet . . . . .	56
7.5	CER - the HIP Certificate Packet . . . . .	57
7.6	UPDATE - the HIP Update Packet . . . . .	57
7.7	NOTIFY - the HIP Notify Packet . . . . .	58
7.8	CLOSE - the HIP association closing packet . . . . .	59
7.9	CLOSE_ACK - the HIP closing acknowledgment packet . . . . .	59
8.	Packet processing . . . . .	61
8.1	Processing outgoing application data . . . . .	61
8.2	Processing incoming application data . . . . .	62
8.3	HMAC and SIGNATURE calculation and verification . . . . .	63
8.3.1	HMAC calculation . . . . .	63
8.3.2	Signature calculation . . . . .	63
8.4	Initiation of a HIP exchange . . . . .	64
8.4.1	Sending multiple I1s in parallel . . . . .	65
8.4.2	Processing incoming ICMP Protocol Unreachable messages . . . . .	65
8.5	Processing incoming I1 packets . . . . .	66
8.5.1	R1 Management . . . . .	66
8.5.2	Handling malformed messages . . . . .	67
8.6	Processing incoming R1 packets . . . . .	67
8.6.1	Handling malformed messages . . . . .	68
8.7	Processing incoming I2 packets . . . . .	69
8.7.1	Handling malformed messages . . . . .	70
8.8	Processing incoming R2 packets . . . . .	70
8.9	Sending UPDATE packets . . . . .	71
8.10	Receiving UPDATE packets . . . . .	71
8.10.1	Handling a SEQ parameter in a received UPDATE message . . . . .	72
8.10.2	Handling an ACK parameter in a received UPDATE packet . . . . .	72
8.11	Processing CER packets . . . . .	73
8.12	Processing NOTIFY packets . . . . .	73



<a href="#">8.13</a>	Processing CLOSE packets . . . . .	<a href="#">73</a>
<a href="#">8.14</a>	Processing CLOSE_ACK packets . . . . .	<a href="#">73</a>
<a href="#">8.15</a>	Dropping HIP associations . . . . .	<a href="#">73</a>
<a href="#">9.</a>	HIP KEYMAT . . . . .	<a href="#">75</a>
<a href="#">10.</a>	HIP Fragmentation Support . . . . .	<a href="#">77</a>
<a href="#">11.</a>	HIP Policies . . . . .	<a href="#">78</a>
<a href="#">12.</a>	Security Considerations . . . . .	<a href="#">79</a>
<a href="#">13.</a>	IANA Considerations . . . . .	<a href="#">82</a>
<a href="#">14.</a>	Acknowledgments . . . . .	<a href="#">83</a>
<a href="#">15.</a>	References . . . . .	<a href="#">84</a>
<a href="#">15.1</a>	Normative references . . . . .	<a href="#">84</a>
<a href="#">15.2</a>	Informative references . . . . .	<a href="#">85</a>
	Authors' Addresses . . . . .	<a href="#">86</a>
<a href="#">A.</a>	Probabilities of HIT collisions . . . . .	<a href="#">87</a>
<a href="#">B.</a>	Probabilities in the cookie calculation . . . . .	<a href="#">88</a>
<a href="#">C.</a>	Using responder cookies . . . . .	<a href="#">89</a>
<a href="#">D.</a>	Example checksums for HIP packets . . . . .	<a href="#">92</a>
<a href="#">D.1</a>	IPv6 HIP example (I1) . . . . .	<a href="#">92</a>
<a href="#">D.2</a>	IPv4 HIP packet (I1) . . . . .	<a href="#">92</a>
<a href="#">D.3</a>	TCP segment . . . . .	<a href="#">92</a>
<a href="#">E.</a>	384-bit group . . . . .	<a href="#">94</a>
	Intellectual Property and Copyright Statements . . . . .	<a href="#">95</a>

## **1. Introduction**

The Host Identity Protocol (HIP) provides a rapid exchange of Host Identities between two hosts. The protocol is designed to be resistant to Denial-of-Service (DoS) and Man-in-the-middle (MitM) attacks, and when used together with another suitable security protocol, such as Encapsulated Security Payload (ESP) [23], it 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 use as a packet identifier, or as an 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 base exchange**

The HIP base exchange is a two-party cryptographic protocol that consists of four packets. The first party is called the Initiator and the second party the Responder. 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, to be used with Encapsulated Security Payload (ESP) [23] and other end-to-end security protocols. The base protocol lacks the details for security association management and 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 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 defined as 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-bit IPv4-compatible ones and 128-bit IPv6-compatible ones. 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. The 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 expected to be carried explicitly in the headers of user data packets. Depending on the form of further communication, other methods are used to map the data packet to the these representatives of host identities. For example, if ESP is used to protect data traffic, the Security Parameter Index (SPI) can be used for this purpose. In some cases, this makes it possible to use HIP without an additional explicit protocol header.

#### **3.1 Host Identity Tag (HIT)**

The Host Identity Tag is a 128 bits long 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 11](#).

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 [\[29\]](#). 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 [\[25\]](#).

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](#) Restricting HIT values**

To facilitate experimentation and make certain kind of implementations easier, the following restrictions are temporarily placed on HITs. These restriction are to be lifted at the end of 2008. That is, after January 1st 2009, any implementation claiming conformance to this specification MUST accept any HITs from peers and be able to process them normally.

The restrictions: Before the end of 2008, all implementations SHOULD restrict the HITs they generate to ones whose upper-most (left-most) two bits are either binary01 or10. That is, when generating new HIs, if the resulting HIT has as its first two bits as00 or11, the implementation SHOULD generate new HIs until it generates one that fulfills this restriction. Additionally, a conforming implementation



MAY refuse to communicate with a peer that has a HIT with the upper-most bits either 00 or 11. When refusing a HIP connection on this basis, the implementation MAY send an R2 with a NOTIFY payload, with the NOTIFY code being UNSUPPORTED\_HIT\_VALUE\_RANGE. Any such NOTIFYS may be rate-limited

A rationale: One way to experimentally implement HIP is to use unmodified IPv6, TCP and UDP implementations in the stack, using HITs in the place of IPv6 addresses. This modification makes it easier to use existing IPv6 data structures to hold HITs and to distinguish between the two data types. If the IPv6 address space and the HIT value space overlap, it becomes hard to define secure IPsec policies without explicitly tagging the values either as HITs or IPv6 addresses.

### **3.1.2 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 (n\_len) 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 as 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.RSA.n_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.

#### **4. Host Identity Protocol**

The Host Identity Protocol is IP protocol TBD (number will be assigned by IANA). The HIP payload ([Section 6.1](#)) header could be carried in every datagram. However, since HIP datagrams are relatively large (at least 40 bytes), it is desirable to 'compress' the HIP header so that the HIP header only occur in datagrams to establish or change HIP state. The actual method for header 'compression' and matching data packets with existing HIP associations (if any) is defined in separate extension documents, describing transport formats and methods. All HIP implementations MUST implement, at minimum, the ESP transport format for HIP [[23](#)].

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

##### **4.1 HIP base exchange**

The HIP base exchange serves to manage the establishment of state between an Initiator and a Responder. The last three packets of the exchange, R1, I2, and R2, constitute a standard authenticated Diffie-Hellman key exchange for session key generation. During the Diffie-Hellman key exchange, a piece of keying material is generated. The HIP association keys are drawn from this keying material. If other cryptographic keys are needed, e.g., to be used with ESP, they are expected to be drawn from the same keying material.

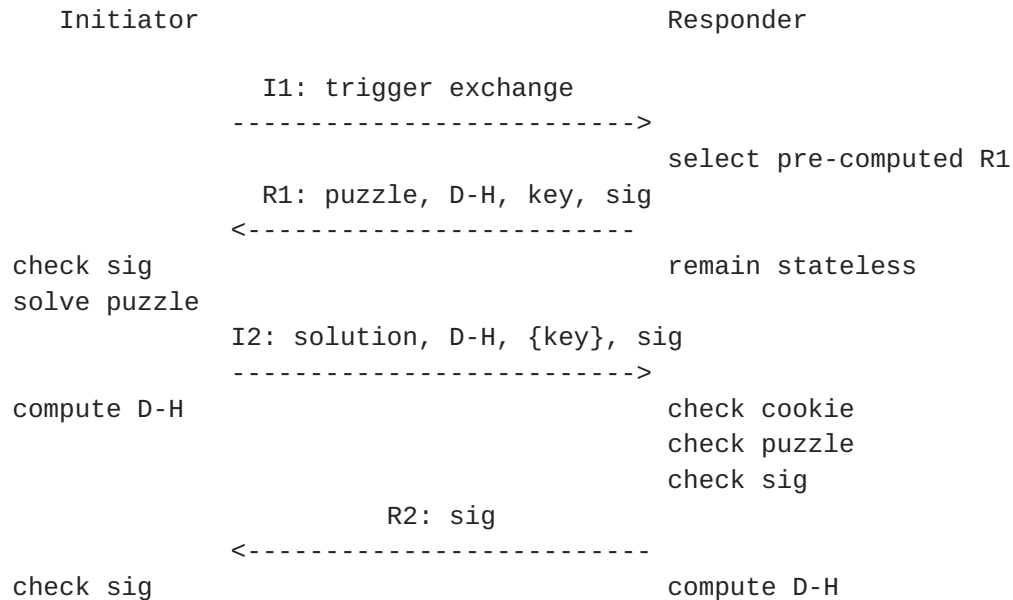
The Initiator first sends a trigger packet, I1, to the Responder. The packet contains only the HIT of the Initiator and possibly the HIT of the Responder, if it is known.

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 base exchange. The packet is signed.

The base exchange is illustrated below.



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 other combinations 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 C](#) 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 [27].

The Responder can set the puzzle 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 puzzle difficulty value K appropriately; see below.

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. The value K sets the difficulty of the puzzle.

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 in the 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 C](#) for an example on how to implement this.

Using the Opaque data field in an ECHO\_REQUEST parameter, 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 in various ways; 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.

NOTE: The protocol developers explicitly considered whether a memory bound function should be used for the puzzle instead of a CPU bound function. The decision was not to use memory bound functions. At the time of the decision the idea of memory bound functions was relatively new and their IPR status were unknown. Once there is more experience about memory bound functions and once their IPR status is better known, it may be reasonable to reconsider this decision.

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 puzzle 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 an R1.  
Saves I and K for a Delta time.

#### Initiator

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

$$\text{Ltrunc}(\text{SHA-1}(\text{I} \parallel \text{HIT-I} \parallel \text{HIT-R} \parallel \text{J}), \text{K}) == 0$$
  
Sends I and J in an I2.

#### Responder

Verifies that the received I is a saved one.  
Finds the right K based on I.  
Computes  $V := \text{Ltrunc}(\text{SHA-1}(\text{I} \parallel \text{HIT-I} \parallel \text{HIT-R} \parallel \text{J}), \text{K})$   
Rejects if  $V \neq 0$   
Accept if  $V == 0$

The  $\text{Ltrunc}(\text{SHA-1}(), \text{K})$  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 association using keying material from the session key (see [Section 9](#)), and uses the association to encrypt its public authentication key, i.e., host identity. The resulting I2 contains the Initiator's Diffie-Hellman key and its 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 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 base exchange. 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 for user data**

When computing TCP and UDP checksums on user data packets that flow through 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 user data security associations, add new security associations, or change IP addresses associated with hosts. The UPDATE packet is used for those and other similar purposes. This document only specifies the UPDATE packet format and basic processing rules, with mandatory TLVs. The actual usage is defined in separate specifications.

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 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 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 6.2.9](#), 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 user data on various HIP packets. However, currently the HIP header is a terminal header, and not followed by any other headers.

#### **[4.7](#) Transport Formats**

The actual data transmission format, used for user data after the HIP base exchange, is not defined in this document. Such transport formats and methods are described in separate specifications. All HIP implementations MUST implement, at minimum, the ESP transport format for HIP [[23](#)].

When new transport formats are defined, the corresponding parameters MUST have smaller type value than the ESP\_TRANSFORM parameter. The order in which the transport formats are presented in the R1 packet, is the preferred order. The last of the transport formats MUST be ESP transport format, represented by the ESP\_TRANSFORM parameter.

## 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 (data)
I <-- R (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 HIP association.

The system with data to send has no state with the receiver, but the receiver has a residual HIP association.

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 association is 'discovered' and deleted, and the new association is established.

The system with data to send has an HIP association, but the receiver does not.

The system sends data on the outbound user data security association. The receiver 'detects' the situation when it receives a user data packet that it cannot match to any HIP association. The receiving host MUST discard this packet. Optionally, the receiving host MAY send an ICMP packet with the Parameter Problem type to inform about non-existing HIP association (see [Section 6.3](#)), and it MAY initiate a new HIP



negotiation. However, responding with these optional mechanisms is implementation or policy dependent.

## **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 a user data packet that cannot be matched to any existing HIP association, 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 referred HIP-related association information. Reacting to such traffic depends on the implementation and the environment where the implementation is used.

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 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, and R2 packets only. Other states may be introduced by mechanisms in other drafts (such as mobility and multihoming).

#### [5.4.1](#) HIP States

State	Explanation
UNASSOCIATED	State machine start
I1-SENT	Initiating HIP
I2-SENT	Waiting to finish HIP
R2-SENT	Waiting to finish HIP
ESTABLISHED	HIP association established
CLOSING	HIP association closing, no data can be sent
CLOSED	HIP association closed, no data can be sent
E-FAILED	HIP exchange failed

#### [5.4.2](#) HIP State Processes

```

+-----+
|UNASSOCIATED| Start state
+-----+

```

User data to send requiring a new HIP association, 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 user data for unknown HIP association, optionally send ICMP as defined in

[Section 6.3](#)

and stay at UNASSOCIATED

Receive CLOSE, 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

Move to ESTABLISHED after an implementation specific time.

+-----+

| ESTABLISHED | HIP association 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 HIP association, establish a new  
    HIP association, to to R2-SENT

    if fail, stay at ESTABLISHED

Receive R1, drop and stay at ESTABLISHED

Receive R2, drop and stay at ESTABLISHED

Receive user data for HIP association, process and stay at ESTABLISHED

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.

User data 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
```

```

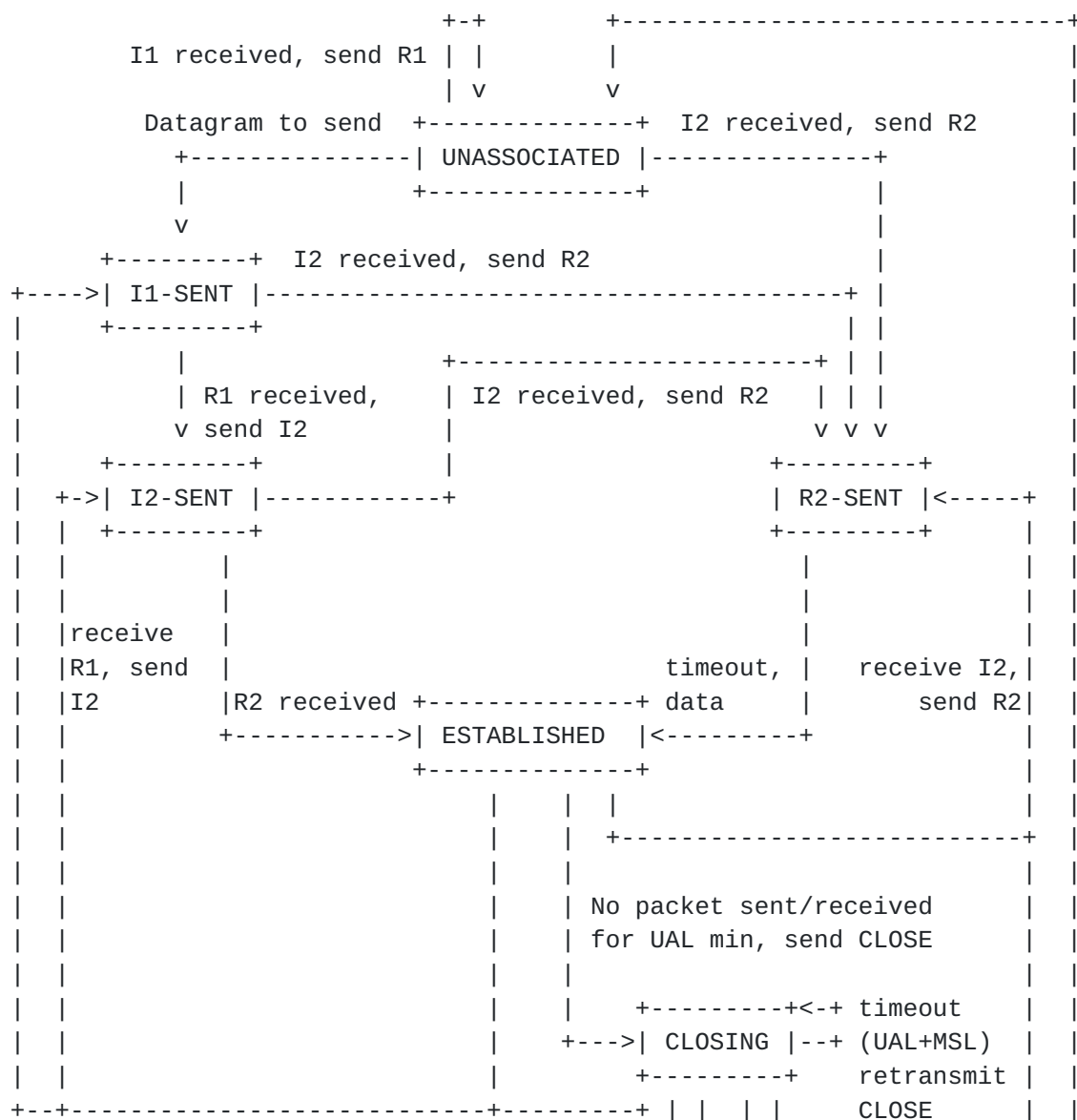
+-----+
| E-FAILED | HIP failed to establish association with peer
+-----+

```

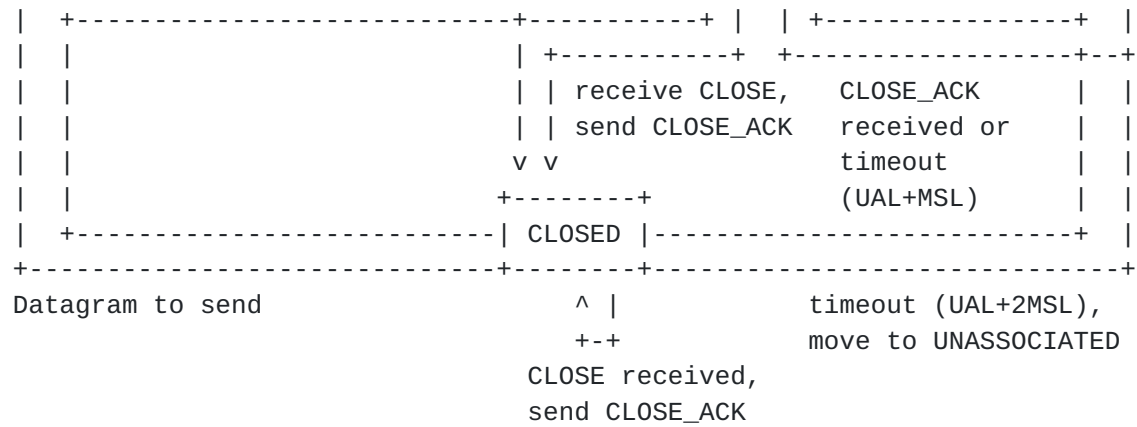
Move to UNASSOCIATED after an implementation specific time. Re-negotiation is possible after moving to UNASSOCIATED state.

### 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.



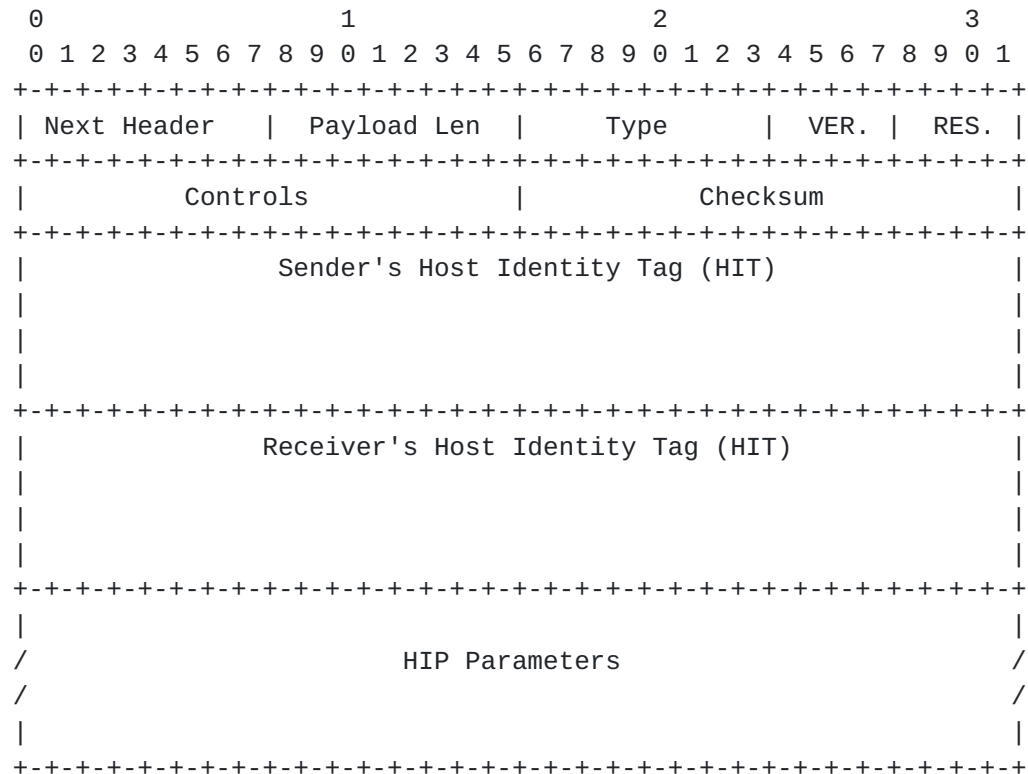




## 6. Packet formats

### 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 an unimplemented Next Header value is received.

The Header Length field contains the length of the HIP Header and 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  $(255 \times 8) - 32 = 2008$  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:

```
+-----+
| SHT | DHT | | | | | | | | C | A |
+-----+
```

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.

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-aware 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:  $(\text{HIP Header Length} + 1) * 8$ .

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 and other information. The HIP Parameters consists of ordered parameters, encoded in TLV format.

The following parameter types are currently defined.

TLV	Type	Length	Data
R1_COUNTER	2	12	System Boot Counter
PUZZLE	5	12	K and Random #I
SOLUTION	7	20	K, Random #I and puzzle solution J
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
ENCRYPTED	21	variable	Encrypted part of I2 or CER packets
HOST_ID	35	variable	Host Identity with Fully Qualified Domain Name or NAI
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, except for type values from 2048 to 4095 which are reserved for new transport forms. 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.

Parameters using type values from 2048 up to 4095 are transport formats. Currently, one transport format is defined: the ESP transport format [23]. The order of these parameters does not follow the order of their type value, but they are put in the packet in order of preference. First one of the transport formats is the most preferred, and so on.



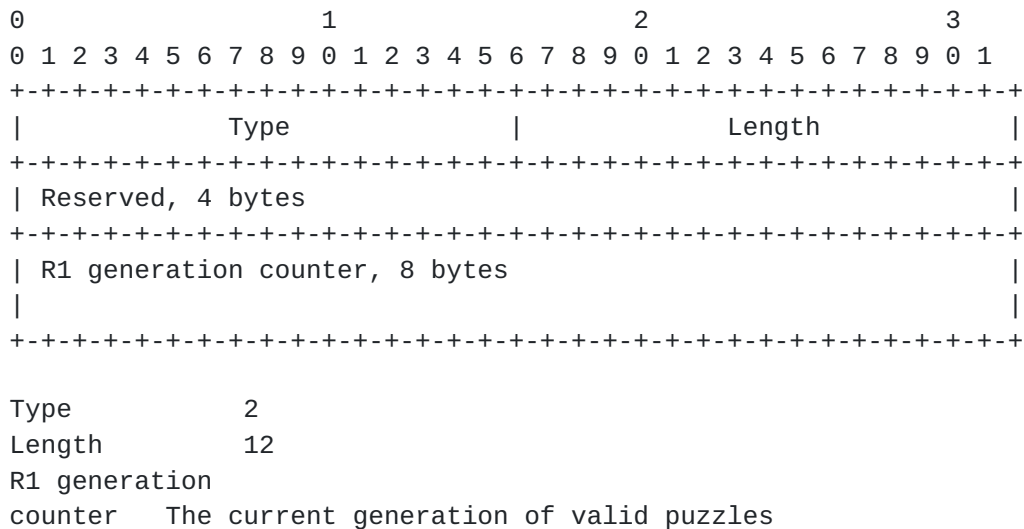
### **6.2.2 Defining new parameters**

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 13](#).

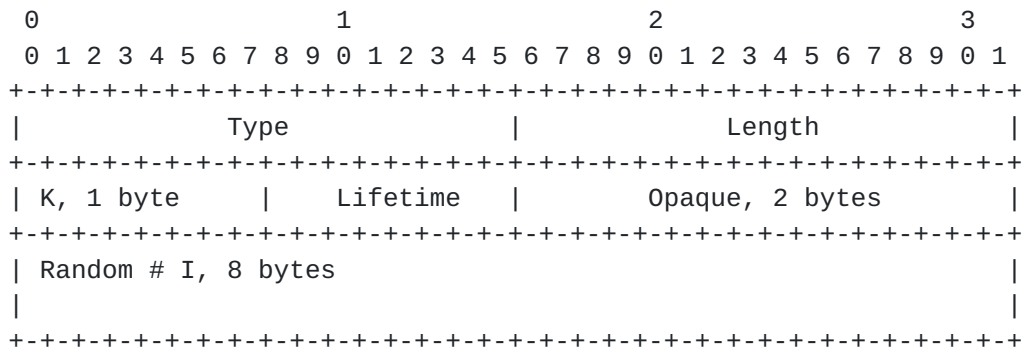
### 6.2.3 R1\_COUNTER



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 in verbatim) by the Initiator in the I2.

### 6.2.4 PUZZLE



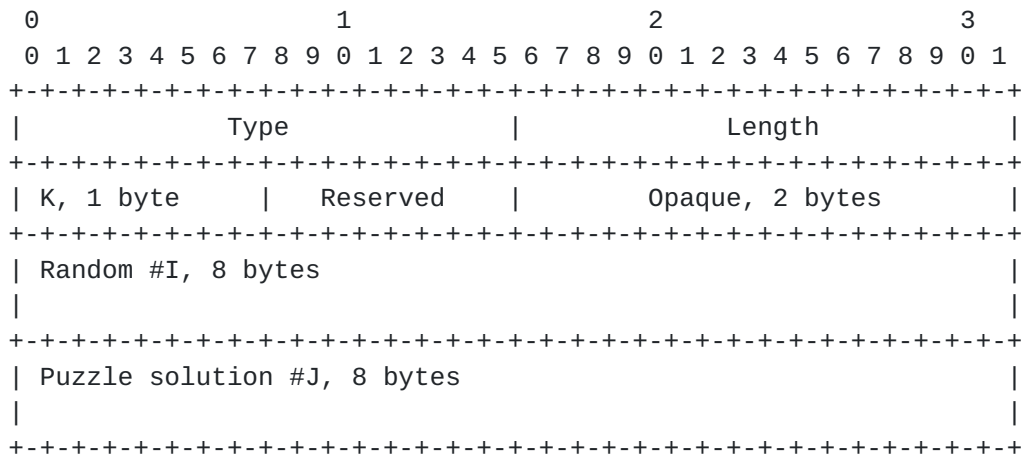
Type	5
Length	12
K	K is the number of verified bits
Lifetime	Puzzle lifetime $2^{(\text{value}-32)}$ seconds
Opaque	Data set by the Responder, indexing the puzzle
Random #I	random number

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  $2^{(\text{Lifetime}-32)}$  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.

### 6.2.5 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.

6.2.6 DIFFIE\_HELLMAN

0															1															2															3									
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1																							
+-----																																																						

The following Group IDs have been defined:

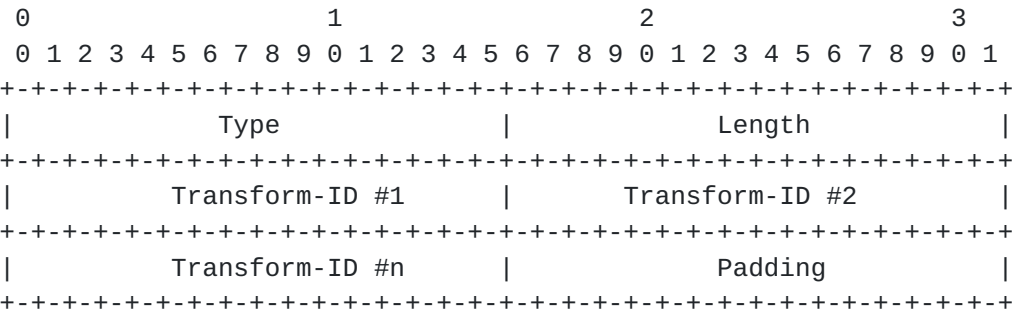
Group	Value
Reserved	0
384-bit group	1
OAKLEY well known group 1	2
1536-bit MODP group	3
3072-bit MODP group	4
6144-bit MODP group	5
8192-bit MODP group	6

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 E.

To avoid unnecessary failures during the base exchange, the rest of the groups SHOULD be implemented in hosts where resources are adequate.

6.2.7 HIP\_TRANSFORM



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

The following Suite-IDs are defined ([20],[24]):

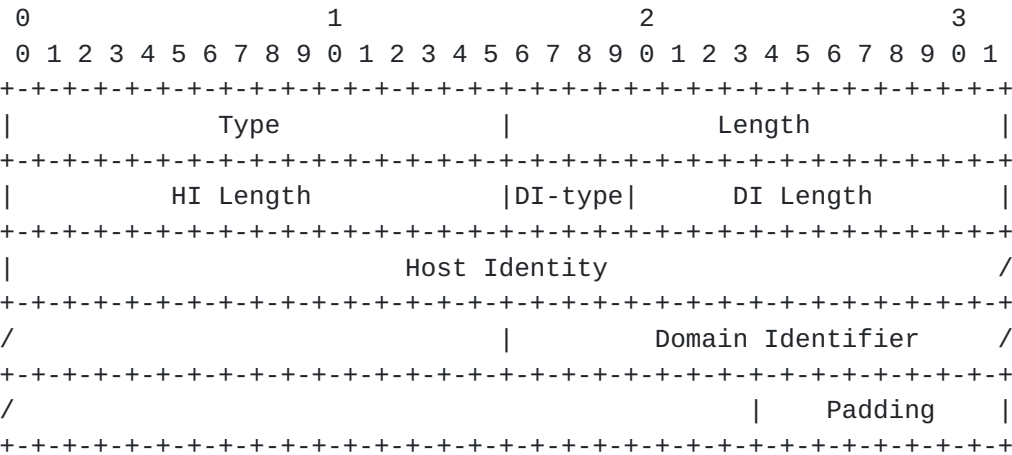
XXX: Deprecate MD5 in the light of recent development?

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

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: AES-CBC with HMAC-SHA1 and NULL-ENCRYPTION with HMAC-SHA1.

6.2.8 HOST\_ID



Type 35  
Length length in octets, excluding Type, Length, and Padding  
HI Length Length of the Host Identity in octets  
DI-type type of the following Domain Identifier field  
DI Length length of the FQDN or NAI in octets  
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:

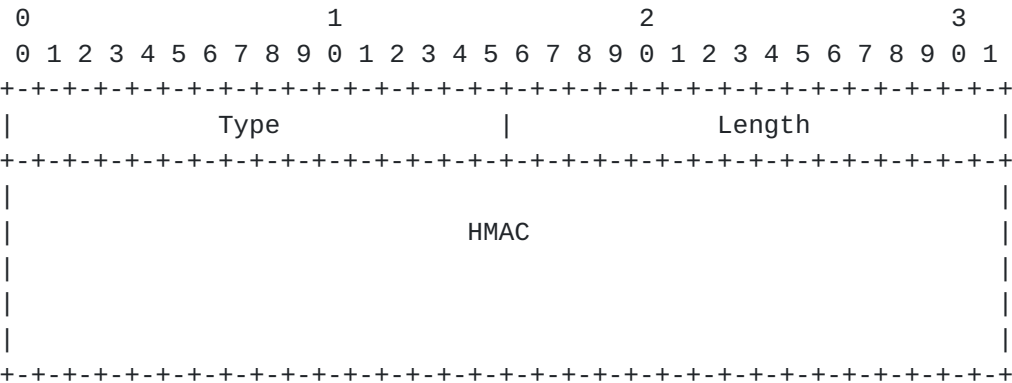
Algorithms	Values
RESERVED	0
DSA	3 [ <a href="#">RFC2536</a> ] (RECOMMENDED)
RSA	5 [ <a href="#">RFC3110</a> ] (REQUIRED)

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.



6.2.10 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](#)

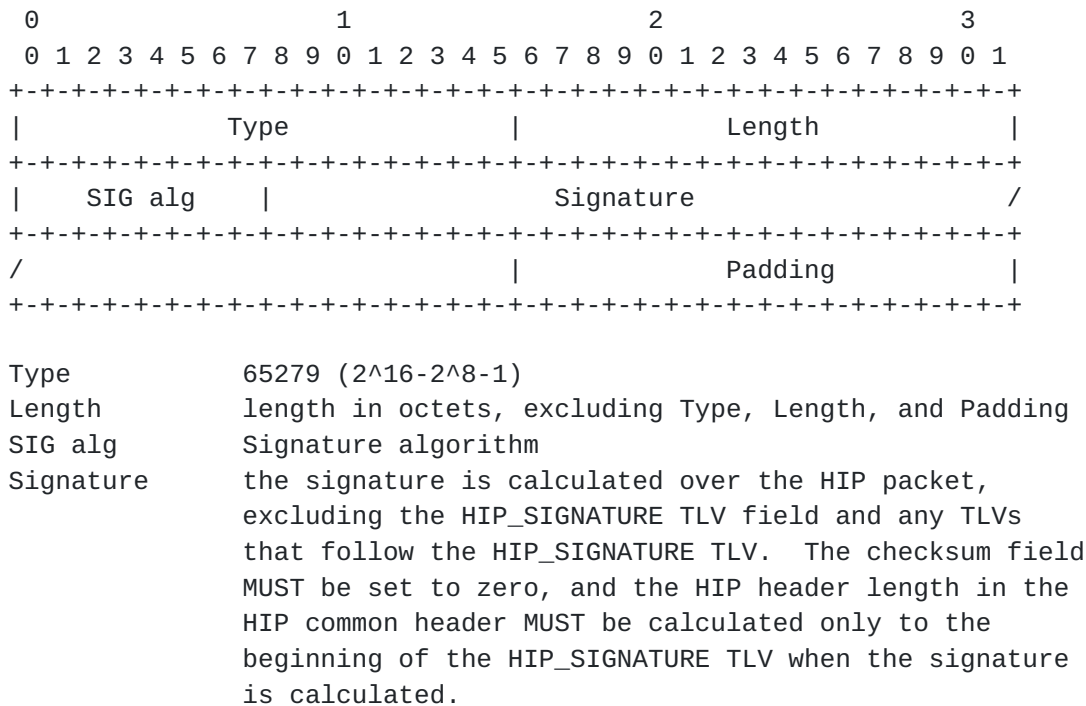
6.2.11 HMAC\_2

The TLV structure is the same as in [Section 6.2.10](#). 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](#)

### 6.2.12 HIP\_SIGNATURE



The signature algorithms are defined in [Section 6.2.8](#). 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](#)

### **6.2.13 HIP\_SIGNATURE\_2**

The TLV structure is the same as in [Section 6.2.12](#). 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.



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



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.

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



```

0          1          2          3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-----+-----+-----+-----+-----+-----+-----+-----+
|                                     |                               |
|      Type                         |       Length                 |
+-+-+-+-----+-----+-----+-----+-----+-----+-----+-----+
|               Opaque data (variable length)                       |
+-+-+-+-----+-----+-----+-----+-----+-----+-----+-----+

```

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 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 from 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].

### **6.3.4 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 HIP base exchange, one is for updating, one is for sending certificates, one for sending notifications, and two for closing a HIP association.

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,
          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 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 6.2.13](#). 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 ( [R1\_COUNTER,]  
          SOLUTION,  
          DIFFIE\_HELLMAN,  
          HIP\_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 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 6.2.12](#). 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 ( 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 ( [SEQ, ACK, ] 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.

## **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\_REQUEST 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. 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. In the case of user data carried in a specific transport format, the transport format document specifies how the incoming packets are matched with the active associations.

### **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 [\[29\]](#)) or a completely new API. The exact format and method for transferring the data from the source HIP host to the destination HIP host is defined in the corresponding transport format document. The actual data is transmitted in the network using the 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 is 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 passed to transport handling. The possible transport formats are defined in separate documents, of which the ESP transport format for HIP is mandatory for all HIP implementations.
  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 wrapping the packet into the output format.

## **8.2 Processing incoming application data**

The transport format and method (defined in separate specifications) determines the format in which incoming HIP packets arrive to the host. The following steps define the conceptual processing rules for incoming datagrams. The specific transport format and method specifications define in more detail the packet processing, related to the method.

1. The incoming datagram is mapped to an existing HIP association, typically using some information from the packet. For example, such mapping may be based on IPsec Security Parameter Index (SPI) or a protocol port number.
2. The specific transport format is unwrapped, in a way depending on the transport format, yielding a packet that looks like a standard (unencrypted) IP packet. If possible, this step SHOULD also verify that the packet was indeed (once) sent by the remote HIP host, as identified by the HIP association.
3. The IP addresses in the datagram are replaced with the HITs associated with the HIP association. Note that this IP-address-to-HIT conversion step MAY also be performed at some other point in the stack.
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 6.2.10](#) and HMAC\_2 TLV in [Section 6.2.11](#). 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 6.2.12](#) and the



HIP\_SIGNATURE\_2 TLV in [Section 6.2.13](#).

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 in 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 an 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 6.2.13](#).
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 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 initializes the remaining variables in the associated state, including Update ID counters.
15. The system prepares and sends an I2, as described in [Section 7.3](#).
16. 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.
17. 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 C](#) 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 the HIP\_TRANSFORM parameter, which MUST 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 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 6.2.10](#).
  11. The system MUST verify the HIP\_SIGNATURE according to [Section 6.2.12](#) 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 system initializes the remaining variables in the associated state, including Update ID counters.
  16. 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.
  17. Upon successful processing of an I2 in state ESTABLISHED, the old HIP association is dropped and a new one is installed, an R2 is sent, and the state machine transitions to R2-SENT.
  18. 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 HIP association, 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 6.2.11](#).
3. The system MUST verify the HIP signature according to the procedures in [Section 6.2.12](#).
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. Upon successful processing of the R2, the state machine moves to state ESTABLISHED.

### **[8.9](#) Sending UPDATE packets**

A host sends an UPDATE packet when it wants to update some information related to a HIP association. There are a number of likely situations, e.g. mobility management and rekeying of an existing ESP Security Association. The following paragraphs define the conceptual rules for sending an UPDATE packet to the peer. Additional steps can be defined in other documents where the UPDATE packet is used.

1. The system increments its own Update ID value by one.
2. The system creates an UPDATE packet that contains a SEQ parameter with the current value of Update ID. The UPDATE packet may also include an ACK of the Update ID found in the received UPDATE SEQ parameter, if any.
3. The system sends the created UPDATE packet and starts an UPDATE timer. The default value for the timer is  $2 * \text{RTT estimate}$ .
4. If the UPDATE timer expires, the UPDATE is resent. The UPDATE can be resent UPDATE\_RETRY\_MAX times. The UPDATE timer SHOULD be exponentially backed off for subsequent retransmissions.

### **[8.10](#) Receiving 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. Typically, an UPDATE message also carries optional parameters whose handling is defined in separate documents.

1. If there is no corresponding HIP association, the implementation MAY reply with an ICMP Parameter Problem, as specified in [Section 6.3.4](#).



2. If the association is in the ESTABLISHED state and the SEQ parameter is present, the UPDATE is processed and replied as described in [Section 8.10.1](#).
3. Additionally (or alternatively), if the association is in the ESTABLISHED state and there is an ACK (of outstanding Update ID) in the UPDATE, the UPDATE is processed as described in [Section 8.10.2](#).

#### **[8.10.1](#) Handling a SEQ parameter in a received UPDATE message**

1. If the Update ID in the received SEQ is smaller than the stored Update ID for the peer host, the packet MUST BE dropped as a duplicate.
2. If the Update ID in the received SEQ is equal to the stored Update ID for the host, the packet is treated as a retransmission. 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. The system MUST acknowledge, again, 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. The system MAY verify the SIGNATURE in the UPDATE packet. If the verification fails, the packet SHOULD be dropped and an error message logged.
5. If a new SEQ parameter is being processed, the system MUST record the Update ID in the received SEQ parameter, for replay protection.
6. An UPDATE acknowledgement packet with ACK parameter is prepared and send to the peer.

#### **[8.10.2](#) Handling an ACK parameter in a received UPDATE packet**

1. The UPDATE packet with ACK must match to an earlier sent UPDATE packet. If no match is found, the packet MUST be dropped.
2. The system MUST verify the HMAC in the UPDATE packet. If the verification fails, the packet MUST be dropped.
3. The system MAY verify the SIGNATURE in the UPDATE packet. If the verification fails, the packet SHOULD be dropped and an error message logged.
4. The corresponding UPDATE timer is stopped (see [Section 8.9](#)) so that the now acknowledged UPDATE is no longer retransmitted.

### **8.11 Processing CER packets**

Processing CER packets is OPTIONAL, and currently undefined.

### **8.12 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.

### **8.13 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 6.3.4](#).

### **8.14 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.

### **8.15 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 a HIP association, it deletes 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.

## 9. HIP KEYMAT

HIP keying material is derived from the Diffie-Hellman Kij produced during the HIP base 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

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

## **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. 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.

## **12. Security Considerations**

HIP is designed to provide secure authentication of hosts. HIP also attempts to limit the exposure of the host to various denial-of-service and man-in-the-middle (MitM) 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.

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 was not sent.

A second form of DoS attack arrives in the I2 HIP packet. Once the attacking Initiator has solved the puzzle, 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 the ESTABLISHED state. 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 signal 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 6.3.4](#)) 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.

### **13. IANA Considerations**

IANA has assigned IP Protocol number TBD to HIP.

IANA needs to create registries for:

1. HIP packet types
2. HIP parameter types

#### **14. Acknowledgments**

The drive to create HIP came to being after attending the MALLOC meeting at the 43rd IETF meeting. 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, which later evolved into the R1 generation counter, 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 transferred 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 whose name is missing.

Once the HIP Working Group was founded in early 2004, a number of changes were introduced through the working group process. Most notably, the original draft was split in two, one containing the base exchange and the other one defining how to use ESP.

## **15. References**

### **15.1 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)", Internet-Draft [draft-ietf-ipsec-esp-v3-05](#), April 2003.
- [20] Kaufman, C., "Internet Key Exchange (IKEv2) Protocol", Internet-Draft [draft-ietf-ipsec-ikev2-07](#), April 2003.
- [21] Moskowitz, R., "Host Identity Protocol Architecture", Internet-Draft [draft-moskowitz-hip-arch-03](#), May 2003.
- [22] NIST, "FIPS PUB 180-1: Secure Hash Standard", April 1995.
- [23] Jokela, P., Moskowitz, R. and P. Nikander, "Using ESP transport format with HIP", Internet-Draft [draft-jokela-hip-esp-00](#), January 2005.

## **15.2 Informative references**

- [24] Bellovin, S. and W. Aiello, "Just Fast Keying (JFK)", Internet-Draft [draft-ietf-ipsec-jfk-04](#), July 2002.
- [25] Moskowitz, R. and P. Nikander, "Using Domain Name System (DNS) with Host Identity Protocol (HIP)", Internet-Draft [draft-nikander-hip-dns-00](#) (to be issued), June 2003.
- [26] Nikander, P., "SPI assisted NAT traversal (SPINAT) with Host Identity Protocol (HIP)", Internet-Draft [draft-nikander-hip-nat-00](#) (to be issued), June 2003.
- [27] Crosby, SA. and DS. Wallach, "Denial of Service via Algorithmic Complexity Attacks", in Proceedings of Usenix Security Symposium 2003, Washington, DC., August 2003.

- [28] Nikander, P., "A Bound End-to-End Tunnel (BEET) mode for ESP", Internet-Draft [draft-nikander-esp-beet-mode-00](#) (expired), Oct 2003.
- [29] Henderson, T., "Using HIP with Legacy Applications", Internet-Draft [draft-henderson-hip-applications-00.txt](#), Feb 2005.

## Authors' Addresses

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

Email: [rgm@icsalabs.com](mailto:rgm@icsalabs.com)

Pekka Nikander  
Ericsson Research NomadicLab  
JORVAS FIN-02420  
FINLAND

Phone: +358 9 299 1  
Email: [pekka.nikander@nomadiclab.com](mailto:pekka.nikander@nomadiclab.com)

Petri Jokela  
Ericsson Research NomadicLab  
JORVAS FIN-02420  
FINLAND

Phone: +358 9 299 1  
Email: [petri.jokela@nomadiclab.com](mailto:petri.jokela@nomadiclab.com)

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

Email: [thomas.r.henderson@boeing.com](mailto:thomas.r.henderson@boeing.com)

**[Appendix A](#). 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  $9 \times 10^{16}$  population.

## [Appendix B](#). 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 a 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

$$\lim_{k \rightarrow \infty} (1 - 2^{-k})^{(2^k)} = 0.36788$$

$$\lim_{k \rightarrow \infty} (1 - 2^{-k})^{(2^{(k+1)})} = 0.13534$$

$$\lim_{k \rightarrow \infty} (1 - 2^{-k})^{(2^{(k+2)})} = 0.01832$$

$$\lim_{k \rightarrow \infty} (1 - 2^{-k})^{(2^{(k+3)})} = 0.000335$$

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 C](#). 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  $f$ :

$$f(\text{IP-I}, \text{IP-R}, \text{HIT-I}, \text{HIT-R}) \rightarrow \text{cookie-index}$$

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 D](#). 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.

### [D.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	

### [D.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).

### [D.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 E](#). 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:  $2^{384} - 2^{320} - 1 + 2^{64} * \{ [ 2^{254} \text{ pi} ] + 5857 \}$

Its hexadecimal value is:

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

The generator is: 2.

## Intellectual Property Statement

The IETF takes no position regarding the validity or scope of any Intellectual Property Rights or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; nor does it represent that it has made any independent effort to identify any such rights. Information on the procedures with respect to rights in RFC documents can be found in [BCP 78](#) and [BCP 79](#).

Copies of IPR disclosures made to the IETF Secretariat and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this specification can be obtained from the IETF on-line IPR repository at <http://www.ietf.org/ipr>.

The IETF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights that may cover technology that may be required to implement this standard. Please address the information to the IETF at [ietf-ipr@ietf.org](mailto:ietf-ipr@ietf.org).

## Disclaimer of Validity

This document and the information contained herein are provided on an "AS IS" basis and THE CONTRIBUTOR, THE ORGANIZATION HE/SHE REPRESENTS OR IS SPONSORED BY (IF ANY), THE INTERNET SOCIETY AND THE INTERNET ENGINEERING TASK FORCE DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

## Copyright Statement

Copyright (C) The Internet Society (2005). This document is subject to the rights, licenses and restrictions contained in [BCP 78](#), and except as set forth therein, the authors retain all their rights.

## Acknowledgment

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