Network Working Group
Internet-Draft
Obsoletes: 5201 (if approved)
Intended status: Standards Track
Expires: September 10, 2010

R. Moskowitz, Ed.
ICSAlabs
P. Nikander
P. Jokela
Ericsson Research NomadicLab
T. Henderson
The Boeing Company
T. Heer
RWTH Aachen University,
Distributed Systems Group

March 9, 2010

Host Identity Protocol
draft-moskowitz-hip-rfc5201-bis-01

Abstract

This document specifies the details of the Host Identity Protocol (HIP). HIP allows consenting hosts to securely establish and maintain shared IP-layer state, allowing separation of the identifier and locator roles of IP addresses, thereby enabling continuity of communications across IP address changes. HIP is based on a Sigma-compliant Diffie-Hellman key exchange, using public key identifiers from a new Host Identity namespace for mutual peer authentication. The protocol is designed to be resistant to denial-of-service (DoS) and man-in-the-middle (MitM) attacks. When used together with another suitable security protocol, such as the Encapsulated Security Payload (ESP), it provides integrity protection and optional encryption for upper-layer protocols, such as TCP and UDP.

This document obsoletes RFC 5201 and addresses the concerns raised by the IESG, particularly that of crypto agility. It also incorporates lessons learned from the implementations of RFC 5201.

Status of This Memo

This Internet-Draft is submitted to IETF in full conformance with the provisions of BCP 78 and BCP 79.

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 September 10, 2010.

Copyright Notice

Copyright (c) 2010 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the BSD License.

This document may contain material from IETF Documents or IETF Contributions published or made publicly available before November 10, 2008. The person(s) controlling the copyright in some of this material may not have granted the IETF Trust the right to allow modifications of such material outside the IETF Standards Process. Without obtaining an adequate license from the person(s) controlling the copyright in such materials, this document may not be modified outside the IETF Standards Process, and derivative works of it may not be created outside the IETF Standards Process, except to format it for publication as an RFC or to translate it into languages other than English.

Table of Contents

1. Introduction 6
1.1. A New Namespace and Identifiers 6
1.2. The HIP Base Exchange 7
1.3. Memo Structure 8
2. Terms and Definitions 8
2.1. Requirements Terminology 8
2.2. Notation 8
2.3. Definitions 8
3. Host Identifier (HI) and Its Representations 9

3.1. Host Identity Tag (HIT) 10
3.2. Generating a HIT from an HI 10
4. Protocol Overview 11
4.1. Creating a HIP Association 12
4.1.1. HIP Puzzle Mechanism 13
4.1.2. Puzzle Exchange 14
4.1.3. Authenticated Diffie-Hellman Protocol 15
4.1.4. HIP Replay Protection 16
4.1.5. Refusing a HIP Exchange 17
4.1.6. HIP Opportunistic Mode 17
4.2. Updating a HIP Association 19
4.3. Error Processing 20
4.4. HIP State Machine 21
4.4.1. HIP States 21
4.4.2. HIP State Processes 22
4.4.3. Simplified HIP State Diagram 29
4.5. User Data Considerations 31
4.5.1. TCP and UDP Pseudo-Header Computation for User Data . 31
4.5.2. Sending Data on HIP Packets 31
4.5.3. Transport Formats 31
4.5.4. Reboot and SA Timeout Restart of HIP 31
4.6. Certificate Distribution 32
5. Packet Formats 32
5.1. Payload Format 32
5.1.1. Checksum 33
5.1.2. HIP Controls 34
5.1.3. HIP Fragmentation Support 34
5.2. HIP Parameters 35
5.2.1. TLV Format 38
5.2.2. Defining New Parameters 39
5.2.3. R1_COUNTER 40
5.2.4. PUZZLE 41
5.2.5. SOLUTION 42
5.2.6. DIFFIE_HELLMAN 43
5.2.7. HIP_TRANSFORM 44
5.2.8. HOST_ID 45
5.2.9. OGA_LIST 46
5.2.10. HMAC 47
5.2.11. HMAC_2 47
5.2.12. HIP_SIGNATURE 48
5.2.13. HIP_SIGNATURE_2 49
5.2.14. SEQ 49
5.2.15. ACK 50
5.2.16. ENCRYPTED 51
5.2.17. NOTIFICATION 52
5.2.18. ECHO_REQUEST_SIGNED 56
5.2.19. ECHO_REQUEST_UNSIGNED 56
5.2.20. ECHO_RESPONSE_SIGNED 57

5.2.21. ECHO_RESPONSE_UNSIGNED 58
5.3. HIP Packets 58
5.3.1. I1 - the HIP Initiator Packet 59
5.3.2. R1 - the HIP Responder Packet 60
5.3.3. I2 - the Second HIP Initiator Packet 62
5.3.4. R2 - the Second HIP Responder Packet 63
5.3.5. UPDATE - the HIP Update Packet 64
5.3.6. NOTIFY - the HIP Notify Packet 65
5.3.7. CLOSE - the HIP Association Closing Packet 65
5.3.8. CLOSE_ACK - the HIP Closing Acknowledgment Packet . . 66
5.4. ICMP Messages 66
5.4.1. Invalid Version 67
5.4.2. Other Problems with the HIP Header and Packet Structure 67
5.4.3. Invalid Puzzle Solution 67
5.4.4. Non-Existing HIP Association 67
6. Packet Processing 68
6.1. Processing Outgoing Application Data 68
6.2. Processing Incoming Application Data 69
6.3. Solving the Puzzle 70
6.4. HMAC and SIGNATURE Calculation and Verification 71
6.4.1. HMAC Calculation 71
6.4.2. Signature Calculation 73
6.5. HIP KEYMAT Generation 75
6.6. Initiation of a HIP Exchange 77
6.6.1. Sending Multiple I1s in Parallel 78
6.6.2. Processing Incoming ICMP Protocol Unreachable Messages 78
6.7. Processing Incoming I1 Packets 78
6.7.1. R1 Management 80
6.7.2. Handling Malformed Messages 80
6.8. Processing Incoming R1 Packets 80
6.8.1. Handling Malformed Messages 82
6.9. Processing Incoming I2 Packets 82
6.9.1. Handling Malformed Messages 85
6.10. Processing Incoming R2 Packets 85
6.11. Sending UPDATE Packets 86
6.12. Receiving UPDATE Packets 87
6.12.1. Handling a SEQ Parameter in a Received UPDATE Message 87
6.12.2. Handling an ACK Parameter in a Received UPDATE Packet 88
6.13. Processing NOTIFY Packets 89
6.14. Processing CLOSE Packets 89
6.15. Processing CLOSE_ACK Packets 89
6.16. Handling State Loss 90
7. HIP Policies 90
8. Security Considerations 90

9. IANA Considerations 93
10. Acknowledgments 95
11. References 96
11.1. Normative References 96
11.2. Informative References 97
Appendix A. Using Responder Puzzles 99
Appendix B. Generating a Public Key Encoding from an HI 100
Appendix C. Example Checksums for HIP Packets 101
C.1. IPv6 HIP Example (I1) 102
C.2. IPv4 HIP Packet (I1) 102
C.3. TCP Segment 102
Appendix D. 384-Bit Group 103
Appendix E. OAKLEY Well-Known Group 1 103
Appendix F. ORCHID Generation Algorithms for HITs 103
F.1. SHA-1 Truncated to 96 Bits 104
F.2. SHA-256 Truncated to 96 Bits 104

1. Introduction

This memo specifies the details of the Host Identity Protocol (HIP). A high-level description of the protocol and the underlying architectural thinking is available in the separate HIP architecture description [rfc4423-bis]. Briefly, the HIP architecture proposes an alternative to the dual use of IP addresses as "locators" (routing labels) and "identifiers" (endpoint, or host, identifiers). In HIP, public cryptographic keys, of a public/private key pair, are used as Host Identifiers, to which higher layer protocols are bound instead of an IP address. By using public keys (and their representations) as host identifiers, dynamic changes to IP address sets can be directly authenticated between hosts, and if desired, strong authentication between hosts at the TCP/IP stack level can be obtained.

This memo specifies the base HIP protocol ("base exchange") used between hosts to establish an IP-layer communications context, called HIP association, prior to communications. It also defines a packet format and procedures for updating an active HIP association. Other elements of the HIP architecture are specified in other documents, such as.

- "Using the Encapsulating Security Payload (ESP) Transport Format with the Host Identity Protocol (HIP)" [RFC5202]: how to use the Encapsulating Security Payload (ESP) for integrity protection and optional encryption

- "End-Host Mobility and Multihoming with the Host Identity Protocol" [RFC5206]: how to support mobility and multihoming in HIP

- "Host Identity Protocol (HIP) Domain Name System (DNS) Extensions" [RFC5205]: how to extend DNS to contain Host Identity information

- "Host Identity Protocol (HIP) Rendezvous Extension" [RFC5204]: using a rendezvous mechanism to contact mobile HIP hosts

1.1. A New Namespace and Identifiers

The Host Identity Protocol introduces a new namespace, the Host Identity namespace. Some ramifications of this new namespace are explained in the HIP architecture description [rfc4423-bis].

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. The HIT has an important security property in that it is self-certifying (see Section 3).

1.2. The HIP Base Exchange

The HIP base exchange is a two-party cryptographic protocol used to establish communications context between hosts. The base exchange is a Sigma-compliant [KRA03] four-packet exchange. 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, the Responder starts a puzzle exchange in the 2nd packet, with the Initiator completing it in the 3rd packet before the Responder stores any state from the exchange.

The exchange can use the Diffie-Hellman output to encrypt the Host Identity of the Initiator in the 3rd packet (although Aura, et al., [AUR03] notes that such operation may interfere with packet-inspecting middleboxes), or the Host Identity may instead be sent unencrypted. 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 to flow 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.

An existing HIP association can be updated using the update mechanism defined in this document, and when the association is no longer needed, it can be closed using the defined closing mechanism.

Finally, HIP is designed as an end-to-end authentication and key establishment protocol, to be used with Encapsulated Security Payload (ESP) [RFC5202] and other end-to-end security protocols. The base protocol does not cover all the fine-grained policy control found in Internet Key Exchange (IKE) [RFC4306] that allows IKE to support complex gateway policies. Thus, HIP is not a replacement for IKE.

1.3. Memo Structure

The rest of this memo is structured as follows. Section 2 defines the central keywords, notation, and terms used throughout the rest of the document. Section 3 defines the structure of the Host Identity and its various representations. Section 4 gives an overview of the HIP base exchange protocol. Sections 5 and 6 define the detail packet formats and rules for packet processing. Finally, Sections 7, 8, and 9 discuss policy, security, and IANA considerations, respectively.

2. Terms and Definitions

2.1. Requirements Terminology

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 RFC 2119 [RFC2119].

2.2. Notation

[x] indicates that x is optional.

{x} indicates that x is encrypted.

X(y) indicates that y is a parameter of X.

<x>i indicates that x exists i times.

--> signifies "Initiator to Responder" communication (requests).

<-- signifies "Responder to Initiator" communication (replies).

| signifies concatenation of information-- e.g., X | Y is the concatenation of X with Y.

Ltrunc (H(x), K) denotes the lowest order K bits of the result of the hash function H on the input x.

2.3. Definitions

Unused Association Lifetime (UAL): Implementation-specific time for which, if no packet is sent or received for this time interval, a host MAY begin to tear down an active association.

Maximum Segment Lifetime (MSL): Maximum time that a TCP segment is expected to spend in the network.

Exchange Complete (EC): Time that the host spends at the R2-SENT before it moves to ESTABLISHED state. The time is n * I2 retransmission timeout, where n is about I2_RETRIES_MAX.

ORCHID Generation Algorithm (OGA): Hash algorithm used to generate a Host Identity Tag (HIT) from the Host Identity public key based on the ORCHID specifications [RFC4843-bis]. Currently truncated SHA-1 [FIPS.95-1.1993] and truncated SHA-256 [FIPS.180-2.2002] are defined.

Responder's HIT Hash Algorithm (RHASH): Hash algorithm used for various hash calculations in this document. The algorithm is the same as is used to generate the Responder's HIT. The RHASH is defined by the ORCHID Generation Algorithm represented by the respective index bits in the Responder's HIT.

Opportunistic mode: HIP base exchange where the Responder's HIT is not known a priori to the Initiator.

3. Host Identifier (HI) and Its Representations

In this section, the properties of the Host Identifier and Host Identifier Tag are discussed, and the exact format for them is defined. In HIP, the 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 [rfc4423-bis] for more details about the difference between an identity and the corresponding identifier.

HIP implementations MUST support the Rivest Shamir Adelman (RSA/SHA-2) [RFC5702] public key algorithm, and SHOULD support the RSA/SHA-1 [RFC3110] and the Digital Signature Algorithm (DSA) [RFC2536] algorithms; other algorithms MAY be supported.

A hashed encoding 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.

The structure of the HIT is defined in [RFC4843-bis] The HIT consists

of three parts: first, an IANA assigned prefix to distinguish it from other IPv6 addresses. Second, a four-bit encoding of the algorithm that was used for generating the hashed representation of the public key. Third, a 96-bit hashed representation of the Host Identity. The encoding of the ORCHID generation algorithm and the exact algorithm for generating the hashed representation is specified in Appendix F.

Carrying HIs and HITs in the header of user data packets would increase the overhead of packets. Thus, it is not expected that they are carried in every packet, but other methods are used to map the data packets to the corresponding HIs. In some cases, this makes it possible to use HIP without any additional headers in the user data packets. For example, if ESP is used to protect data traffic, the Security Parameter Index (SPI) carried in the ESP header can be used to map the encrypted data packet to the correct HIP association.

3.1. Host Identity Tag (HIT)

The Host Identity Tag is a 128-bit value -- a hashed encoding of the Host Identifier. There are two advantages of using a hashed encoding over the actual Host Identity public key 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.

RFC 4843-bis [RFC4843-bis] specifies 128-bit hash-based identifiers, called Overlay Routable Cryptographic Hash Identifiers (ORCHIDs). Their prefix, allocated from the IPv6 address block, is defined in [RFC4843-bis]. The Host Identity Tag is a type of ORCHID.

3.2. Generating a HIT from an HI

The HIT MUST be generated according to the ORCHID generation method described in [RFC4843-bis] using a context ID value of 0xF0EF F02F BFF4 3D0F E793 0C3C 6E61 74EA (this tag value has been generated randomly by the editor of this specification), and an input that encodes the Host Identity field (see Section 5.2.8) present in a HIP payload packet. The hash function and the algorithm used for generating the HIT from the HI depends on the ORCHID Generation Algorithm (OGA) and is indicated by the four bits of the OGA field in the ORCHID. Currently, truncated SHA-1 and SHA-256, as defined below, are the only available OGAs. If a new OGA is needed in the future for HIT generation, a new OGA identifier has to be specified in supplementary documents.

For Identities that are either RSA or Digital Signature Algorithm

(DSA) public keys, this input consists of the public key encoding as specified in the corresponding DNSSEC document, taking the algorithm-specific portion of the RDATA part of the KEY RR. There are currently only two defined public key algorithms: RSA/SHA-1 and DSA. Hence, either of the following applies:

The RSA public key is encoded as defined in [RFC3110] 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 and the preceding HI fields including the exponent (e). Thus, the data taht serves as input for the OGA has the same length as the HI. The fields MUST be encoded in network byte order, as defined in [RFC3110].

The DSA public key is encoded as defined in [RFC2536] 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]. 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].

In Appendix B, the public key encoding process is illustrated using pseudo-code.

4. Protocol Overview

The following material is an overview of the HIP protocol operation, and does not contain all details of the packet formats or the packet processing steps. Sections 5 and 6 describe in more detail the packet formats and packet processing steps, respectively, and are normative in case of any conflicts with this section.

The protocol number 139 has been assigned by IANA to the Host Identity Protocol.

The HIP payload (Section 5.1) header could be carried in every IP datagram. However, since HIP headers are relatively large (40 bytes), it is desirable to 'compress' the HIP header so that the HIP header only occurs in control packets used to establish or change HIP association state. The actual method for header 'compression' and for matching data packets with existing HIP associations (if any) is defined in separate documents, describing transport formats and methods. All HIP implementations MUST implement, at minimum, the ESP transport format for HIP [RFC5202].

4.1. Creating a HIP Association

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.

The HIP base exchange serves to manage the establishment of state between an Initiator and a Responder. The first packet, I1, initiates the exchange, and the last three packets, R1, I2, and R2, constitute an authenticated Diffie-Hellman [DIF76] 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.

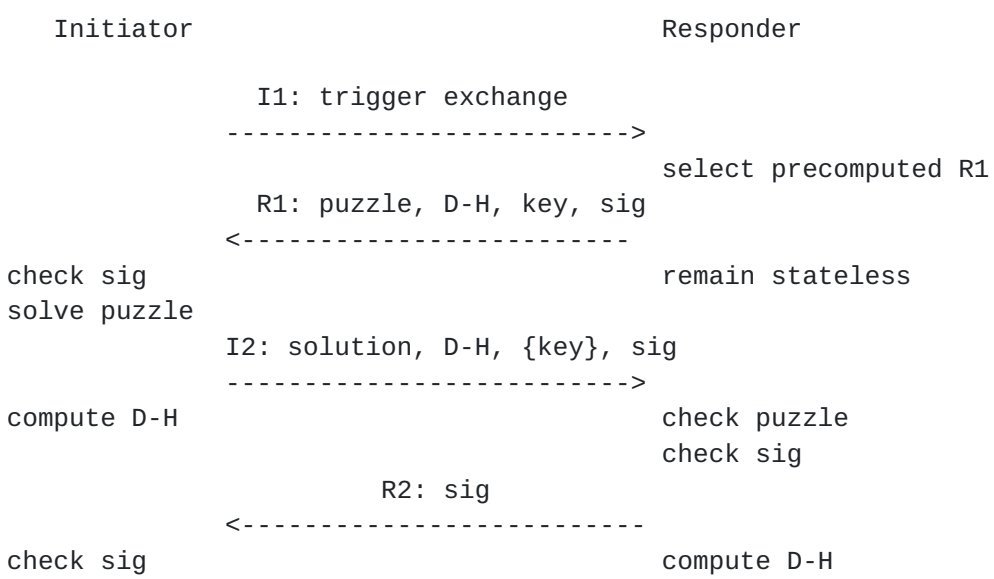
The Initiator first sends a trigger packet, I1, to the Responder. The packet contains only the HIT of the Initiator and possibly the HIT of the Responder, if it is known. Note that in some cases it may be possible to replace this trigger packet by some other form of a trigger, in which case the protocol starts with the Responder sending the R1 packet.

The second packet, R1, starts the actual exchange. It contains a puzzle -- a cryptographic challenge that the Initiator must solve before continuing the exchange. The level of difficulty of the puzzle can be adjusted based on level of trust with the Initiator, current load, or other factors. In addition, the R1 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. The R1 also contains lists of cryptographic algorithms supported by the Responder. Based on these lists, the Initiator can continue, abort, or restart the base exchange with a different selection of cryptographic algorithms.

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. The term "key" refers to the Host Identity public key, and "sig" represents a signature using such a key. The packets contain other parameters not shown in this figure.



4.1.1. HIP Puzzle Mechanism

The purpose of the HIP puzzle 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 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 puzzle 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 correctly formatted I2 can be rejected 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.

The Responder can remain stateless and drop most spoofed I2s because puzzle calculation is based on the Initiator's Host Identity Tag. 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 can verify that the puzzle has been solved using the Initiator's HIT. This makes 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 HITs. The method does not protect from an attacker that uses fixed HITs, though. Against such an attacker a viable approach may be to create a piece of local state, and remember that the puzzle check has previously failed. See Appendix A for one possible implementation. Implementations SHOULD include sufficient randomness to the algorithm so that algorithmic complexity attacks become impossible [CRO03].

The Responder can set the puzzle difficulty for Initiator, based on its level of trust of the Initiator. Because the puzzle is not included in the signature calculation, the Responder can use pre-calculated R1 packets and include the puzzle just before sending the R1 to 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.

4.1.2. Puzzle Exchange

The Responder starts the puzzle 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 hash over this concatenation using the RHASH algorithm. 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 zeros. The Initiator SHOULD give up after exceeding the puzzle lifetime in the PUZZLE parameter (Section 5.2.4). 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 precomputation 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 A for an example on how to implement this.

Using the Opaque data field in an ECHO_REQUEST_SIGNED (Section 5.2.18) or in an ECHO_REQUEST_UNSIGNED parameter (Section 5.2.19), 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 some secret, the sent I, and possibly other related data. Using the same secret, the received I (from the I2), and the other related data (if any), the Receiver can verify that it has itself sent the I to the Initiator. The Responder MUST periodically change such a used

secret.

It is RECOMMENDED that the Responder generates a new puzzle and a new R1 once every few minutes. Furthermore, it is RECOMMENDED that the Responder remembers an old puzzle at least 2*Lifetime seconds after the puzzle has been deprecated. These time values allow a slower Initiator to solve the puzzle while limiting the usability that an old, solved puzzle 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 to NOT 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.

4.1.3. Authenticated Diffie-Hellman Protocol

The packets R1, I2, and R2 implement a standard authenticated Diffie-Hellman exchange. The Responder sends one or two public Diffie-Hellman keys 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 gets one or two public Diffie-Hellman values from the Responder. If there are two values, it selects the value corresponding to the strongest supported Group ID and computes the Diffie-Hellman session key (Kij). It creates a HIP association using keying material from the session key (see Section 6.5), and may use the association to encrypt its public authentication key, i.e., Host Identity. The resulting I2 contains the Initiator's Diffie-Hellman key and its (optionally 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.

4.1.4. 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 puzzle 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 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. The R1 counter SHOULD NOT roll over.

A host may receive more than one R1, either due to sending multiple I1s (Section 6.6.1) or due to a replay of an old R1. When sending multiple I1s, an Initiator SHOULD wait for a small amount of time (a reasonable time may be 2 * expected RTT) 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.

4.1.5. 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's 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.

After receiving the R1 packet, the Initiator can determine whether the Responder supports the required cryptographic operations to successfully establish a HIP association. The choice of which kind of source HIT (determined by the ORCHID generation algorithm) is used by the Initiator is made by the Initiator before sending the I1 packet. If the Initiator determines that the Responder does not support the chosen HIT type, the Initiator MAY restart the base exchange with a source HIT that is supported by the Responder. The Initator MAY abort the base exchange without noticing the Responder if the Responder does not support a suitable HIT type.

4.1.6. HIP Opportunistic Mode

It is possible to initiate a HIP negotiation even if the Responder's HI (and HIT) is unknown. In this case, the connection initializing I1 packet contains NULL (all zeros) as the destination HIT. This kind of connection setup is called opportunistic mode.

There are both security and API issues involved with the opportunistic mode.

Given that the Responder's HI is not known by the Initiator, there must be suitable API calls that allow the Initiator to request, directly or indirectly, that the underlying kernel initiate the HIP base exchange solely based on locators. The Responder's HI will be tentatively available in the R1 packet, and in an authenticated form once the R2 packet has been received and verified. Hence, it could be communicated to the application via new API mechanisms. However, with a backwards-compatible API the application sees only the locators used for the initial contact. Depending on the desired semantics of the API, this can raise the following issues:

- The actual locators may later change if an UPDATE message is used, even if from the API perspective the session still appears to be between specific locators. The locator update is still secure, however, and the session is still between the same nodes.

- Different sessions between the same locators may result in connections to different nodes, if the implementation no longer remembers which identifier the peer had in another session. This is possible when the peer's locator has changed for legitimate reasons or when an attacker pretends to be a node that has the peer's locator. Therefore, when using opportunistic mode, HIP MUST NOT place any expectation that the peer's HI returned in the R1 message matches any HI previously seen from that address.

  If the HIP implementation and application do not have the same understanding of what constitutes a session, this may even happen within the same session. For instance, an implementation may not know when HIP state can be purged for UDP-based applications.

- As with all HIP exchanges, the handling of locator-based or interface-based policy is unclear for opportunistic mode HIP. An application may make a connection to a specific locator because the application has knowledge of the security properties along the network to that locator. If one of the nodes moves and the locators are updated, these security properties may not be maintained. Depending on the security policy of the application, this may be a problem. This is an area of ongoing study. As an example, there is work to create an API that applications can use to specify their security requirements in a similar context [btns-c-api].

In addition, the following security considerations apply. The generation counter mechanism will be less efficient in protecting against replays of the R1 packet, given that the Responder can choose a replay that uses any HI, not just the one given in the I1 packet .

More importantly, the opportunistic exchange is vulnerable to man-in-the-middle attacks, because the Initiator does not have any public key information about the peer. To assess the impacts of this vulnerability, we compare it to vulnerabilities in current, non-HIP-capable communications.

An attacker on the path between the two peers can insert itself as a man-in-the-middle by providing its own identifier to the Initiator and then initiating another HIP session towards the Responder. For this to be possible, the Initiator must employ opportunistic mode, and the Responder must be configured to accept a connection from any HIP-enabled node.

An attacker outside the path will be unable to do so, given that it cannot respond to the messages in the base exchange.

These properties are characteristic also of communications in the current Internet. A client contacting a server without employing end-to-end security may find itself talking to the server via a man-in-the-middle, assuming again that the server is willing to talk to anyone.

If end-to-end security is in place, then the worst that can happen in both the opportunistic HIP and normal IP cases is denial-of-service; an entity on the path can disrupt communications, but will be unable to insert itself as a man-in-the-middle.

However, once the opportunistic exchange has successfully completed, HIP provides integrity protection and confidentiality for the communications, and can securely change the locators of the endpoints.

As a result, it is believed that the HIP opportunistic mode is at least as secure as current IP.

4.2. 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 parameters. 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 under certain circumstances.

- UPDATE is protected by both HMAC and HIP_SIGNATURE parameters, since processing UPDATE signatures alone is a potential DoS attack against intermediate systems.

- UPDATE packets are explicitly acknowledged by the use of an acknowledgment parameter that echoes an individual sequence number received from the peer. A single UPDATE packet may contain both a

sequence number and one or more acknowledgment numbers (i.e., piggybacked acknowledgment(s) for the peer's UPDATE).

The UPDATE packet is defined in Section 5.3.5.

4.3. Error Processing

HIP error processing behavior depends on whether or not there exists an active HIP association. In general, if a 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 5.4 for more details.

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 a 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 type Parameter Problem, to inform the sender that the HIP association does not exist (see Section 5.4), and it MAY initiate a new HIP negotiation. However, responding with these optional mechanisms is implementation or policy dependent.

4.4. HIP State Machine

The HIP protocol itself has little state. In the HIP base exchange, there is an Initiator and a Responder. Once the security associations (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 6 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 specifications (such as mobility and multihoming).

4.4.1. HIP States

State	Explanation
UNASSOCIATED	State machine start
I1-SENT	Initiating base exchange
I2-SENT	Waiting to complete base exchange
R2-SENT	Waiting to complete base exchange
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

Table 1: HIP States

4.4.2. HIP State Processes

System behavior in state UNASSOCIATED, Table 2.

Trigger	Action
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 5.4 and stay at UNASSOCIATED
Receive CLOSE	Optionally send ICMP Parameter Problem and stay at UNASSOCIATED
Receive ANYOTHER	Drop and stay at UNASSOCIATED

Table 2: UNASSOCIATED - Start state

System behavior in state I1-SENT, Table 3.

Trigger	Action
Receive I1	If the local HIT is smaller than the peer HIT, drop I1 and stay at I1-SENT If the local HIT is greater than the peer HIT, 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 OGA of own HIT is not supported by peer, select supported own HIT, send I1 stay at I1-SENT If successful, send I2 and go to I2-SENT If fail, stay at I1-SENT
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

Table 3: I1-SENT - Initiating HIP

System behavior in state I2-SENT, Table 4.

Trigger	Action
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 and local HIT is smaller than the peer HIT, drop I2 and stay at I2-SENT
	If successful and local HIT is greater than the peer HIT, send R2 and go to R2-SENT
	If fail, stay at I2-SENT
Receive R2, process	If successful, go to ESTABLISHED
	If fail, stay at I2-SENT
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

Table 4: I2-SENT - Waiting to finish HIP

System behavior in state R2-SENT, Table 5.

Trigger	Action
Receive I1	Send R1 and stay at R2-SENT
Receive I2, process	If successful, send R2 and cycle at R2-SENT
	If fail, stay at R2-SENT
Receive R1	Drop and stay at R2-SENT
Receive R2	Drop and stay at R2-SENT
Receive data or UPDATE	Move to ESTABLISHED
Exchange Complete Timeout	Move to ESTABLISHED

Table 5: R2-SENT - Waiting to finish HIP

System behavior in state ESTABLISHED, Table 6.

Trigger	Action
Receive I1	Send R1 and stay at ESTABLISHED
Receive I2, process with puzzle and possible Opaque data verification	If successful, send R2, drop old HIP association, establish a new HIP association, go 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 fail, stay at ESTABLISHED

Table 6: ESTABLISHED - HIP association established

System behavior in state CLOSING, Table 7.

Trigger	Action
User data to send, requires the creation of another incarnation of the HIP association	Send 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 fail, stay at CLOSING
Receive CLOSE_ACK, process	If successful, discard state and go to UNASSOCIATED
	If fail, 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

Table 7: CLOSING - HIP association has not been used for UAL minutes

System behavior in state CLOSED, Table 8.

Trigger	Action
Datagram to send, requires the creation of another incarnation of the HIP association	Send 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 fail, stay at CLOSED
Receive CLOSE_ACK, process	If successful, discard state and go to UNASSOCIATED
	If fail, stay at CLOSED
Receive ANYOTHER	Drop and stay at CLOSED
Timeout (UAL+2MSL)	Discard state, and go to UNASSOCIATED

Table 8: CLOSED - CLOSE_ACK sent, resending CLOSE_ACK if necessary

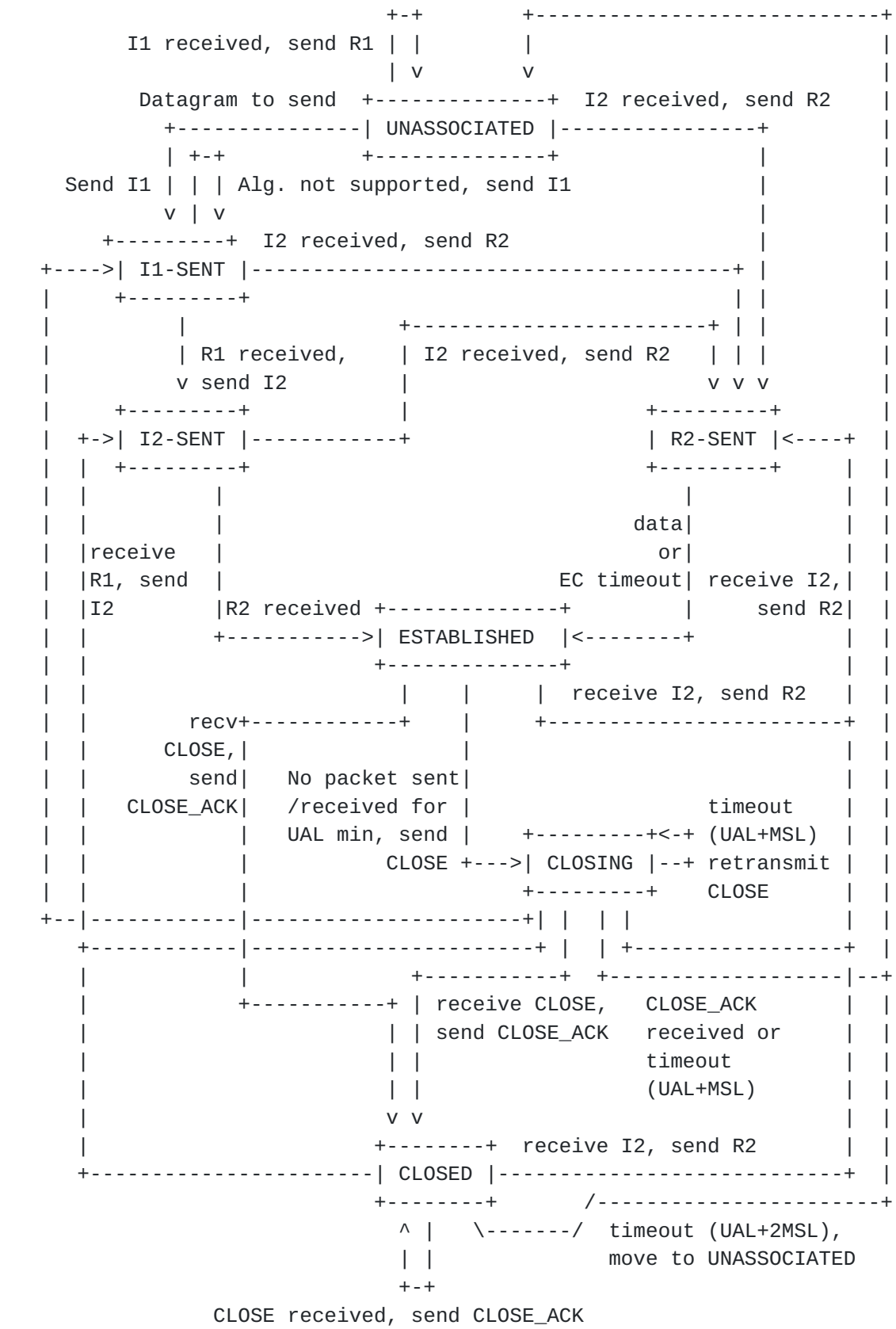
System behavior in state E-FAILED, Table 9.

Trigger	Action
Wait for implementation-specific time	Go to UNASSOCIATED. Re-negotiation is possible after moving to UNASSOCIATED state.

Table 9: E-FAILED - HIP failed to establish association with peer

4.4.3. 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.


I1 received, send R1
Datagram to send
UNASSOCIATED
I2 received, send R2
Send I1
Alg. not supported, send I1
I2 received, send R2
I1-SENT
R1 received, send I2
I2 received, send R2
I2-SENT
R2-SENT
receive R1, send I2
R2 received
ESTABLISHED
data or EC timeout
receive I2, send R2
recv CLOSE, send CLOSE_ACK
No packet sent /received for UAL min, send CLOSE
CLOSING
timeout (UAL+MSL) retransmit CLOSE
receive CLOSE, send CLOSE_ACK
CLOSE_ACK received or timeout (UAL+MSL)
CLOSED
receive I2, send R2
timeout (UAL+2MSL), move to UNASSOCIATED
CLOSE received, send CLOSE_ACK

4.5. User Data Considerations

4.5.1. 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, the IPv6 pseudo-header format [RFC2460] MUST be used, even if the actual addresses on the packet are IPv4 addresses. 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 5.1.1.

4.5.2. 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.

4.5.3. 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 [RFC5202].

When new transport formats are defined, they get the type value from the HIP Transform type value space 2048-4095. 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.

4.5.4. 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 the HIP association times out, it has lost its HIP state. If the host that lost state has a datagram to send to the peer, it simply restarts the HIP base exchange. After the base exchange has completed, the Initiator can create a new SA and start sending data. The peer does not reset its state until it receives a valid I2 HIP packet.

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, and with 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.

If the host, that apparently has lost its state, decides to restart the HIP base exchange, it sends an I1 packet to the peer. After the base exchange has been completed successfully, the Initiator can create a new HIP association and the peer drops its old SA and creates a new one.

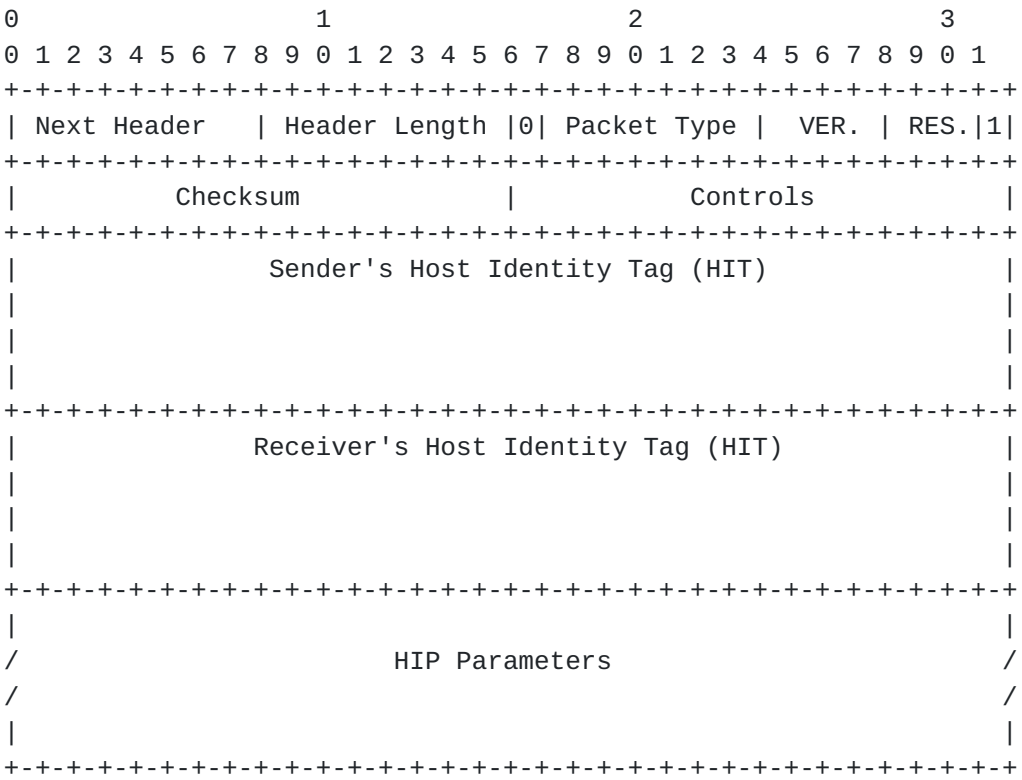
4.6. Certificate Distribution

This document does not define how to use certificates or how to transfer them between hosts. These functions are expected to be defined in a future specification. A parameter type value, meant to be used for carrying certificates, is reserved, though: CERT, Type 768; see Section 5.2.

5. Packet Formats

5.1. 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, current 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-byte units, excluding the first 8 bytes. Since all HIP headers MUST contain the sender's and receiver's HIT fields, the minimum value for this field is 4, and conversely, the maximum length of the HIP Parameters field is (255*8)-32 = 2008 bytes. Note: this sets an additional limit for sizes of parameters included in the Parameters field, independent of the individual 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 three bits are reserved for future use. They MUST be zero when sent, and they SHOULD be ignored when handling a received packet.

The two fixed bits in the header are reserved for potential SHIM6 compatibility [RFC5533]. For implementations adhering (only) to this specification, they MUST be set as shown when sending and MUST be ignored when receiving. This is to ensure optimal forward compatibility. Note that for implementations that implement other compatible specifications in addition to this specification, the corresponding rules may well be different. For example, in the case that the forthcoming SHIM6 protocol happens to be compatible with this specification, an implementation that implements both this specification and the SHIM6 protocol may need to check these bits in order to determine how to handle the packet.

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

5.1.1. Checksum

Since the checksum covers the source and destination addresses in the IP header, it must be recomputed on HIP-aware NAT devices.

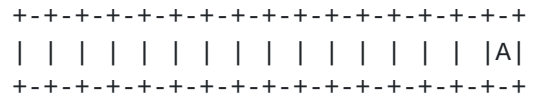
If IPv6 is used to carry the HIP packet, the pseudo-header [RFC2460] contains the source and destination IPv6 addresses, HIP packet length in the pseudo-header length field, a zero field, and the HIP protocol number (see Section 4) in the Next Header field. The length field is in bytes and can be calculated from the HIP header length field: (HIP Header Length + 1) * 8.

In case of using IPv4, the IPv4 UDP pseudo-header format [RFC0768] 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 (see Section 4), and the length is calculated as in the IPv6 case.

5.1.2. HIP Controls

The HIP Controls section conveys information about the structure of the packet and capabilities of the host.

The following fields have been defined:



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.

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

5.1.3. HIP Fragmentation Support

A HIP implementation must support IP fragmentation/reassembly. Fragment reassembly MUST be implemented in both IPv4 and IPv6, but fragment generation is REQUIRED to be implemented in IPv4 (IPv4 stacks and networks will usually do this by default) and RECOMMENDED to be implemented in IPv6. In IPv6 networks, the minimum MTU is larger, 1280 bytes, than in IPv4 networks. 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 IPv4 networks, 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 for path MTU discovery

does not bring any value to HIP in IPv4 networks. HIP-aware NAT devices MUST perform any IPv4 reassembly/fragmentation.

All HIP implementations have to be careful while employing a reassembly algorithm so that the algorithm is sufficiently resistant to DoS attacks.

Because certificate chains can cause the packet to be fragmented and fragmentation can open implementation to denial-of-service attacks [KAU03], it is strongly recommended that the separate document specifying the certificate usage in the HIP Base Exchange defines the usage of "Hash and URL" formats rather than including certificates in exchanges. With this, most problems related to DoS attacks with fragmentation can be avoided.

5.2. HIP Parameters

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

The following parameter types are currently defined.

TLV	Type	Length	Data
R1_COUNTER	128	12	System Boot Counter
PUZZLE	257	12	K and Random #I
SOLUTION	321	20	K, Random #I and puzzle solution J
SEQ	385	4	Update packet ID number
ACK	449	variable	Update packet ID number
DIFFIE_HELLMAN	513	variable	public key
HIP_TRANSFORM	577	variable	HIP Encryption and Integrity Transform
ENCRYPTED	641	variable	Encrypted part of I2 packet
HOST_ID	705	variable	Host Identity with Fully-Qualified Domain FQDN (Name) or Network Access Identifier (NAI)
OGA_LIST	715	variable	List of the OGAs supported by the Responder
CERT	768	variable	HI Certificate; used to transfer certificates. Usage is not currently defined, but it will be specified in a separate document once needed.
NOTIFICATION	832	variable	Informational data
ECHO_REQUEST_SIGNED	897	variable	Opaque data to be echoed back; under signature

ECHO_RESPONSE_SIGNED	961	variable	Opaque data echoed back; under signature
HMAC	61505	variable	HMAC-based message authentication code, with key material from HIP_TRANSFORM
HMAC_2	61569	variable	HMAC based message authentication code, with key material from HIP_TRANSFORM. Compared to HMAC, the HOST_ID parameter is included in HMAC_2 calculation.
HIP_SIGNATURE_2	61633	variable	Signature of the R1 packet
HIP_SIGNATURE	61697	variable	Signature of the packet
ECHO_REQUEST_UNSIGNED	63661	variable	Opaque data to be echoed back; after signature
ECHO_RESPONSE_UNSIGNED	63425	variable	Opaque data echoed back; after signature

Because the ordering (from lowest to highest) of HIP parameters is strictly enforced (see Section 5.2.1), the parameter type values for existing parameters have been spaced to allow for future protocol extensions. Parameters numbered between 0-1023 are used in HIP handshake and update procedures and are covered by signatures. Parameters numbered between 1024-2047 are reserved. Parameters numbered between 2048-4095 are used for parameters related to HIP transform types. Parameters numbered between 4096 and (2^16 - 2^12) 61439 are reserved. Parameters numbered between 61440-62463 are used for signatures and signed MACs. Parameters numbered between 62464-63487 are used for parameters that fall outside of the signed area of the packet. Parameters numbered between 63488-64511 are used for rendezvous and other relaying services. Parameters numbered between 64512-65535 are reserved.

5.2.1. 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 in the packet such that their types form an increasing order. If the parameter can exist multiple times in the packet, the type value may be the same in consecutive parameters. If the order does not follow this rule, the packet is considered to be malformed and it MUST be discarded.

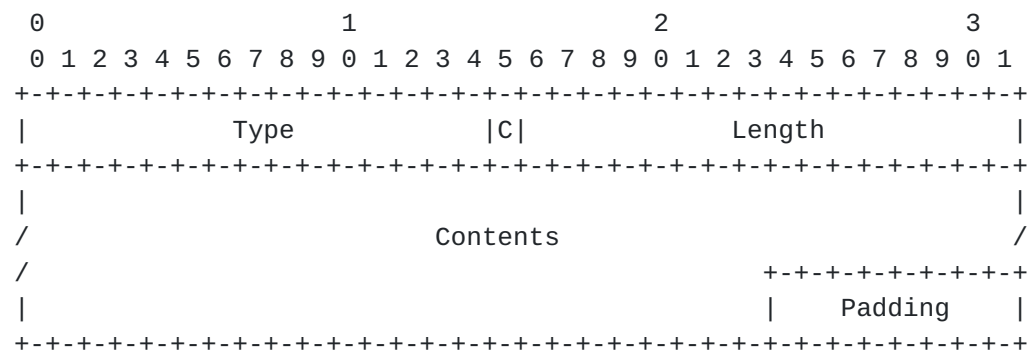
Parameters using type values from 2048 up to 4095 are transport formats. Currently, one transport format is defined: the ESP transport format [RFC5202]. The order of these parameters does not follow the order of their type value, but they are put in the packet in order of preference. The first of the transport formats it the most preferred, and so on.

All of the TLV parameters have a length (including Type and Length fields), which is a multiple of 8 bytes. When needed, padding MUST be added to the end of the parameter so that the total length becomes a multiple of 8 bytes. This rule ensures proper alignment of data. Any added padding bytes MUST be zeroed by the sender, and their values SHOULD NOT be checked by the receiver.

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

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

where % is the modulo operator



Type	Type code for the parameter. 16 bits long, C-bit being part of the Type code.
C	Critical. One if this parameter is critical, and MUST be recognized by the recipient, zero otherwise. The C bit is considered to be a part of the Type field. Consequently, critical parameters are always odd and non-critical ones have an even value.
Length	Length of the Contents, in bytes.
Contents	Parameter specific, defined by Type
Padding	Padding, 0-7 bytes, added if needed

Critical parameters MUST be recognized by the recipient. If a recipient encounters a critical parameter that it does not recognize, it MUST NOT process the packet any further. It MAY send an ICMP or NOTIFY, as defined in Section 4.3.

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

5.2.2. 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 increasing order by Type code, thereby limiting the order of parameters (see Section 5.2.1).

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.

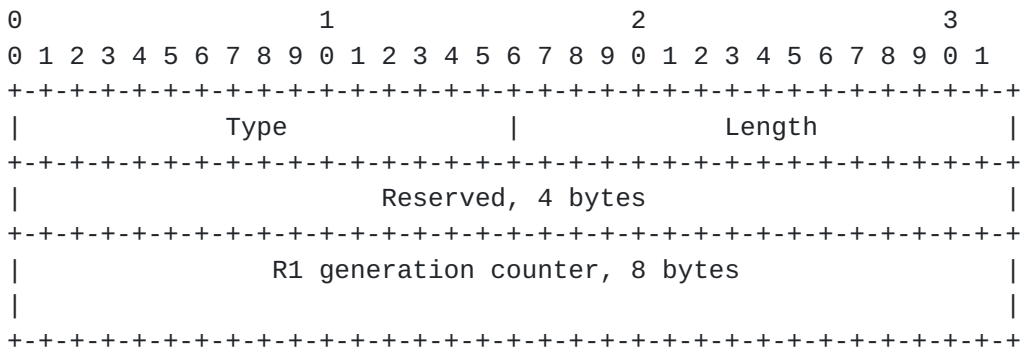
2. A new parameter may be critical only if an old recipient ignoring it would cause security problems. In general, new parameters

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 set the associated feature off, such that the critical parameter is not sent at all. The configuration option must be well documented. Implementations operating in a mode adhering to this specification MUST disable the sending of new critical parameters. 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. See Section 9 for allocation rules regarding Type codes.

5.2.3. R1_COUNTER

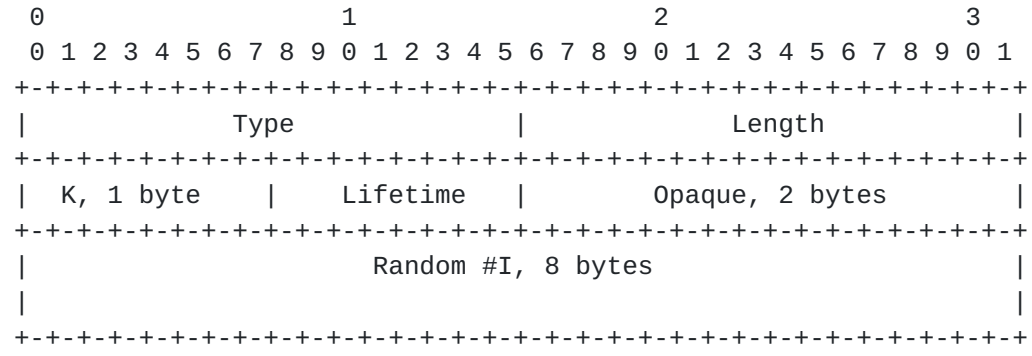


Type	128
Length	12
R1 generation counter	The current generation of valid puzzles

The R1_COUNTER parameter contains a 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 verbatim) by the Initiator in the I2.

5.2.4. PUZZLE



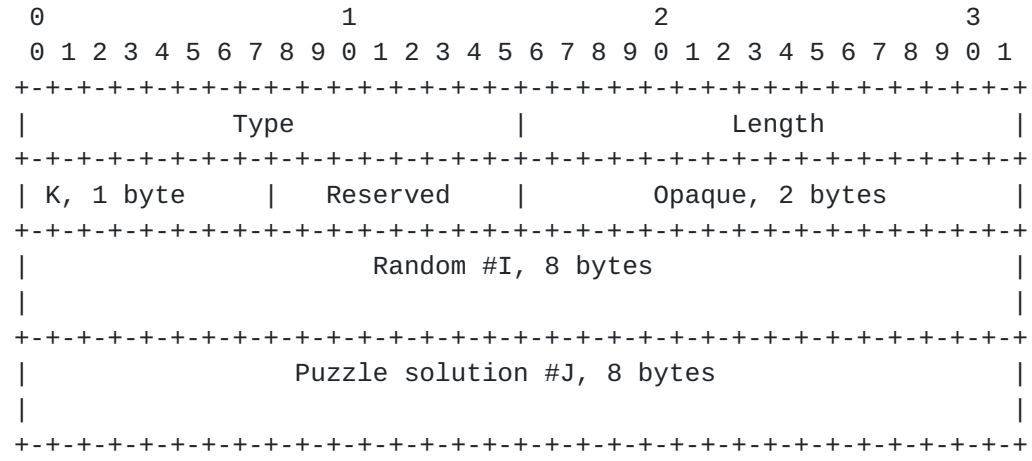
Type	257
Length	12
K	K is the number of verified bits
Lifetime	puzzle lifetime 2^(value-32) seconds
Opaque	data set by the Responder, indexing the puzzle
Random #I	random number

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

The PUZZLE parameter contains the puzzle difficulty K and a 64-bit puzzle random integer #I. The Puzzle Lifetime indicates the time during which the puzzle solution is valid, and sets a time limit that should not be exceeded by the Initiator while it attempts to solve the puzzle. The lifetime is indicated as a power of 2 using the formula 2^(Lifetime-32) seconds. A puzzle MAY be augmented with an ECHO_REQUEST_SIGNED or an ECHO_REQUEST_UNSIGNED parameter included in the R1; the contents of the echo request are then echoed back in the ECHO_RESPONSE_SIGNED or in the ECHO_RESPONSE_UNSIGNED, allowing the Responder to use the included information as a part of its puzzle processing.

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

5.2.5. SOLUTION

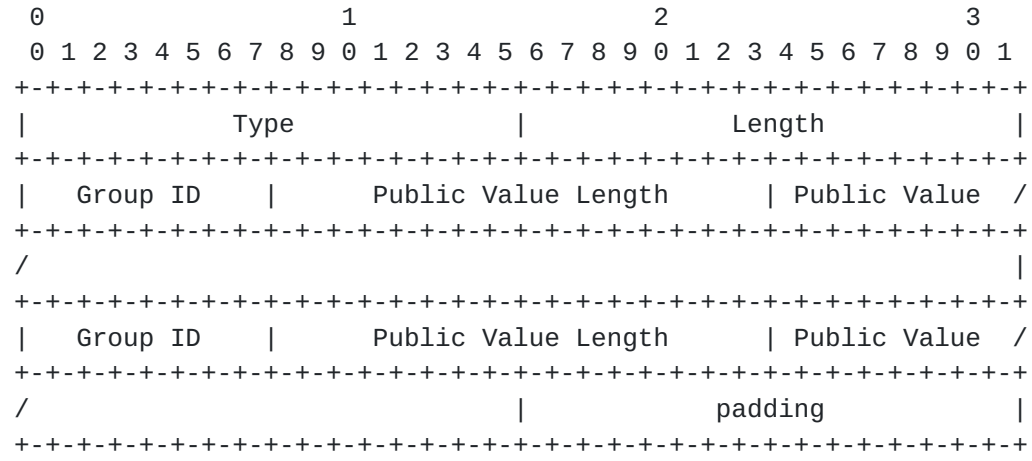


Type	321
Length	20
K	K is the number of verified bits
Reserved	zero when sent, ignored when received
Opaque	copied unmodified from the received PUZZLE parameter
Random #I	random number
Puzzle solution #J	random number

Random #I and Random #J are represented as 64-bit integers, K as an 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.

5.2.6. DIFFIE_HELLMAN



Type	513
Length	length in octets, excluding Type, Length, and padding
Group ID	defines values for p and g
Public Value Length	length of the following Public Value in octets
Public Value	the sender's public Diffie-Hellman key

The following Group IDs have been defined:

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

The MODP Diffie-Hellman groups are defined in [RFC3526]. The OAKLEY well-known group 1 is defined in Appendix E.

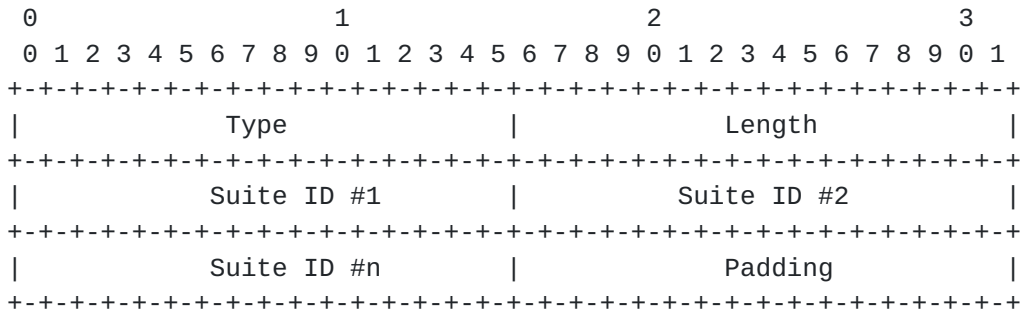
The sender can include at most two different Diffie-Hellman public values in the DIFFIE_HELLMAN parameter. This gives the possibility, e.g., for a server to provide a weaker encryption possibility for a PDA host that is not powerful enough. It is RECOMMENDED that the Initiator, receiving more than one public value, selects the stronger one, if it supports it.

A HIP implementation MUST implement 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). It

is REQUIRED that the default configuration allows Group ID 1 usage, but it is RECOMMENDED that applications that need stronger security turn Group ID 1 support off. Equipment powerful enough SHOULD implement also Group ID 5. The 384-bit group is defined in Appendix D.

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

5.2.7. HIP_TRANSFORM



Type 577
Length length in octets, excluding Type, Length, and padding
Suite ID defines the HIP Suite to be used

The following Suite IDs are defined ([RFC4307],[RFC2451]):

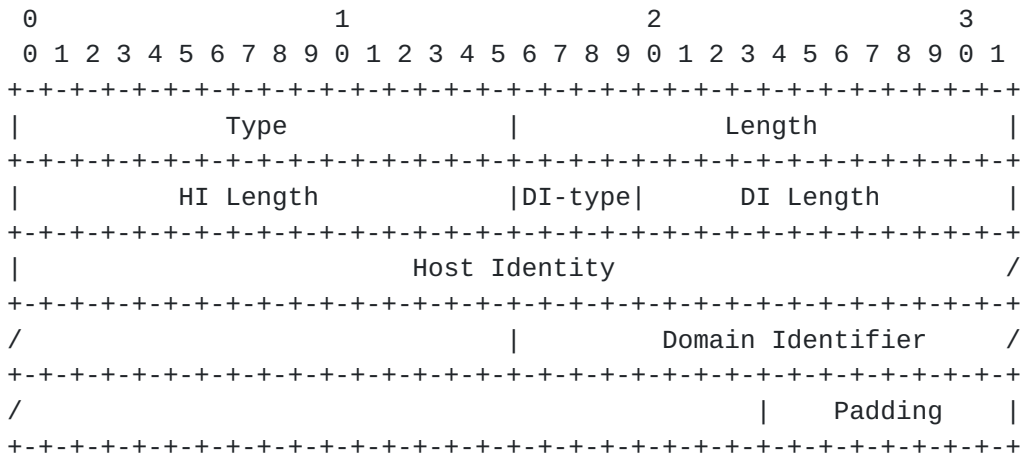
Suite ID	Value
RESERVED	0
AES-CBC with HMAC-SHA-1	1
3DES-CBC with HMAC-SHA-1	2
3DES-CBC with HMAC-MD5	3
BLOWFISH-CBC with HMAC-SHA-1	4
NULL-ENCRYPT with HMAC-SHA-1	5
NULL-ENCRYPT with HMAC-MD5	6

The sender of a HIP_TRANSFORM parameter MUST make sure that there are no more than six (6) HIP Suite IDs in one HIP_TRANSFORM parameter. Conversely, a recipient MUST be prepared to handle received transport parameters that contain more than six Suite IDs by accepting the first six Suite IDs and dropping the rest. The limited number of transforms sets the maximum size of HIP_TRANSFORM parameter. As the default configuration, the HIP_TRANSFORM parameter MUST contain at least one of the mandatory Suite IDs. There MAY be a configuration option that allows the administrator to override this default.

The Responder lists supported and desired Suite IDs in order of preference in the R1, up to the maximum of six Suite IDs. The Initiator MUST choose only one of the corresponding Suite IDs. That Suite ID will be used for generating the I2.

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

5.2.8. HOST_ID



Type	705
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 RFC 4034 [RFC4034] format. The algorithms used in RDATA format are the following:

Algorithms	Values
RESERVED	0
DSA	3 [RFC2536] (RECOMMENDED)
RSA/SHA-1	5 [RFC3110] (RECOMMENDED)
RSA/SHA-2	7 [RFC5702] (REQUIRED)

The following DI-types have been defined:

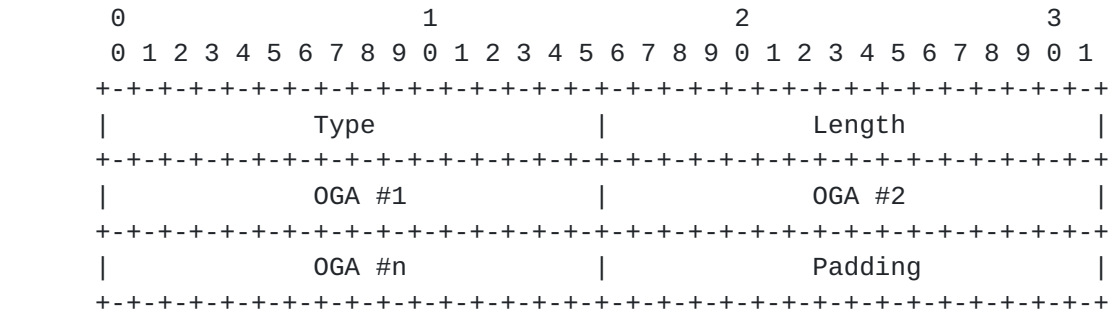
Type	Value
none included	0
FQDN	1
NAI	2

FQDN	Fully Qualified Domain Name, in binary format.
NAI	Network Access Identifier

The format for the FQDN is defined in RFC 1035 [RFC1035] Section 3.1. The format for NAI is defined in [RFC4282]

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

5.2.9. OGA_LIST



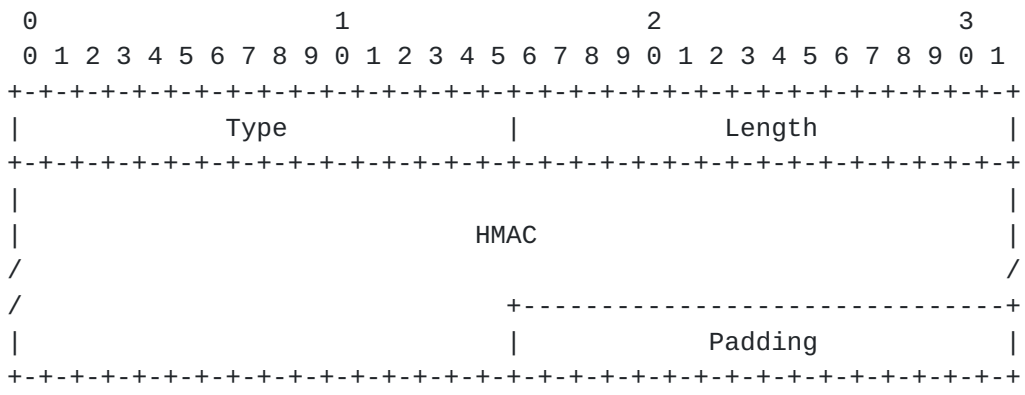
Type	715
Length	length in octets, excluding Type, Length, and padding
OGA	defines the ORCHID generation algorithm available to the host

The following OGAs are defined:

Host Identity Hash	Value
RESERVED	0
SHA-1	1
SHA-256	2

The OGA_LIST parameter contains a list of the supported OGAs of the Responder. The Responder sends the OGA list in the signed part of the R1 packet. Based on the OGA list, the Initiator can determine which OGAs of its HITs are supported by the Responder.

5.2.10. HMAC



Type	61505
Length	length in octets, excluding Type, Length, and Padding
HMAC	HMAC computed over the HIP packet, excluding the HMAC parameter and any following parameters, such as HIP_SIGNATURE, HIP_SIGNATURE_2, ECHO_REQUEST_UNSIGNED, or ECHO_RESPONSE_UNSIGNED. 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 size of the HMAC is the natural size of the hash computation output depending on the used hash function.

The HMAC calculation and verification process is presented in Section 6.4.1.

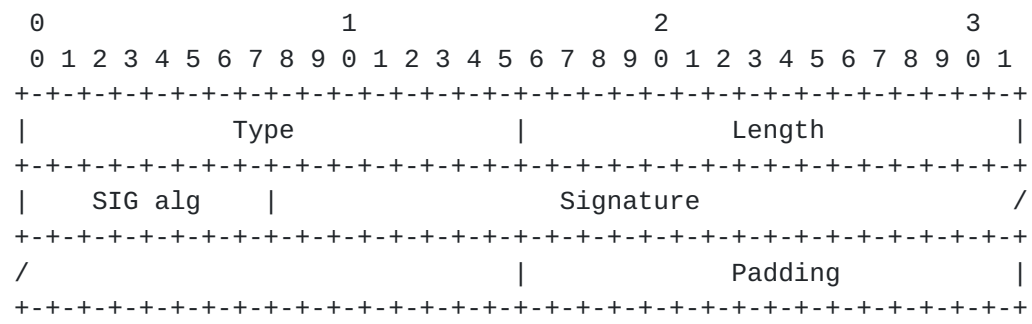
5.2.11. HMAC_2

The parameter structure is the same as in Section 5.2.10. The fields are:

Type	61569
Length	length in octets, excluding Type, Length, and Padding
HMAC	HMAC computed over the HIP packet, excluding the HMAC parameter and any following parameters such as HIP_SIGNATURE, HIP_SIGNATURE_2, ECHO_REQUEST_UNSIGNED, or ECHO_RESPONSE_UNSIGNED, and including an additional sender's HOST_ID parameter 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 size of the HMAC is the natural size of the hash computation output depending on the used hash function.

The HMAC calculation and verification process is presented in Section 6.4.1.

5.2.12. HIP_SIGNATURE



Type	61697
Length	length in octets, excluding Type, Length, and Padding
SIG alg	signature algorithm
Signature	the signature is calculated over the HIP packet, excluding the HIP_SIGNATURE parameter and any parameters that follow the HIP_SIGNATURE parameter. The checksum field MUST be set to zero, and the HIP header length in the HIP common header MUST be calculated only to the beginning of the HIP_SIGNATURE parameter when the signature is calculated.

The signature algorithms are defined in Section 5.2.8. The signature in the Signature field is encoded using the proper method depending on the signature algorithm (e.g., according to [RFC3110] in case of RSA/SHA-1, according to [RFC5702] in case of RSA/SHA-256, or

according to [RFC2536] in case of DSA).

The HIP_SIGNATURE calculation and verification process is presented in Section 6.4.2.

5.2.13. HIP_SIGNATURE_2

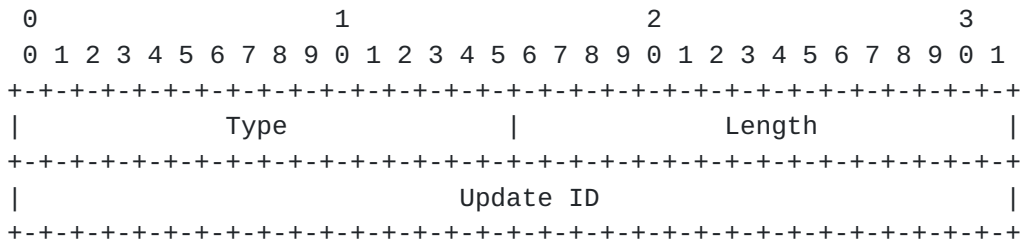
The parameter structure is the same as in Section 5.2.12. The fields are:

Type	61633
Length	length in octets, excluding Type, Length, and Padding
SIG alg	signature algorithm
Signature	Within the R1 packet that contains the HIP_SIGNATURE_2 parameter, the Initiator's HIT, the checksum field, and the Opaque and Random #I fields in the PUZZLE parameter MUST be set to zero while computing the HIP_SIGNATURE_2 signature. Further, the HIP packet length in the HIP header MUST be adjusted as if the HIP_SIGNATURE_2 was not in the packet during the signature calculation, i.e., the HIP packet length points to the beginning of the HIP_SIGNATURE_2 parameter during signing and verification.

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

Signature calculation and verification follows the process in Section 6.4.2.

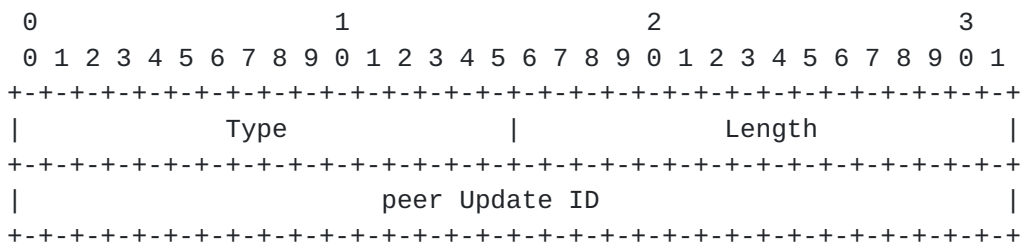
5.2.14. SEQ



Type	385
Length	4
Update ID	32-bit sequence number

The Update ID is an unsigned quantity, initialized by a host to zero upon moving to ESTABLISHED state. The Update ID has scope within a single HIP association, and not across multiple associations or multiple hosts. The Update ID is incremented by one before each new UPDATE that is sent by the host; the first UPDATE packet originated by a host has an Update ID of 0.

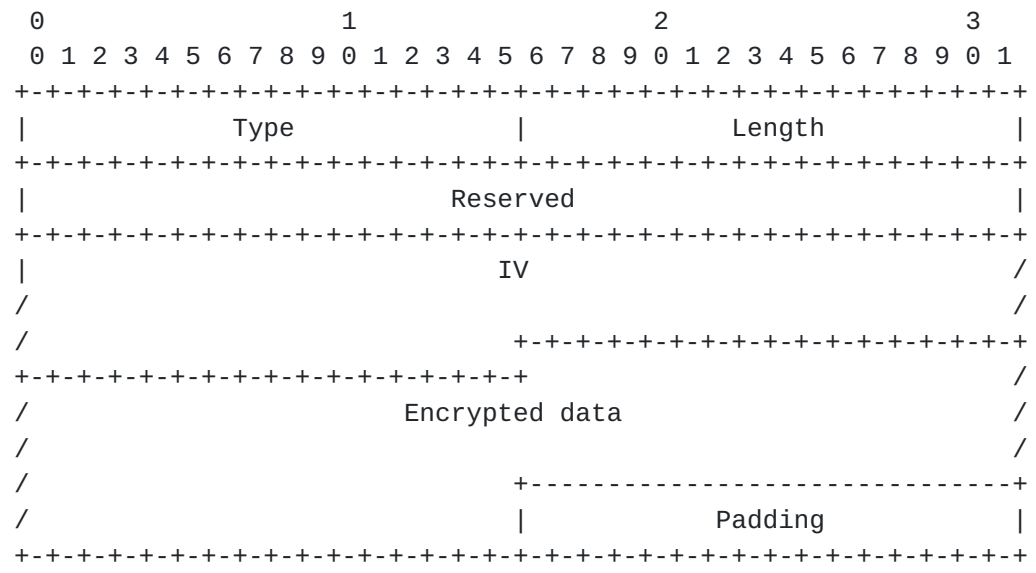
5.2.15. ACK



Type	449
Length	variable (multiple of 4)
peer Update ID	32-bit sequence number corresponding to the Update ID being ACKed.

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

5.2.16. ENCRYPTED



Type	641
Length	length in octets, excluding Type, Length, and Padding
Reserved	zero when sent, ignored when received
IV	Initialization vector, if needed, otherwise nonexistent. The length of the IV is inferred from the HIP transform.
Encrypted data	The data is encrypted using an encryption algorithm as defined in HIP transform.

The ENCRYPTED parameter encapsulates another parameter, the encrypted data, which holds one or more HIP parameters in block encrypted form.

Consequently, the first fields in the encapsulated parameter(s) are Type and Length of the first such parameter, allowing the contents to be easily parsed after decryption.

The field labelled "Encrypted data" consists of the output of one or more HIP parameters concatenated together that have been passed through an encryption algorithm. Each of these inner parameters is padded according to the rules of Section 5.2.1 for padding individual parameters. As a result, the concatenated parameters will be a block of data that is 8-byte aligned.

Some encryption algorithms require that the data to be encrypted must be a multiple of the cipher algorithm block size. In this case, the above block of data MUST include additional padding, as specified by the encryption algorithm. The size of the extra padding is selected so that the length of the unencrypted data block is a multiple of the

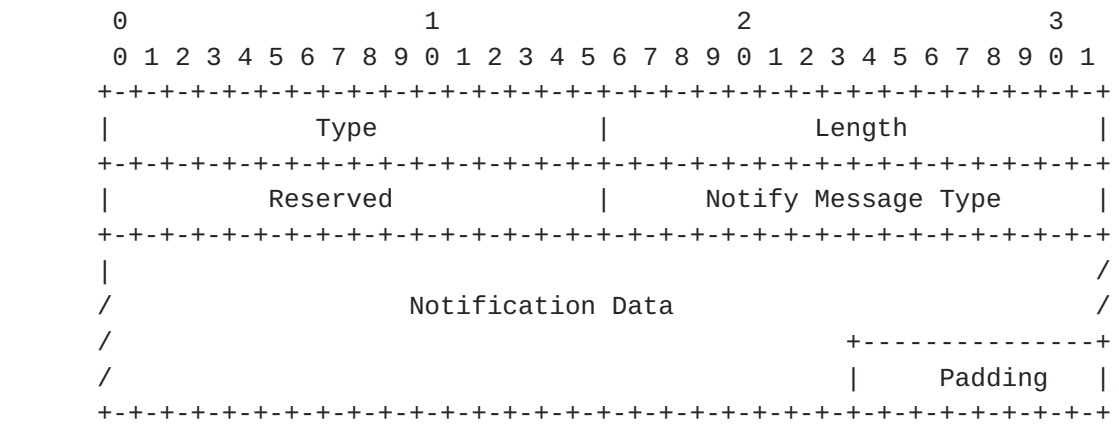
cipher block size. The encryption algorithm may specify padding bytes other than zero; for example, AES [FIPS.197.2001] uses the PKCS5 padding scheme (see section 6.1.1 of [RFC2898]) where the remaining n bytes to fill the block each have the value n. This yields an "unencrypted data" block that is transformed to an "encrypted data" block by the cipher suite. This extra padding added to the set of parameters to satisfy the cipher block alignment rules is not counted in HIP TLV length fields, and this extra padding should be removed by the cipher suite upon decryption.

Note that the length of the cipher suite output may be smaller or larger than the length of the set of parameters to be encrypted, since the encryption process may compress the data or add additional padding to the data.

Once this encryption process is completed, the Encrypted data field is ready for inclusion in the Parameter. If necessary, additional Padding for 8-byte alignment is then added according to the rules of Section 5.2.1.

5.2.17. NOTIFICATION

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



Type	832
Length	length in octets, excluding Type, Length, and Padding
Reserved	zero when sent, ignored when received
Notify Message Type	specifies the type of notification
Notification Data	informational or error data transmitted in addition to the Notify Message Type. Values for this field are type specific (see below).
Padding	any Padding, if necessary, to make the parameter a multiple of 8 bytes.

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

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

Types in the range 0-16383 are intended for reporting errors and in the range 16384-65535 for other status information. An implementation that receives a NOTIFY packet with a NOTIFICATION 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.

NOTIFICATION 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 packets whose HMAC (if present) and SIGNATURE have been verified. This status MUST be sent in response to any error not covered by one of the other status types, and should not contain details to avoid leaking information to someone probing a node. 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, except when the signature verification fails in a NOTIFY message.

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 parameter.

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).

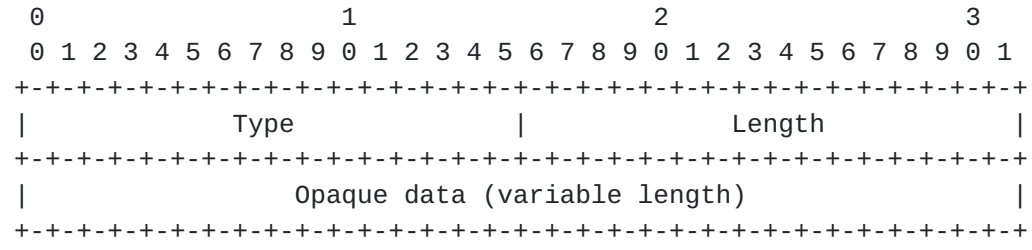
SERVER_BUSY_PLEASE_RETRY 44

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 the Initiator's request. The Initiator may retry; however, the Initiator MUST find another (different) puzzle solution for any such retries. Note that the Initiator may need to obtain a new puzzle with a new I1/R1 exchange.

NOTIFY MESSAGES - STATUS TYPES	Value
I2_ACKNOWLEDGEMENT	16384

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

5.2.18. ECHO_REQUEST_SIGNED

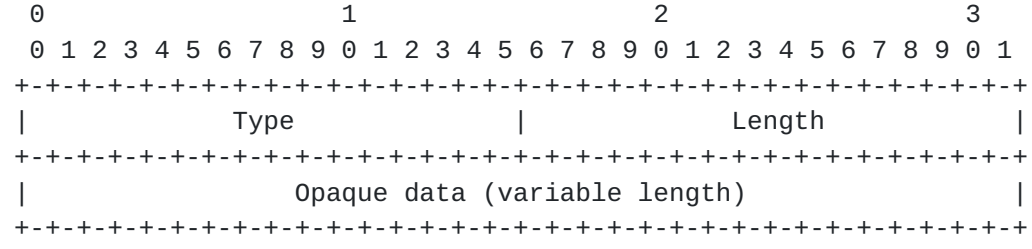


Type	897
Length	variable
Opaque data	opaque data, supposed to be meaningful only to the node that sends ECHO_REQUEST_SIGNED and receives a corresponding ECHO_RESPONSE_SIGNED or ECHO_RESPONSE_UNSIGNED.

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

The ECHO_REQUEST_SIGNED and corresponding echo response parameters MAY be used for any purpose where a node wants to carry some state in a request packet and get it back in a response packet. The ECHO_REQUEST_SIGNED is covered by the HMAC and SIGNATURE. A HIP packet can contain only one ECHO_REQUEST_SIGNED or ECHO_REQUEST_UNSIGNED parameter. The ECHO_REQUEST_SIGNED parameter MUST be responded to with a corresponding echo response. ECHO_RESPONSE_SIGNED SHOULD be used, but if it is not possible, e.g., due to a middlebox-provided response, it MAY be responded to with an ECHO_RESPONSE_UNSIGNED.

5.2.19. ECHO_REQUEST_UNSIGNED



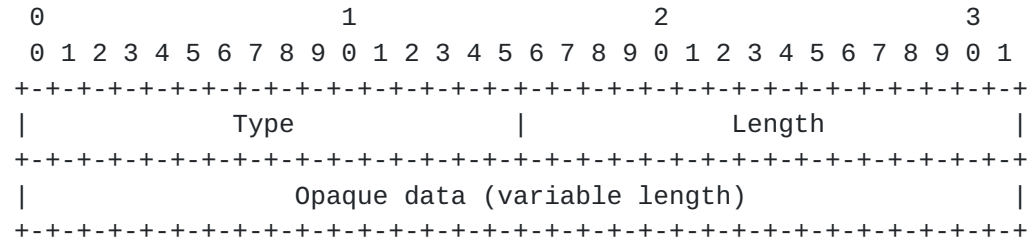
Type	63661
Length	variable
Opaque data	opaque data, supposed to be meaningful only to the node that sends ECHO_REQUEST_UNSIGNED and receives a corresponding ECHO_RESPONSE_UNSIGNED.

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

The ECHO_REQUEST_UNSIGNED and corresponding echo response parameters MAY be used for any purpose where a node wants to carry some state in a request packet and get it back in a response packet. The ECHO_REQUEST_UNSIGNED is not covered by the HMAC and SIGNATURE. A HIP packet can contain one or more ECHO_REQUEST_UNSIGNED parameters. It is possible that middleboxes add ECHO_REQUEST_UNSIGNED parameters in HIP packets passing by. The sender has to create the Opaque field so that it can later identify and remove the corresponding ECHO_RESPONSE_UNSIGNED parameter.

The ECHO_REQUEST_UNSIGNED parameter MUST be responded to with an ECHO_RESPONSE_UNSIGNED parameter.

5.2.20. ECHO_RESPONSE_SIGNED

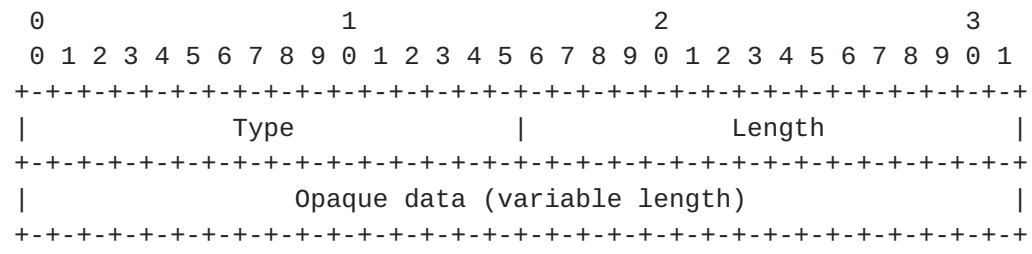


Type	961
Length	variable
Opaque data	opaque data, copied unmodified from the ECHO_REQUEST_SIGNED or ECHO_REQUEST_UNSIGNED parameter that triggered this response.

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

The ECHO_REQUEST_SIGNED and ECHO_RESPONSE_SIGNED parameters MAY be used for any purpose where a node wants to carry some state in a request packet and get it back in a response packet. The ECHO_RESPONSE_SIGNED is covered by the HMAC and SIGNATURE.

5.2.21. ECHO_RESPONSE_UNSIGNED



Type	63425
Length	variable
Opaque data	opaque data, copied unmodified from the ECHO_REQUEST_SIGNED or ECHO_REQUEST_UNSIGNED parameter that triggered this response.

The ECHO_RESPONSE_UNSIGNED parameter contains an opaque blob of data that the sender of the ECHO_REQUEST_SIGNED or ECHO_REQUEST_UNSIGNED wants to get echoed back. The opaque data is copied unmodified from the corresponding echo request parameter.

The echo request and ECHO_RESPONSE_UNSIGNED parameters MAY be used for any purpose where a node wants to carry some state in a request packet and get it back in a response packet. The ECHO_RESPONSE_UNSIGNED is not covered by the HMAC and SIGNATURE.

5.3. HIP Packets

There are eight basic HIP packets (see Table 10). Four are for the HIP base exchange, one is for updating, one is for sending notifications, and two are for closing a HIP association.

Packet type	Packet name
1	I1 - the HIP Initiator Packet
2	R1 - the HIP Responder Packet
3	I2 - the Second HIP Initiator Packet
4	R2 - the Second HIP Responder Packet
16	UPDATE - the HIP Update Packet
17	NOTIFY - the HIP Notify Packet
18	CLOSE - the HIP Association Closing Packet
19	CLOSE_ACK - the HIP Closing Acknowledgment Packet

Table 10: HIP packets and packet type numbers

Packets consist of the fixed header as described in Section 5.1, followed by the parameters. The parameter part, in turn, consists of zero or more TLV-coded parameters.

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

See Notation (Section 2.2) for used operations.

In the future, an OPTIONAL upper-layer payload MAY follow the HIP header. The Next Header 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.

5.3.1. 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 to use opportunistic mode by using NULL (all zeros) as the Responder's HIT. See also "HIP Opportunistic Mode" (Section 4.1.6).

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.

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

5.3.2. 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_SIGNED, ]
             HIP_SIGNATURE_2 )
             <, ECHO_REQUEST_UNSIGNED >i)
```

Valid control bits: A

If the Responder's HI is an anonymous one, the A control MUST be set.

The Initiator's HIT MUST match the one received in I1. If the Responder has multiple HIs, the Responder's HIT used MUST match Initiator's request. If the Initiator used opportunistic mode, the Responder may select freely among its HIs. See also "HIP Opportunistic Mode" (Section 4.1.6).

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 no longer accepted.

The Puzzle contains a Random #I and the difficulty K. The difficulty K indicates the number of lower-order bits, in the puzzle hash result, that must be zeros; see Section 4.1.2. 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 precomputed R1 just prior sending it to the peer.

The Diffie-Hellman value is ephemeral, and one value SHOULD be used only for one connection. Once the Responder has received a valid response to an R1 packet, that Diffie-Hellman value SHOULD be deprecated. Because it is possible that the Responder has sent the same Diffie-Hellman value to different hosts simultaneously in corresponding R1 packets, those responses should also be accepted. However, as a defense against I1 storms, an implementation MAY propose, and re-use if not avoidable, the same Diffie-Hellman value for a period of time, for example, 15 minutes. By using a small number of different puzzles for a given Diffie-Hellman value, the R1 packets can be precomputed and delivered as quickly as I1 packets arrive. A scavenger process should clean up unused Diffie-Hellman values and puzzles.

Re-using Diffie-Hellman public keys opens up the potential security risk of more than one Initiator ending up with the same keying material (due to faulty random number generators). Also, more than one Initiator using the same Responder public key half may lead to potentially easier cryptographic attacks and to imperfect forward security.

However, these risks involved in re-using the same key are statistical; that is, the authors are not aware of any mechanism that would allow manipulation of the protocol so that the risk of the re-use of any given Responder Diffie-Hellman public key would differ from the base probability. Consequently, it is RECOMMENDED that implementations avoid re-using the same D-H key with multiple Initiators, but because the risk is considered statistical and not known to be manipulable, the implementations MAY re-use a key in order to ease resource-constrained implementations and to increase the probability of successful communication with legitimate clients even under an I1 storm. In particular, when it is too expensive to generate enough precomputed R1 packets to supply each potential Initiator with a different D-H key, the Responder MAY send the same D-H key to several Initiators, thereby creating the possibility of multiple legitimate Initiators ending up using the same Responder-side public key. However, as soon as the Responder knows that it

will use a particular D-H key, it SHOULD stop offering it. This design is aimed to allow resource-constrained Responders to offer services under I1 storms and to simultaneously make the probability of D-H key re-use both statistical and as low as possible.

If a future version of this protocol is considered, we strongly recommend that these issues be studied again. Especially, the current design allows hosts to become potentially more vulnerable to a statistical, low-probability problem during I1 storm attacks than what they are if no attack is taking place; whether this is acceptable or not should be reconsidered in the light of any new experience gained.

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 [RFC3602] with HMAC-SHA-1-96 [RFC2404].

The ECHO_REQUEST_SIGNED and ECHO_REQUEST_UNSIGNED contains data that the sender wants to receive unmodified in the corresponding response packet in the ECHO_RESPONSE_SIGNED or ECHO_RESPONSE_UNSIGNED parameter.

The signature is calculated over the whole HIP envelope, after setting the Initiator's HIT, header checksum, as well as the Opaque field and the Random #I in the PUZZLE parameter temporarily to zero, and excluding any parameters that follow the signature, as described in Section 5.2.13. This allows the Responder to use precomputed R1s. The Initiator SHOULD validate this signature. It SHOULD check that the Responder's HI received matches with the one expected, if any.

5.3.3. 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 } or HOST_ID,
             [ ECHO_RESPONSE_SIGNED ,]
             HMAC,
             HIP_SIGNATURE
```

<, ECHO_RESPONSE_UNSIGNED>i))

Valid control bits: A

The HITs used MUST match the ones used previously.

If the Initiator's 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 RHASH(I | ... | J) MUST be zero.

The Diffie-Hellman value is ephemeral. If precomputed, a scavenger process should clean up unused Diffie-Hellman values. The Responder may re-use Diffie-Hellman values under some conditions as specified in Section 5.3.2.

The HIP_TRANSFORM contains the single encryption and integrity transform selected by the Initiator, that will be used to protect the HI exchange. The chosen transform MUST correspond to one offered by the Responder in the R1. All implementations MUST support the AES transform [RFC3602].

The Initiator's HI MAY be encrypted using the HIP_TRANSFORM encryption algorithm. The keying material is derived from the Diffie-Hellman exchanged as defined in Section 6.5.

The ECHO_RESPONSE_SIGNED and ECHO_RESPONSE_UNSIGNED contain the unmodified Opaque data copied from the corresponding echo request parameter.

The HMAC is calculated over the whole HIP envelope, excluding any parameters after the HMAC, as described in Section 6.4.1. The Responder MUST validate the HMAC.

The signature is calculated over the whole HIP envelope, excluding any parameters after the HIP_SIGNATURE, as described in Section 5.2.12. The Responder MUST validate this signature. It MAY use either the HI in the packet or the HI acquired by some other means.

5.3.4. 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 the whole HIP envelope, with Responder's HOST_ID parameter concatenated with the HIP envelope. The HOST_ID parameter is removed after the HMAC calculation. The procedure is described in Section 6.4.1.

The signature is calculated over the whole HIP envelope.

The Initiator MUST validate both the HMAC and the signature.

5.3.5. UPDATE - the HIP Update Packet

Support for the UPDATE packet is MANDATORY.

The HIP header values for the UPDATE packet:

```
  Header:
    Packet Type = 16
    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 forgo 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 retransmitted without incrementing SEQ. If the same subset of parameters is included in multiple UPDATEs with different SEQs, the host MUST ensure that the receiver's processing of the parameters multiple times will not result in a protocol error.

5.3.6. 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. NOTIFY packets are unacknowledged. The receiver can handle the packet only as informational, and SHOULD NOT change its HIP state (Section 4.4.1) based purely on a received NOTIFY packet.

The HIP header values for the NOTIFY packet:

```
  Header:
    Packet Type = 17
    SRC HIT = Sender's HIT
    DST HIT = Recipient's HIT, or zero if unknown

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

Valid control bits: None

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

5.3.7. CLOSE - the HIP Association Closing Packet

The HIP header values for the CLOSE packet:

```
  Header:
    Packet Type = 18
    SRC HIT = Sender's HIT
    DST HIT = Recipient's HIT

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

Valid control bits: none

The sender MUST include an ECHO_REQUEST_SIGNED 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_RESPONSE_SIGNED corresponding to the received ECHO_REQUEST_SIGNED.

5.3.8. CLOSE_ACK - the HIP Closing Acknowledgment Packet

The HIP header values for the CLOSE_ACK packet:

```
  Header:
    Packet Type = 19
    SRC HIT = Sender's HIT
    DST HIT = Recipient's HIT

  IP ( HIP ( ECHO_RESPONSE_SIGNED, 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.

5.4. ICMP Messages

When a HIP implementation detects a problem with an incoming packet, and it either cannot determine the identity of the sender of the packet or does not have any existing HIP association with the sender of the packet, it MAY respond with an ICMP packet. Any such replies MUST be rate-limited as described in [RFC2463]. In most cases, the ICMP packet will have the Parameter Problem type (12 for ICMPv4, 4 for ICMPv6), with the Pointer field pointing to the field that caused the ICMP message to be generated.

5.4.1. Invalid Version

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

5.4.2. Other Problems with the HIP Header and Packet Structure

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

5.4.3. Invalid Puzzle Solution

If a HIP implementation receives an I2 packet that has an invalid puzzle solution, the behavior 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 into 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 [RFC0792].

5.4.4. 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, and with the Pointer pointing to 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, and NOTIFY. When introducing new packet types, a specification SHOULD define the appropriate rules for sending or not sending this kind of ICMP reply.

6. 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 4.4. 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 association 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.

6.1. Processing Outgoing Application Data

In a HIP host, an application can send application-level data using an identifier specified via the underlying API. The API can be a backwards-compatible API (see [RFC5338]), using identifiers that look similar to IP addresses, or a completely new API, providing enhanced services related to Host Identities. Depending on the HIP implementation, the identifier provided to the application may be different; for example, it can be a HIT or an IP address.

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 transferred in the network using the appropriate source and destination IP addresses.

In this document, conceptual processing rules are defined only for the base case where both hosts have only single usable IP addresses; the multi-address multi-homing case will be specified separately.

The following conceptual algorithm describes the steps that are required for handling 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.

3. If there is no active HIP association with the given <source, destination> HIT pair, one must be created by running the base exchange. While waiting for the base exchange to complete, the implementation SHOULD queue at least one packet per HIP association to be formed, and it MAY queue more than one.

4. Once there is an active HIP association 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. Before sending the packet, the HITs in the datagram are replaced with suitable IP addresses. For IPv6, the rules defined in [RFC3484] 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.

6.2. Processing Incoming Application Data

The following conceptual algorithm describes the incoming datagram handling when HITs are used at the receiving host as application-level identifiers. More detailed steps for processing packets are defined in corresponding transport format documents.

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 the ESP Security Parameter Index (SPI).

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.

   Depending on the used transport mode, the verification method can vary. While the HI (as well as HIT) is used as the higher-layer identifier, the verification method has to verify that the data packet was sent by a node identity and that the actual identity maps to this particular HIT. When using ESP transport format [RFC5202], the verification is done using the SPI value in the data packet to find the corresponding SA with associated HIT and key, and decrypting the packet with that associated key.

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. When demultiplexing the datagram, the right upper-layer socket is based on the HITs.

6.3. Solving the Puzzle

This subsection describes the puzzle-solving details.

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 hash is created by concatenating, in network byte order, the following data, in the following order and using the RHASH algorithm:

64-bit random value I, in network byte order, as appearing in R1 and I2.

128-bit Initiator's HIT, in network byte order, as appearing in the HIP Payload in R1 and I2.

128-bit Responder's 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 puzzle, the K low-order bits of the resulting RHASH digest must be zero.

Notes:

i) The length of the data to be hashed is 48 bytes.

ii) All the data in the hash input MUST be in network byte order.

iii) The order of the Initiator's and Responder's HITs are different in the R1 and I2 packets; see Section 5.1. Care must be taken to copy the values in the right order to the hash input.

The following procedure describes the processing steps involved, assuming that the Responder chooses to precompute the R1 packets:

```
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:
   Ltrunc( RHASH( I | HIT-I | HIT-R | J ), 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 := Ltrunc( RHASH( I | HIT-I | HIT-R | J ), K )
   Rejects if V != 0
   Accept if V == 0
```

6.4. HMAC and SIGNATURE Calculation and Verification

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

6.4.1. HMAC Calculation

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

Both the Initiator and the Responder should take some care when verifying or calculating the HMAC_2. Specifically, the Responder should preserve other parameters than the HOST_ID when sending the R2. Also, the Initiator has to preserve the HOST_ID exactly as it was received in the R1 packet.

The scope of the calculation for HMAC and HMAC_2 is:

```
HMAC: { HIP header | [ Parameters ] }
```

where Parameters include all HIP parameters of the packet that is being calculated with Type values from 1 to (HMAC's Type value - 1)

and exclude parameters with Type values greater or equal to HMAC's Type value.

During HMAC calculation, the following applies:

- In the HIP header, the Checksum field is set to zero.
- In the HIP header, the Header Length field value is calculated to the beginning of the HMAC parameter.

Parameter order is described in Section 5.2.1.

HMAC_2: { HIP header | [Parameters] | HOST_ID }

where Parameters include all HIP parameters for the packet that is being calculated with Type values from 1 to (HMAC_2's Type value - 1) and exclude parameters with Type values greater or equal to HMAC_2's Type value.

During HMAC_2 calculation, the following applies:

- In the HIP header, the Checksum field is set to zero.
- In the HIP header, the Header Length field value is calculated to the beginning of the HMAC_2 parameter and added to the length of the concatenated HOST_ID parameter length.
- HOST_ID parameter is exactly in the form it was received in the R1 packet from the Responder.

Parameter order is described in Section 5.2.1, except that the HOST_ID parameter in this calculation is added to the end.

The HMAC parameter is defined in Section 5.2.10 and the HMAC_2 parameter in Section 5.2.11. The HMAC calculation and verification process (the process applies both to HMAC and HMAC_2 except where HMAC_2 is mentioned separately) is as follows:

Packet sender:

1. Create the HIP packet, without the HMAC, HIP_SIGNATURE, HIP_SIGNATURE_2, or any other parameter with greater Type value than the HMAC parameter has.
2. In case of HMAC_2 calculation, add a HOST_ID (Responder) parameter to the end of the packet.

3. Calculate the Header Length field in the HIP header including the added HOST_ID parameter in case of HMAC_2.

4. Compute the HMAC using either HIP-gl or HIP-lg integrity key retrieved from KEYMAT as defined in Section 6.5.

5. In case of HMAC_2, remove the HOST_ID parameter from the packet.

6. Add the HMAC parameter to the packet and any parameter with greater Type value than the HMAC's (HMAC_2's) that may follow, including possible HIP_SIGNATURE or HIP_SIGNATURE_2 parameters

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 parameter, as well as all other parameters that follow it with greater Type value including possible HIP_SIGNATURE or HIP_SIGNATURE_2 fields, saving the contents if they will be needed later.

3. In case of HMAC_2, build and add a HOST_ID parameter (with Responder information) to the packet. The HOST_ID parameter should be identical to the one previously received from the Responder.

4. Recalculate the HIP packet length in the HIP header and clear the Checksum field (set it to all zeros). In case of HMAC_2, the length is calculated with the added HOST_ID parameter.

5. Compute the HMAC using either HIP-gl or HIP-lg integrity key as defined in Section 6.5 and verify it against the received HMAC.

6. Set Checksum and Header Length field in the HIP header to original values.

7. In case of HMAC_2, remove the HOST_ID parameter from the packet before further processing.

6.4.2. Signature Calculation

The following process applies both to the HIP_SIGNATURE and HIP_SIGNATURE_2 parameters. When processing HIP_SIGNATURE_2, the only difference is that instead of HIP_SIGNATURE parameter, the HIP_SIGNATURE_2 parameter 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 parameter is defined in Section 5.2.12 and the HIP_SIGNATURE_2 parameter in Section 5.2.13.

The scope of the calculation for HIP_SIGNATURE and HIP_SIGNATURE_2 is:

HIP_SIGNATURE: { HIP header | [Parameters] }

where Parameters include all HIP parameters for the packet that is being calculated with Type values from 1 to (HIP_SIGNATURE's Type value - 1).

During signature calculation, the following apply:

- In the HIP header, the Checksum field is set to zero.
- In the HIP header, the Header Length field value is calculated to the beginning of the HIP_SIGNATURE parameter.

Parameter order is described in Section 5.2.1.

HIP_SIGNATURE_2: { HIP header | [Parameters] }

where Parameters include all HIP parameters for the packet that is being calculated with Type values from 1 to (HIP_SIGNATURE_2's Type value - 1).

During signature calculation, the following apply:

- In the HIP header, the Initiator's HIT field and Checksum fields are set to zero.
- In the HIP header, the Header Length field value is calculated to the beginning of the HIP_SIGNATURE_2 parameter.
- PUZZLE parameter's Opaque and Random #I fields are set to zero.

Parameter order is described in Section 5.2.1.

Signature calculation and verification process (the process applies both to HIP_SIGNATURE and HIP_SIGNATURE_2 except in the case where HIP_SIGNATURE_2 is separately mentioned):

Packet sender:

1. Create the HIP packet without the HIP_SIGNATURE parameter or any parameters that follow the HIP_SIGNATURE parameter.

2. Calculate the Length field and zero the Checksum field in the HIP header. In case of HIP_SIGNATURE_2, set Initiator's HIT field in the HIP header as well as PUZZLE parameter's Opaque and Random #I fields to zero.

3. Compute the signature using the private key corresponding to the Host Identifier (public key).

4. Add the HIP_SIGNATURE parameter to the packet.

5. Add any parameters that follow the HIP_SIGNATURE parameter.

6. Recalculate the Length field in the HIP header, and calculate the Checksum field.

Packet receiver:

1. Verify the HIP header Length field.

2. Save the contents of the HIP_SIGNATURE parameter and any parameters following the HIP_SIGNATURE parameter 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). In case of HIP_SIGNATURE_2, set Initiator's HIT field in HIP header as well as PUZZLE parameter's Opaque and Random #I fields to zero.

4. Compute the signature and verify it against the received signature using the packet sender's Host Identifier (public key).

5. Restore the original packet by adding removed parameters (in step 2) and resetting the values that were set to zero (in step 3).

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

6.5. HIP KEYMAT Generation

HIP keying material is derived from the Diffie-Hellman session key, Kij, produced during the HIP base exchange (Section 4.1.3). 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   = RHASH( Kij | sort(HIT-I | HIT-R) | I | J | 0x01 )
K2   = RHASH( Kij | K1 | 0x02 )
K3   = RHASH( Kij | K2 | 0x03 )
...
K255 = RHASH( Kij | K254 | 0xff )
K256 = RHASH( 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.

I and J values are from the puzzle and its solution that were exchanged in R1 and I2 messages when this HIP association was set up. Both hosts have to store I and J values for the HIP association for future use.

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

SHA-256 256 bits

NULL 0 bits

If other key sizes are used, they must be treated as different encryption algorithms and defined separately.

6.6. 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 4.5.4.

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 5.3.1. 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. See also "HIP Opportunistic Mode" (Section 4.1.6).

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.

6.6.1. 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. That is, if it retries to initialize the connection after timeout, it MUST NOT send the I1 packet to more than one destination address. These limitations are placed in order to avoid congestion of the network, and potential DoS attacks that might happen, e.g., because someone's claim to have hundreds or thousands of addresses could generate a huge number of I1 messages from the Initiator.

As the Responder is not guaranteed to distinguish the duplicate I1s it receives at several of its addresses (because it avoids storing states when it answers back an R1), the Initiator may receive several duplicate R1s.

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 6.8.

6.6.2. Processing Incoming ICMP Protocol Unreachable Messages

A host may receive an ICMP 'Destination Protocol Unreachable' message as a response to sending a 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.

6.7. Processing Incoming I1 Packets

An implementation SHOULD reply to an I1 with an R1 packet, unless the implementation is unable or unwilling to set up a HIP association. If the implementation is unable to set up 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 set up 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.

It is RECOMMENDED that the HIP state machine does not 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's 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 Responder is in I1-SENT state, it must make a comparison between the sender's HIT and its own (i.e., the receiver's) HIT. If the sender's HIT is greater than its own HIT, it should drop the I1 and stay at I1-SENT. If the sender's HIT is smaller than its own HIT, it should send R1 and stay at I1-SENT. The HIT comparison goes similarly as in Section 6.5.

4. 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 5.3.2.

5. The R1 MUST contain the received Responder's 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.

6. The Responder sends the R1 to the source IP address of the I1 packet.

6.7.1. 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, and SHOULD be deprecated and not used once a valid response I2 packet has been received from an Initiator. During an I1 message storm, an R1 packet may be re-used beyond this limit. 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.

6.7.2. 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 5.4.

6.8. 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). Unless the I1 was sent in opportunistic mode (see Section 4.1.6), the IP addresses in the received R1 packet SHOULD be ignored and, when looking up the right HIP association, the received R1 SHOULD be matched against the

associations using only the HITs. If a match exists, the system 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, and the 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 5.2.13.

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 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.

8. The system SHOULD attempt to validate the HIT against the received Host Identity by using the received Host Identity to construct a HIT and verify that it matches the Sender's HIT.

9. The system MUST store the received R1 generation counter for future reference.

10. The system attempts to solve the puzzle in R1. The system MUST terminate the search after exceeding the remaining lifetime of the puzzle. If the puzzle is not successfully solved, the implementation may either resend I1 within the retry bounds or abandon the HIP exchange.

11. 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 6.5. 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.

12. 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.

13. The system initializes the remaining variables in the associated state, including Update ID counters.

14. The system prepares and sends an I2, as described in Section 5.3.3.

15. 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.

16. 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.

6.8.1. 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 possibly good R1, after which it MAY try again by sending a new I1 packet.

6.9. 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 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 A 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's state machine is in the R2-SENT state, the system 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 the state machine stays in R2-SENT.

4. If the system's state machine is in the I2-SENT state, the system makes a comparison between its local and sender's HITs (similarly as in Section 6.5). If the local HIT is smaller than the sender's HIT, it should drop the I2 packet, use the peer Diffie-Hellman key and nonce I from the R1 packet received earlier, and get the local Diffie-Hellman key and nonce J from the I2 packet sent to the peer earlier. Otherwise, the system should process the received I2 packet and drop any previously derived Diffie-Hellman keying material Kij it might have formed upon sending the I2 previously. The peer Diffie-Hellman key and the nonce J are taken from the just arrived I2 packet. The local Diffie-Hellman key and the nonce I are the ones that were earlier sent in the R1 packet.

5. If the system's state machine is in the I1-SENT state, and the HITs in the I2 match those used in the previously sent I1, the system uses this received I2 as the basis for the HIP association it was trying to form, and stops retransmitting I1 (provided that the I2 passes the below additional checks).

6. If the system's state machine is in any other state than R2-SENT, the system 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.

7. The system MUST validate the solution to the puzzle by computing the hash described in Section 5.3.3 using the same RHASH algorithm.

8. 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.

9. 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 6.5. If the Diffie-Hellman Group ID is unsupported, the I2 packet is silently dropped.

10. The encrypted HOST_ID is decrypted by the Initiator encryption key defined in Section 6.5. If the decrypted data is not a HOST_ID parameter, the I2 packet is silently dropped.

11. The implementation SHOULD also verify that the Initiator's HIT in the I2 corresponds to the Host Identity sent in the I2. (Note: some middleboxes may not able to make this verification.)

12. The system MUST verify the HMAC according to the procedures in Section 5.2.10.

13. The system MUST verify the HIP_SIGNATURE according to Section 5.2.12 and Section 5.3.3.

14. If the checks above are valid, then the system proceeds with further I2 processing; otherwise, it discards the I2 and its state machine remains in the same state.

15. The I2 packet may have the A bit set -- in this case, the system MAY choose to refuse it by dropping the I2 and the state machine returns to state UNASSOCIATED. If the A bit is set, the Initiator's HIT is anonymous and should not be stored.

16. The system initializes the remaining variables in the associated state, including Update ID counters.

17. Upon successful processing of an I2 when the system's state machine is in state UNASSOCIATED, I1-SENT, I2-SENT, or R2-SENT, an R2 is sent and the system's state machine transitions to state R2-SENT.

18. Upon successful processing of an I2 when the system's state machine is in state ESTABLISHED, the old HIP association is dropped and a new one is installed, an R2 is sent, and the system's state machine transitions to R2-SENT.

19. Upon the system's state machine transitioning to R2-SENT, the system starts a timer. The state machine transitions to

ESTABLISHED if 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 system's state machine has moved to ESTABLISHED). If the timer expires (allowing for maximal retransmissions of I2s), the state machine transitions to ESTABLISHED.

6.9.1. Handling Malformed Messages

If an implementation receives a malformed I2 message, the behavior SHOULD depend on how many 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 5.4.

6.10. Processing Incoming R2 Packets

An R2 received in states UNASSOCIATED, I1-SENT, or ESTABLISHED 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 an incoming R2 packet:

1. The system MUST verify that the HITs in use correspond to the HITs that were received in the R1.

2. The system MUST verify the HMAC_2 according to the procedures in Section 5.2.11.

3. The system MUST verify the HIP signature according to the procedures in Section 5.2.12.

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.

6.11. 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.

The system first determines whether there are any outstanding UPDATE messages that may conflict with the new UPDATE message under consideration. When multiple UPDATEs are outstanding (not yet acknowledged), the sender must assume that such UPDATEs may be processed in an arbitrary order. Therefore, any new UPDATEs that depend on a previous outstanding UPDATE being successfully received and acknowledged MUST be postponed until reception of the necessary ACK(s) occurs. One way to prevent any conflicts is to only allow one outstanding UPDATE at a time. However, allowing multiple UPDATEs may improve the performance of mobility and multihoming protocols.

The following steps define the conceptual processing rules for sending UPDATE packets.

1. The first UPDATE packet is sent with Update ID of zero. Otherwise, the system increments its own Update ID value by one before continuing the below steps.

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 peer's Update ID found in a 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 * RTT estimate. If multiple UPDATEs are outstanding, multiple timers are in effect.

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. If no acknowledgment is received from the peer after UPDATE_RETRY_MAX times, the HIP association is considered to be broken and the state machine should move from state ESTABLISHED to state CLOSING as depicted in Section 4.4.3. The UPDATE timer is cancelled upon receiving an ACK from the peer that acknowledges receipt of the UPDATE.

6.12. Receiving UPDATE Packets

When a system receives an UPDATE packet, its processing depends on the state of the HIP association and the presence and values of the SEQ and ACK parameters. Typically, an UPDATE message also carries optional parameters whose handling is defined in separate documents.

For each association, the peer's next expected in-sequence Update ID ("peer Update ID") is stored. Initially, this value is zero. Update ID comparisons of "less than" and "greater than" are performed with respect to a circular sequence number space.

The sender may send multiple outstanding UPDATE messages. These messages are processed in the order in which they are received at the receiver (i.e., no resequencing is performed). When processing UPDATEs out-of-order, the receiver MUST keep track of which UPDATEs were previously processed, so that duplicates or retransmissions are ACKed and not reprocessed. A receiver MAY choose to define a receive window of Update IDs that it is willing to process at any given time, and discard received UPDATEs falling outside of that window.

The following steps define the conceptual processing rules for receiving UPDATE packets.

1. If there is no corresponding HIP association, the implementation MAY reply with an ICMP Parameter Problem, as specified in Section 5.4.4.

2. If the association is in the ESTABLISHED state and the SEQ (but not ACK) parameter is present, the UPDATE is processed and replied to as described in Section 6.12.1.

3. If the association is in the ESTABLISHED state and the ACK (but not SEQ) parameter is present, the UPDATE is processed as described in Section 6.12.2.

4. If the association is in the ESTABLISHED state and there is both an ACK and SEQ in the UPDATE, the ACK is first processed as described in Section 6.12.2, and then the rest of the UPDATE is processed as described in Section 6.12.1.

6.12.1. Handling a SEQ Parameter in a Received UPDATE Message

The following steps define the conceptual processing rules for handling a SEQ parameter in a received UPDATE packet.

1. If the Update ID in the received SEQ is not the next in the sequence of Update IDs and is greater than the receiver's window for new UPDATEs, the packet MUST be dropped.

2. If the Update ID in the received SEQ corresponds to an UPDATE that has recently been processed, 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 stored packet would be OK, though.) It is recommended that a host cache UPDATE packets sent with ACKs 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 parameters in the UPDATE are then processed. The system MUST record the Update ID in the received SEQ parameter, for replay protection.

6. An UPDATE acknowledgment packet with ACK parameter is prepared and sent to the peer. This ACK parameter may be included in a separate UPDATE or piggybacked in an UPDATE with SEQ parameter, as described in Section 5.3.5. The ACK parameter MAY acknowledge more than one of the peer's Update IDs.

6.12.2. Handling an ACK Parameter in a Received UPDATE Packet

The following steps define the conceptual processing rules for handling an ACK parameter in a received UPDATE packet.

1. The sequence number reported in the ACK must match with an earlier sent UPDATE packet that has not already been acknowledged. If no match is found or if the ACK does not acknowledge a new UPDATE, the packet MUST either be dropped if no SEQ parameter is present, or the processing steps in Section 6.12.1 are followed.

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 6.11) so that the now acknowledged UPDATE is no longer retransmitted. If multiple UPDATEs are newly acknowledged, multiple timers are stopped.

6.13. Processing NOTIFY Packets

Processing NOTIFY packets is OPTIONAL. If processed, any errors in a received NOTIFICATION parameter SHOULD be logged. Received errors MUST be considered only as informational, and the receiver SHOULD NOT change its HIP state (Section 4.4.1) purely based on the received NOTIFY message.

6.14. Processing CLOSE Packets

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

6.15. Processing CLOSE_ACK Packets

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

The CLOSE_ACK uses HMAC and SIGNATURE for verification. The state is discarded when the state changes to UNASSOCIATED and, after that, the host MAY respond with an ICMP Parameter Problem to an incoming CLOSE message (see Section 5.4.4).

6.16. Handling State Loss

In the case of system crash and unanticipated state loss, the system SHOULD delete the corresponding HIP state, including the keying material. That is, the state SHOULD NOT be stored on stable storage. If the implementation does drop the state (as RECOMMENDED), it 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.

7. 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 HI, at least one of which SHOULD be reserved for anonymous usage. Although anonymous HIs will be rarely used as Responders' 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's HIT to Responder's HIT. This ACL SHOULD also include preferred transform and local lifetimes.

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.

8. 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.

The 384-bit Diffie-Hellman Group is targeted to be used in hosts that either do not require or are not powerful enough for handling strong cryptography. Although there is a risk that with suitable equipment the encryption can be broken in real time, the 384-bit group can provide some protection for end-hosts that are not able to handle any

stronger cryptography. When the security provided by the 384-bit group is not enough for applications on a host, the support for this group should be turned off in the configuration.

Denial-of-service attacks often 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, until some Initiator has provided a valid response to such an R1 packet. During an I1 storm, the host may reuse the same D-H value also even if some Initiator has provided a valid response using that particular D-H value. However, such behavior is discouraged and should be avoided. Using the same Diffie-Hellman values and random puzzle #I value has some risks. 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 an 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. This 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 restarting host 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 HIP association. Because both CLOSE and CLOSE_ACK messages contain an HMAC, an outsider cannot close a connection. The presence of an additional SIGNATURE allows middleboxes 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 5.4.4) 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 4.1.4.

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 the HI (after having got the I2 HIP packet) and verify that the HI indeed can be trusted. 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.

The HIP Opportunistic Mode concept has been introduced in this document, but this document does not specify what the semantics of such a connection setup are for applications. There are certain concerns with opportunistic mode, as discussed in Section 4.1.6.

NOTIFY messages are used only for informational purposes and they are unacknowledged. A HIP implementation cannot rely solely on the information received in a NOTIFY message because the packet may have been replayed. It SHOULD NOT change any state information based purely on a received NOTIFY message.

Since not all hosts will ever support HIP, ICMP 'Destination Protocol Unreachable' messages 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. Normally, if an I1 message received by a Responder was a bogus one sent by an attacker, the Responder may receive an ICMP message from the IP address the R1 message was sent to. However, a sophisticated attacker can try to take advantage of such a behavior and try to break up the HIP exchange by sending such an ICMP message to the Responder before the Initiator has a chance to send a valid I2 message. Hence, the Responder SHOULD NOT act on such an ICMP message. Especially, it SHOULD NOT remove any minimal state created when it sent the R1 HIP packet (if it did create one), but wait for either a valid I2 HIP packet or the natural timeout (that is, if R1 packets are tracked at all). Likewise, the Initiator should ignore any ICMP message while waiting for an R2 HIP packet, and should delete any pending state only after a natural timeout.

9. IANA Considerations

IANA has reserved protocol number 139 for the Host Identity Protocol.

This document defines a new 128-bit value under the CGA Message Type namespace [RFC3972], 0xF0EF F02F BFF4 3D0F E793 0C3C 6E61 74EA, to be used for HIT generation as specified in ORCHID [RFC4843-bis].

This document also creates a set of new namespaces. These are described below.

Packet Type

The 7-bit Packet Type field in a HIP protocol packet describes the type of a HIP protocol message. It is defined in Section 5.1. The current values are defined in Sections 5.3.1 through 5.3.8.

New values are assigned through IETF Consensus [RFC2434].

HIP Version

The four-bit Version field in a HIP protocol packet describes the version of the HIP protocol. It is defined in Section 5.1. The only currently defined value is 1. New values are assigned through IETF Consensus.

Parameter Type

The 16-bit Type field in a HIP parameter describes the type of the parameter. It is defined in Section 5.2.1. The current values are defined in Sections 5.2.3 through 5.2.21.

With the exception of the assigned Type codes, the Type codes 0 through 1023 and 61440 through 65535 are reserved for future base protocol extensions, and are assigned through IETF Consensus.

The Type codes 32768 through 49141 are reserved for experimentation. 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.

All other Type codes are assigned through First Come First Served, with Specification Required [RFC2434].

Group ID

The eight-bit Group ID values appear in the DIFFIE_HELLMAN parameter and are defined in Section 5.2.6. New values either from the reserved or unassigned space are assigned through IETF Consensus.

Suite ID

The 16-bit Suite ID values in a HIP_TRANSFORM parameter are defined in Section 5.2.7. New values either from the reserved or unassigned space are assigned through IETF Consensus.

DI-Type

The four-bit DI-Type values in a HOST_ID parameter are defined in Section 5.2.8. New values are assigned through IETF Consensus.

Notify Message Type

The 16-bit Notify Message Type values in a NOTIFICATION parameter are defined in Section 5.2.17.

Notify Message Type values 1-10 are used for informing about errors in packet structures, values 11-20 for informing about problems in parameters containing cryptographic related material, values 21-30 for informing about problems in authentication or packet integrity verification. Parameter numbers above 30 can be used for informing about other types of errors or events. Values 51-8191 are error types reserved to be allocated by IANA. Values 8192-16383 are error types for experimentation. Values 16385-40959 are status types to be allocated by IANA, and values 40960-65535 are status types for experimentation. New values in ranges 51-8191 and 16385-40959 are assigned through First Come First Served, with Specification Required.

10. 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 versions 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 valuable input at early stages of discussions about identifier handling 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 puzzle 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. Erik Nordmark supplied the CLOSE-mechanism for closing connections. Rodney Thayer and Hugh Daniels provided extensive feedback. In the early times of this document, 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 was 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 document was split in two, one containing the base exchange and the other one defining how to use ESP. Some modifications to the protocol proposed by Aura, et al., [AUR03] were added at a later stage.

11. References

11.1. Normative References

[FIPS.180-2.2002] National Institute of Standards and Technology, "Secure Hash Standard", FIPS PUB 180-2, August 2002, <http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf>.

[FIPS.95-1.1993] National Institute of Standards and Technology, "Codes for the Identification of Federal and Federally Assisted Organizations", FIPS PUB 95-1, January 1993.

[RFC0768] Postel, J., "User Datagram Protocol", STD 6, RFC 768, August 1980.

[RFC1035] Mockapetris, P., "Domain names - implementation and specification", STD 13, RFC 1035, November 1987.

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

[RFC2404] Madson, C. and R. Glenn, "The Use of HMAC-SHA-1-96 within ESP and AH", RFC 2404, November 1998.

[RFC2451] Pereira, R. and R. Adams, "The ESP CBC-Mode Cipher Algorithms", RFC 2451, November 1998.

[RFC2460] Deering, S. and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification", RFC 2460, December 1998.

[RFC2463] Conta, A. and S. Deering, "Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification", RFC 2463, December 1998.

[RFC2536] Eastlake, D., "DSA KEYs and SIGs in the Domain Name System (DNS)", RFC 2536, March 1999.

[RFC2898] Kaliski, B., "PKCS #5: Password-Based Cryptography Specification Version 2.0", RFC 2898, September 2000.

[RFC3110] Eastlake, D., "RSA/SHA-1 SIGs and RSA KEYs in the Domain Name System (DNS)", RFC 3110, May 2001.

[RFC3484] Draves, R., "Default Address Selection for Internet Protocol version 6 (IPv6)", RFC 3484, February 2003.

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

[RFC3602] Frankel, S., Glenn, R., and S. Kelly, "The AES-CBC Cipher Algorithm and Its Use with IPsec", RFC 3602, September 2003.

[RFC3972] Aura, T., "Cryptographically Generated Addresses (CGA)", RFC 3972, March 2005.

[RFC4034] Arends, R., Austein, R., Larson, M., Massey, D., and S. Rose, "Resource Records for the DNS Security Extensions", RFC 4034, March 2005.

[RFC4282] Aboba, B., Beadles, M., Arkko, J., and P. Eronen, "The Network Access Identifier", RFC 4282, December 2005.

[RFC4307] Schiller, J., "Cryptographic Algorithms for Use in the Internet Key Exchange Version 2 (IKEv2)", RFC 4307, December 2005.

[RFC4843-bis] Nikander, P., Laganier, J., and F. Dupont, "STUB: An IPv6 Prefix for Overlay Routable Cryptographic Hash Identifiers (ORCHID)", draft-laganier-rfc4843-bis-00 (work in progress), February 2010.

[RFC5202] Jokela, P., Moskowitz, R., and P. Nikander, "Using the Encapsulating Security Payload (ESP) Transport Format with the Host Identity Protocol (HIP)", RFC 5202, April 2008.

[RFC5702] Jansen, J., "Use of SHA-2 Algorithms with RSA in DNSKEY and RRSIG Resource Records for DNSSEC", RFC 5702, October 2009.

11.2. Informative References

[AUR03] Aura, T., Nagarajan, A., and A. Gurtov, "Analysis of the HIP Base Exchange Protocol", in Proceedings of 10th Australasian Conference on Information Security and Privacy, July 2003.

[CRO03] Crosby, SA. and DS. Wallach, "Denial of Service via Algorithmic Complexity Attacks", in Proceedings of Usenix Security Symposium 2003, Washington, DC., August 2003.

[DIF76] Diffie, W. and M. Hellman, "New Directions in Cryptography", IEEE Transactions on Information Theory vol. IT-22, number 6, pages 644-654, Nov 1976.

[FIPS.197.2001] National Institute of Standards and Technology, "Advanced Encryption Standard (AES)", FIPS PUB 197, November 2001, <http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>.

[KAU03] Kaufman, C., Perlman, R., and B. Sommerfeld, "DoS protection for UDP-based protocols", ACM Conference on Computer and Communications Security , Oct 2003.

[KRA03] Krawczyk, H., "SIGMA: The 'SIGn-and-MAc' Approach to Authenticated Diffie-Hellman and Its Use in the IKE-Protocols", in Proceedings of CRYPTO 2003, pages 400-425, August 2003.

[RFC0792] Postel, J., "Internet Control Message Protocol", STD 5, RFC 792, September 1981.

[RFC2412] Orman, H., "The OAKLEY Key Determination Protocol", RFC 2412, November 1998.

[RFC2434] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 2434, October 1998.

[RFC4306] Kaufman, C., "Internet Key Exchange (IKEv2) Protocol", RFC 4306, December 2005.

[RFC5204] Laganier, J. and L. Eggert, "Host Identity Protocol (HIP) Rendezvous Extension", RFC 5204, April 2008.

[RFC5205] Nikander, P. and J. Laganier, "Host Identity Protocol (HIP) Domain Name System (DNS) Extensions", RFC 5205, April 2008.

[RFC5206] Nikander, P., Henderson, T., Vogt, C., and J. Arkko, "End-Host Mobility and Multihoming with the

Host Identity Protocol", RFC 5206, April 2008.

[RFC5338] Henderson, T., Nikander, P., and M. Komu, "Using the Host Identity Protocol with Legacy Applications", RFC 5338, September 2008.

[RFC5533] Nordmark, E. and M. Bagnulo, "Shim6: Level 3 Multihoming Shim Protocol for IPv6", RFC 5533, June 2009.

[btns-c-api] Richardson, M., Williams, N., Komu, M., and S. Tarkoma, "C-Bindings for IPsec Application Programming Interfaces", draft-ietf-btns-c-api-04 (work in progress), March 2009.

[rfc4423-bis] Moskowitz, R. and P. Nikander, "Host Identity Protocol Architecture", draft-moskowitz-hip-rfc4423-bis-00 (work in progress), December 2009.

Appendix A. Using Responder Puzzles

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. If the implementation is based on this appendix, it MAY contain some local modification that makes an attacker's task harder.

The Responder creates a secret value S, that it regenerates periodically. The Responder needs to remember the two latest values of S. Each time the S is regenerated, the R1 generation counter value is incremented by one.

The Responder generates a pre-signed R1 packet. The signature for pre-generated R1s must be recalculated when the Diffie-Hellman key is recomputed or when the R1_COUNTER value changes due to S value regeneration.

When the Initiator sends the I1 packet for initializing a connection, the Responder gets the HIT and IP address from the packet, and generates an I value for the puzzle. The I value is set to the pre-signed R1 packet.

```
I value calculation:
I = Ltrunc( RHASH ( S | HIT-I | HIT-R | IP-I | IP-R ), 64)
```

The RHASH algorithm is the same that is used to generate the Responder's HIT value.

From an incoming I2 packet, the Responder gets the required information to validate the puzzle: HITs, IP addresses, and the information of the used S value from the R1_COUNTER. Using these values, the Responder can regenerate the I, and verify it against the I received in the I2 packet. If the I values match, it can verify the solution using I, J, and difficulty K. If the I values do not match, the I2 is dropped.

```
puzzle_check:
V := Ltrunc( RHASH( I2.I | I2.hit_i | I2.hit_r | I2.J ), K )
if V != 0, drop the packet
```

If the puzzle solution is correct, the I and J values are stored for later use. They are used as input material when keying material is generated.

Keeping state about failed puzzle solutions depends on the implementation. Although it is possible for the Responder not to keep any state information, it still may do so to protect itself against certain attacks (see Section 4.1.1).

Appendix B. Generating a Public Key Encoding from an HI

The following pseudo-code illustrates the process to generate a public key encoding from an HI 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;

}
```

Appendix C. Example Checksums for HIP Packets

The HIP checksum for HIP packets is specified in Section 5.1.1. 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 HITs with the prefix of 2001:10 followed by zeros, followed by a decimal 1 or 2, respectively.

The following example is defined only for testing a checksum calculation. The address format for the IPv4-compatible IPv6 address is not a valid one, but using these IPv6 addresses when testing an IPv6 implementation gives the same checksum output as an IPv4 implementation with the corresponding IPv4 addresses.

C.1. IPv6 HIP Example (I1)

Source Address:	::192.168.0.1	
Destination Address:	::192.168.0.2	
Upper-Layer Packet Length:	40	0x28
Next Header:	139	0x8b
Payload Protocol:	59	0x3b
Header Length:	4	0x4
Packet Type:	1	0x1
Version:	1	0x1
Reserved:	1	0x1
Control:	0	0x0
Checksum:	446	0x1be
Sender's HIT :	2001:10::1	
Receiver's HIT:	2001:10::2	

C.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).

C.3. TCP Segment

Regardless of whether IPv6 or IPv4 is used, the TCP and UDP sockets use the IPv6 pseudo-header format [RFC2460], with the HITs used in place of the IPv6 addresses.

Sender's HIT:	2001:10::1	
Receiver's HIT:	2001:10::2	
Upper-Layer Packet Length:	20	0x14
Next Header:	6	0x06
Source port:	65500	0xffdc
Destination port:	22	0x0016
Sequence number:	1	0x00000001
Acknowledgment number:	0	0x00000000
Header length:	20	0x14
Flags:	SYN	0x02
Window size:	65535	0xffff
Checksum:	28618	0x6fca
Urgent pointer:	0	0x0000

```
0x0000:  6000 0000 0014 0640 2001 0010 0000 0000
0x0010:  0000 0000 0000 0001 2001 0010 0000 0000
0x0020:  0000 0000 0000 0002 ffdc 0016 0000 0001
0x0030:  0000 0000 5002 ffff 6fca 0000
```

Appendix D. 384-Bit Group

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

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

Its hexadecimal value is:

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

The generator is: 2.

Appendix E. OAKLEY Well-Known Group 1

See also [RFC2412] for definition of OAKLEY well-known group 1.

OAKLEY Well-Known Group 1: A 768-bit prime

The prime is 2^768 - 2^704 - 1 + 2^64 * { [2^638 pi] + 149686 }.

The hexadecimal value is:

```
FFFFFFFF FFFFFFFF C90FDAA2 2168C234 C4C6628B 80DC1CD1
29024E08 8A67CC74 020BBEA6 3B139B22 514A0879 8E3404DD
EF9519B3 CD3A431B 302B0A6D F25F1437 4FE1356D 6D51C245
E485B576 625E7EC6 F44C42E9 A63A3620 FFFFFFFF FFFFFFFF
```

This has been rigorously verified as a prime.

The generator is: 22 (decimal)

Appendix F. ORCHID Generation Algorithms for HITs

The HIT as an ORCHID [RFC4843-bis] consists of three parts: A 28-bit prefix, a 4-bit encoding of the ORCHID generation algorithm (OGA) and the representation of the public key. The OGA is an index pointing to the specific aglorithm by which the 96-bit hashed encoding is generated. The OGA is protocol specific and is to be interpreted as defined below for all protocols that use the same context ID as HIP.

The set of used OGAs will be extended to counter the progress in computation capabilities and vulnerabilities in the employed algorithms. The intended use of the OGA is to introduce a new OGA

and phase out an old OGA before it becomes insecure. Since the 4-bit OGA field only permits 16 OGAs to be used in parallel, phased-out OGAs must be reused at some point. In such a case, there will be a rollover of the OGA index and the next newly introduced OGA will start with a lower OGA index than the previoulsy introduced one. The rollover effectively deprecates the reused OGA. For a smooth transition, the OGA should be deprecated a considerable time before the OGA index is reused.

The OGA of the Responder's HIT determines the RHASH and the hash function to be used for the HMAC in HIP control packets. For OGAs that employ a hash function, this hash function is used. For OGAs that do not need a hash function, a hash function is defined in Table 11.

The following OGAs are defined for HIT generation. The input for each generation algorithm is the encoding of the HI as defined in Section 3.2. The output is 96 bits long and is directly used in the ORCHID.

Index	Hash function	Description
0		Reserved
1	SHA-1	SHA-1 truncated to 96 bits
2	SHA-256	SHA-256 truncated to 96 bits

Table 11: Orchid Generation Algorithms

F.1. SHA-1 Truncated to 96 Bits

For generating a HIT with OGA index 1, the input is hashed with SHA-1 [FIPS.95-1.1993] and the left-most 96 bits are used as output: LTRUNC(SHA-1(HI), 96).

F.2. SHA-256 Truncated to 96 Bits

For generating a HIT with OGA index 2, the input is hashed with SHA-256 [FIPS.180-2.2002] and the left-most 96 bits are used as output: LTRUNC(SHA-256(HI), 96).

Authors' Addresses

Robert Moskowitz (editor)
ICSAlabs, An Independent Division of Verizon Business Systems
1000 Bent Creek Blvd, Suite 200
Mechanicsburg, PA
USA

EMail: robert.moskowitz@icsalabs.com

Pekka Nikander
Ericsson Research NomadicLab
JORVAS FIN-02420
FINLAND

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

Petri Jokela
Ericsson Research NomadicLab
JORVAS FIN-02420
FINLAND

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

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

EMail: thomas.r.henderson@boeing.com

Tobias Heer
RWTH Aachen University, Distributed Systems Group
Ahornstrasse 55
Aachen 52062
Germany

EMail: heer@cs.rwth-aachen.de
URI: http://ds.cs.rwth-aachen.de/members/heer