Workgroup: Internet Engineering Task Force
Internet-Draft: draft-bider-ssh-quic-05
Published: 12 July 2020
Intended Status: Informational
Expires: 13 January 2021
Authors: d. bider
Bitvise Limited

QUIC-based UDP Transport for Secure Shell (SSH)

Abstract

The Secure Shell protocol (SSH) [RFC4251] is widely used for purposes including secure remote administration, file transfer using SFTP and SCP, and encrypted tunneling of TCP connections. Because it is based on TCP, SSH suffers similar problems as motivate the HTTP protocol to transition to UDP-based QUIC [QUIC]. These include: unauthenticated network intermediaries can trivially disconnect SSH sessions; SSH connections are lost when mobile clients change IP addresses; performance limitations in OS-based TCP stacks; many round-trips to establish a connection; duplicate flow control on the level of the connection as well as channels. This memo specifies SSH key exchange over UDP and leverages QUIC to provide a UDP-based transport.

Status of This Memo

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

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

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."

This Internet-Draft will expire on 13 January 2021.

Copyright Notice

Table of Contents

1. Introduction

THIS DOCUMENT IS AN EARLY VERSION AND IS A WORK IN PROGRESS.

NON-LATEST DRAFT VERSIONS MUST BE DISREGARDED.

IMPLEMENTATION AT THIS STAGE IS EXPERIMENTAL.

CONTACT THE AUTHOR IF YOU INTEND TO IMPLEMENT.

This memo specifies SSH key exchange over UDP, and then leverages QUIC to provide a UDP-based transport for SSH. QUIC's use of the TLS handshake is replaced with a one-roundtrip SSH/QUIC key exchange. The SSH Authentication Protocol [RFC4252] is then conducted over QUIC stream 0, and the SSH Connection Protocol [RFC4254] is modified to use QUIC streams.

1.1. Requirements Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

2. SSH/QUIC key exchange

2.1. Distinguishing SSH key exchange from QUIC datagrams

UDP datagrams which form the SSH/QUIC key exchange are sent between the same client and server IP addresses and ports as QUIC datagrams. It is therefore necessary for clients and servers to distinguish SSH key exchange datagrams from QUIC datagrams.

A distinction is allowed by that SSH/QUIC only requires the sending of QUIC Short Header Packets. Therefore, all UDP datagrams where the first byte has its high bit set MUST be handled as part of an SSH/QUIC key exchange.

2.2. Wire Encoding

This memo uses wire encoding types "byte", "uint32", "uint64", "mpint" and "string" with meanings as described in [RFC4251].

This memo defines the following new wire encoding type.

"short-str" is a shorter version of "string", encoded as follows:

```
byte        n = short-str-len (unsigned, 0..255)
byte[n]     short-str-value
```

Figure 1

2.3. Obfuscated Envelope

Since SSH servers are commonly used for remote administration, they are a high-value target for password guessing. One of the most common complaints from SSH server administrators is the high frequency of password guessing connections from random clients.

Experience shows that obfuscating the SSH protocol with an obfuscation keyword is a valuable measure which thwarts password guessing. This increases practical security of the SSH ecosystem even if obfuscation does not thwart narrowly targeted attacks.

Every SSH/QUIC connection is parameterized by an obfuscation keyword. The obfuscation keyword is a sequence of Unicode characters entered by a user. Applications MUST permit the user to enter any Unicode characters except code points in the Unicode category "Cc" (Control). These are decimal code points 0..31 and 127..159, inclusive.

An SSH/QUIC server SHOULD allow the administrator to configure an obfuscation keyword for each interface and port on which the server is accepting SSH/QUIC connections. An SSH/QUIC client MUST allow the user to configure an obfuscation keyword separately for outgoing connections to each server address and port.

The obfuscation keyword MUST be optional for users to configure. If a user does not configure it, the obfuscated envelope is applied as if the obfuscation keyword was an empty character sequence.

All SSH/QUIC key exchange packets are sent as UDP datagrams in the following obfuscated envelope:

```
byte[16]  obfs-nonce - high bit of first byte MUST be set
byte[]    obfs-payload
byte[16]  obfs-tag
```

Figure 2

The field "obfs-nonce" contains random bytes generated by the sender of the UDP datagram. The high bit of the first byte of "obfs-nonce" MUST be set to distinguish the packet from QUIC datagrams. See Section 2.1.

The field "obfs-payload" contains the SSH/QUIC key exchange packet encrypted using AEAD_AES_256_GCM [RFC5116]. The AEAD is invoked as follows:

- The secret key K is a SHA-256 digest of the obfuscation keyword in UTF-8 encoding.
- The nonce N is the field "obfs-nonce".
- The plaintext P is the unencrypted packet payload.
- Associated data A is empty.
- The ciphertext C is stored in "obfs-payload".

The length of encrypted "obfs-payload" is implied by the UDP datagram length, and is calculated by subtracting the fixed lengths of "obfs-nonce" and "obfs-tag".

The field "obfs-tag" stores the GCM tag. Receivers MUST check the tag and MUST ignore datagrams where the GCM tag is invalid.

2.4. Packet Size Limits

Clients and servers MUST accept SSH_QUIC_INIT, SSH_QUIC_REPLY and SSH_QUIC_CANCEL packets with unencrypted "obfs-payload" sizes at least up to 32768 bytes. This corresponds to minimum SSH packet size limits which implementations must support as per [RFC4253], Section 6.1.

2.5. Required QUIC Versions and TLS Cipher Suites

Clients and servers are REQUIRED to implement QUIC protocol version 1 once it is standardized in [QUIC] and [QUIC-TLS].

Clients and servers are REQUIRED to implement the TLS cipher suites TLS_AES_128_GCM_SHA256 and TLS_AES_256_GCM_SHA384 [RFC8446]. Other cipher suites are optional.

The requirement to implement any particular QUIC protocol version or TLS cipher suite expires on the 5-year anniversary of the publishing of this memo. At that point, implementers SHOULD consult any new

standards documents if available, or survey the practical use of SSH/QUIC for implementation guidance.

2.6. Random Elements

Unlike SSH over TCP, the packets SSH_QUIC_INIT and SSH_QUIC_REPLY do not provide a "cookie" field for random data. Instead, clients and servers MUST insert random data using the extensibility mechanisms described for each SSH key exchange packet.

At the very minimum, clients and servers MUST insert at least 16 Random Bytes or at least one Random Name, in locations as described for SSH_QUIC_INIT (Section 2.8.1) and SSH_QUIC_REPLY (Section 2.9.2). If at all possible, the random data MUST come from a cryptographically strong random source. Implementations that are unable to meet this requirement MUST still insert the minimum amount of random data, as unpredictably as they are able. Compromising on this requirement reduces the security of any sessions created on the basis of such SSH_QUIC_INIT and SSH_QUIC_REPLY.

Lengths of Random Names and Random Bytes SHOULD be chosen at random such that lengths in the shorter end of the range are significantly more probable, but long lengths are still selected. See Appendix A.

Random Bytes

Random Bytes are generated with values 0..255, in a range of lengths as specified for the particular usage context.

Random Name

A Random Name is generated in one of two forms: Assigned Form or Private Form. One of the two forms is randomly chosen so that Assigned Form, which is shorter, is more likely. The maximum length of a Random Name is 64 bytes.

Assigned Form

A Random Name in Assigned Form is generated as a string of random characters with ASCII values 33..126 (inclusive), except @ and the comma (","). Other characters MUST NOT be included. To avoid collisions as effectively as a random UUID, a Random Name in Assigned Form MUST contain at least 20 random characters if using the complete character set. A Random Name in Assigned Form MUST then be of length 20..64 bytes.

Implementations MAY remove up to 7 characters from the character set -- reducing it to 85..91 characters -- without increasing the minimum length. If the character set is further reduced to 69..84

characters, implementations MUST generate at least 21 random characters instead.

Example Random Names in Assigned Form:

```
d`kbi>AGrj~r{3lo_Q4r
wNT)=/8C<(DB1|tr:>1f[xq>9bG
u7^dE'\EE_}N}^"J5syI?/8jIxup#s7BM:]>{IT_p3Z~<KLa]bIW643XYh07jqZu
```

Figure 3

Private Form

Implementations MAY generate a Random Name in Private Form by first generating a Random Name in Assigned Form, then appending a domain name suffix which the implementer controls. A Random Name generated this way MUST NOT exceed 64 bytes. Example Random Names in Private Form:

```
(qKR8W%&zJu;$RQkWa[b@bitvise.com
BDPhhC_vI?+8$e_CGty->wJDYIBX.4zzQ$@denisbider.com
?`z4bb/}</P[pRJ=SvcCV<k0eUPDIHid#e1giY>&Wuf6O7CE?cA`$j"@bider.us
```

Figure 4

Alternately, implementations MAY generate a Random Name in Anonymous Form with the format "(local)@(domain).example.com". In this case, both "(local)" and "(domain)" are replaced by random ASCII characters from the set A..Z, a..z, and 0..9. This is to ensure that the suffix has valid domain name syntax.

To avoid collisions as effectively as a random UUID, a Random Name in Anonymous Form MUST contain at least 22 random characters. A Random Name in Anonymous Form MUST then be of length 35..64 bytes.

2.7. Errors in Key Exchange

To assist users, clients and servers SHOULD report key exchange errors as follows:

1. If a server cannot send a successful SSH_QUIC_REPLY, it SHOULD send an Error Reply. See Section 2.9.1.

2. If a client receives an invalid SSH_QUIC_REPLY, it SHOULD send an SSH_QUIC_CANCEL. See Section 2.10.

Both packet types use the following extension pairs.

2.7.1. "disc-reason" Extension Pair

"ext-pair-name" contains "disc-reason".

"ext-pair-data" encodes a uint32 with the SSH disconnect reason code. Reason codes are defined in the table "Disconnect Messages Reason Codes and Descriptions" in the IANA registry "Secure Shell (SSH) Protocol Parameters" [IANA-SSH].

2.7.2. "err-desc" Extension Pair

"ext-pair-name" contains "err-desc".

"ext-pair-data" encodes a human-readable error description in any language intended to be relevant to the user, encoded as UTF-8.

Receivers that process error descriptions MUST validate that the description is valid UTF-8. If a description is long, receivers SHOULD truncate it to a reasonable length depending on the processing context. For example, a debug log file can record a full 32 kB error description, while a production log file SHOULD truncate it to a much shorter length.

2.8. SSH_QUIC_INIT

A client begins an SSH/QUIC session by sending one or more copies of SSH_QUIC_INIT. If multiple copies are sent, copies intended for the same connection MUST be identical. A reasonable strategy is to send one copy every 50 - 500 ms until the client receives a valid SSH_QUIC_REPLY or times out. A server MUST remember recently received SSH_QUIC_INIT packets and send identical SSH_QUIC_REPLY responses. If different SSH_QUIC_INIT packets are received from the same client IP address, the server MUST assume they are intended to begin separate connections, even if they specify the same "client-connection-id". A server MAY implement throttling of incoming connections, by IP address or otherwise, where excessive SSH_QUIC_INIT packets are disregarded. Once a server receives QUIC data confirming that a client has processed an SSH_QUIC_REPLY, the server MUST disregard any further identical copies of the same SSH_QUIC_INIT, at least until the SSH/QUIC session started by such an SSH_QUIC_INIT ends.

SSH_QUIC_INIT is an obfuscated datagram (Section 2.3) where "obfs-payload" encrypts the following:

```
byte        SSH_QUIC_INIT = 1        (see Extensibility)
short-str   client-connection-id     (MAY be empty)
short-str   server-name-indication   (MUST NOT be empty)

byte        v = nr-quic-versions     (MUST NOT be zero)
uint32[v]   client-quic-versions

string      client-sig-algs          (MUST NOT be empty)

byte f = nr-trusted-fingerprints     (MAY be zero)
the following 1 field repeated f times:
   short-str   trusted-fingerprint   (MUST NOT be empty)

byte k = nr-client-kex-algs          (MUST NOT be zero)
the following 2 fields repeated k times:
   short-str   client-kex-alg-name   (MUST NOT be empty)
   string      client-kex-alg-data   (MUST NOT be empty)

byte c = nr-cipher-suites            (MUST NOT be zero)
the following 1 field repeated c times:
   short-str   quic-tls-cipher-suite

byte e = nr-ext-pairs                (see Extensibility)
the following 2 fields repeated e times:
   short-str   ext-pair-name         (MUST NOT be empty)
   string      ext-pair-data         (MAY be empty)

byte[0..] padding: all 0xFF to minimal obfs-payload size 1200
```

Figure 5

SSH_QUIC_INIT does not include an SSH version string or compression negotiation. Instead, clients MUST use SSH_MSG_EXT_INFO for these purposes. See Section 4.

SSH_QUIC_INIT does not include a "cookie" field for random data. Clients MUST insert random data using the packet's extensibility mechanisms. See Section 2.8.1 and Section 2.6.

The field "client-connection-id" contains a QUIC Connection ID of length 0..20 bytes. The server will use this as the QUIC Destination Connection ID in QUIC packets sent to the client. Clients MAY send an empty Connection ID if they are using other means of routing connections.

The field "server-name-indication" SHOULD contain the server DNS name if a DNS name was entered by the user when configuring the connection. This can be invaluable in hosting environments: it allows servers to expose to clients multiple distinct identities on the same network address and port. If non-empty, the field MUST

encode the DNS name entered by the user as a string consisting of printable US-ASCII characters. Internationalized domain names MUST be represented in their US-ASCII encoding. If the user connected directly to an IP address, this field MUST be empty. This avoids disclosing private information in case of port forwarded connections. Example non-empty values:

```
localhost
server.example.com
xn--bcher-kva.example
```

Figure 6

The fields "client-quic-versions" enumerate QUIC protocol versions supported by the client. The client MUST send at least one version. The client MUST send supported versions in the order it prefers the server to use them.

The field "client-sig-algs" MUST contain at least one signature algorithm supported by the client for server authentication. These are the same algorithms as used in SSH_MSG_KEXINIT ([RFC4253], Section 7.1) in the field "server_host_key_algorithms". The client MUST send signature algorithms in the order it prefers the server to use them.

The client SHOULD include algorithms in "client-sig-algs" as follows:

- If the client does not yet trust any host key for the server: "client-sig-algs" SHOULD include all signature algorithms supported and enabled by the client for use with any server.

- Otherwise, the client already trusts some host keys for the server. In this case, if the client sends any "trusted-fingerprint" fields, then "client-sig-algs" SHOULD include all signature algorithms supported and enabled by the client for use with any server.

- Otherwise, the client already trusts some host keys for the server, but does not send any "trusted-fingerprint" fields. In this case, "client-sig-algs" MUST include only signature algorithms associated with the host keys the client already trusts for this server.

There MAY be zero or more "trusted-fingerprint" fields. Each "trusted-fingerprint" contains a binary fingerprint of a host key that is trusted for this connection by the client. The fingerprint algorithm is left unspecified. The server SHOULD try to match the fingerprint using all algorithms it supports which produce the provided fingerprint size. The current recommended fingerprint

algorithm is SHA-256, with fingerprint size 32 bytes. Servers MUST tolerate the presence of unrecognized fingerprints of any size. The preference order of trusted fingerprints is dominated by the preference order of algorithms in "client-sig-algs".

The packet MUST include at least one SSH key exchange algorithm, encoded as a pair of "client-kex-alg-name" and "client-kex-alg-data" fields. The field "client-kex-alg-name" MUST specify a key exchange method which would be valid in the field "kex_algorithms" in SSH_MSG_KEXINIT under [RFC4253], Section 7.1. In addition, the key exchange method MUST meet criteria in Section 3.

If the client wishes to simply advertise its support for a particular key exchange algorithm, but does not prefer to use it in this connection, it MAY enumerate the algorithm with empty "client-kex-alg-data". Otherwise, if the client wishes to allow the algorithm to be used, it MUST include non-empty "client-kex-alg-data". In this case, "client-kex-alg-data" contains the client's portion of key exchange inputs as specified in Section 3. The client MAY send multiple key exchange algorithms with filled-out "client-kex-alg-data". The client MUST send these algorithms in the order it prefers the server to use them.

There MUST be at least one "quic-tls-cipher-suite" field. Each of these specifies a TLS cipher suite ([RFC8446], Appendix B.4) which is supported by the client, and which can be used with a version of QUIC ([QUIC], [QUIC-TLS]) supported by the client. The client MUST enumerate supported cipher suites in the order it prefers the server to use them.

The client MAY send any number of extensions, encoded as a pair of "ext-pair-name" and "ext-pair-data" fields. This memo defines no extensions for SSH_QUIC_INIT, but see Section 2.8.1.

The "padding" field contains all 0xFF bytes to ensure that the unencrypted "obfs-payload" for SSH_QUIC_INIT is at least 1200 bytes in length. Servers MUST ignore smaller SSH_QUIC_INIT packets. This is REQUIRED to prevent abuse of SSH_QUIC_INIT for Amplified Reflection DDoS. If the unencrypted size of "obfs-payload" is already 1200 bytes or larger, the padding MAY be omitted.

2.8.1. Extensibility

Implementations MUST allow room for future extensibility of SSH_QUIC_INIT in the following manners:

1. By using a different packet type in the first byte -- this is, a value other than 1 used by SSH_QUIC_INIT. Servers MUST NOT penalize clients for sending unknown packet types unless there

is another reason to penalize the client, such as a blocked IP address or the sheer volume of datagrams.

2. By including algorithms in "client-sig-algs" which are unknown to or not supported by the server. Servers MUST tolerate the presence of such algorithms.

3. By including fingerprints in "trusted-fingerprints" that use algorithms or lengths that are unknown to or not supported by the server. Servers MUST tolerate the presence of such fingerprints.

4. By including SSH key exchange algorithms which are unknown to or not supported by the server, with algorithm data in a format that's unknown to or not supported by the server. Servers MUST tolerate the presence of such algorithms and their data.

5. By including QUIC TLS cipher suites which are unknown to or not supported by the server. Servers MUST tolerate the presence of such cipher suites.

6. By including extensions which are unknown to or not supported by the server, with extension data in a format that's unknown to or not supported by the server. Servers MUST tolerate the presence of such extensions and their data.

Experience shows that any extensibility which is not actively exercised is lost due to implementations that lock down expectations incorrectly. Therefore, all clients MUST do at least one of the following, in each SSH_QUIC_INIT packet, at random:

1. In the field "client-sig-algs", include in a random position at least one Random Name (Section 2.6).

2. In the fields "client-quic-versions", include in a random position a version number of the form 0x0A?A?A?A, where ? indicates a random nibble. See [QUIC], section "Versions". Note the difference from the random version pattern in the server's SSH_QUIC_REPLY. Due to the minimal amount of entropy provided by this rule, this MUST NOT be the only insertion of randomness made in a packet.

3. Include in a random position at least one host key fingerprint consisting of 16..255 Random Bytes (Section 2.6).

4. Include in a random position at least one SSH key exchange algorithm where the field "client-kex-alg-name" contains a Random Name, and the field "client-kex-alg-data" contains 0..1000 Random Bytes.

5. In the fields "quic-tls-cipher-suite", include in a random position at least one entry consisting of 16..255 Random Bytes.

6. In extension pairs, include in a random position at least one extension where the field "ext-pair-name" contains a Random Name, and the field "ext-pair-value" contains 0..1000 Random Bytes.

2.9. SSH_QUIC_REPLY

Implementations MUST take care to prevent abuse of the SSH/QUIC key exchange for Amplified Reflection DDoS attacks. This means:

1. A server MUST NOT send more than one SSH_QUIC_REPLY in response to any individual SSH_QUIC_INIT.

2. A server MUST NOT respond to any SSH_QUIC_INIT with unencrypted "obfs-payload" smaller than 1200 bytes.

3. Before sending an SSH_QUIC_REPLY, the server MUST verify that the reply is shorter than the SSH_QUIC_INIT packet to which it is replying. If this is not the case, the server MUST send an Error Reply (Section 2.9.1). Such an Error Reply MUST be shorter than the SSH_QUIC_INIT packet.

SSH_QUIC_REPLY is an obfuscated datagram (Section 2.3) where "obfs-payload" encrypts the following:

```
byte        SSH_QUIC_REPLY = 2
short-str   client-connection-id
short-str   server-connection-id   (Non-empty except on error)

byte        v = nr-quic-versions   (MUST NOT be zero)
uint32[v]   server-quic-versions

string      server-sig-algs        (MUST NOT be empty)
string      server-kex-algs        (MUST NOT be empty)

byte c = nr-cipher-suites          (MUST NOT be zero)
the following 1 field repeated c times:
   short-str   quic-tls-cipher-suite

byte e = nr-ext-pairs              (see Extensibility)
the following 2 fields repeated e times:
   short-str   ext-pair-name       (MUST NOT be empty)
   string      ext-pair-data       (MAY be empty)

string      server-kex-alg-data    (Non-empty except on error)
```

Figure 7

SSH_QUIC_REPLY does not include an SSH version string or compression negotiation. Instead, servers MUST use SSH_MSG_EXT_INFO for these purposes. See Section 4.

SSH_QUIC_REPLY does not include a "cookie" field for random data. Servers MUST insert random data using the packet's extensibility mechanisms. See Section 2.9.2 and Section 2.6.

The field "client-connection-id" encodes the "client-connection-id" sent by the client in SSH_QUIC_INIT.

The field "server-connection-id" contains a QUIC Connection ID of length 1..20 bytes. The client will use this as the QUIC Destination Connection ID in QUIC packets sent to the server. This field MUST be empty if sending an Error Reply (Section 2.9.1), and MUST NOT be empty otherwise.

The fields "server-quic-versions" enumerate QUIC protocol versions supported by the server. The server MUST send at least one version. The QUIC version used for the connection is the first version enumerated in "client-quic-versions" which is also present in "server-quic-versions". If there is no such version, see Section 2.9.1.

The field "server-sig-algs" MUST contain at least one signature algorithm supported by the server. The server SHOULD enumerate all signature algorithms for which it has host keys. These are the same algorithms as used in SSH_MSG_KEXINIT ([RFC4253], Section 7.1) in the field "server_host_key_algorithms". In the SSH/QUIC key exchange, the server MUST use a host key it possesses that (1) matches any fingerprint enumerated in the "trusted-fingerprint" fields in SSH_QUIC_INIT; and (2) can be used with the earliest possible signature algorithm enumerated in "client-sig-algs". If there are multiple such host keys, the client's preference order in "client-sig-algs" dominates the preference order of "trusted-fingerprint". If there is no such host key, the server MUST use any host key that can be used with the earliest possible signature algorithm enumerated in "client-sig-algs". If there is no such host key either, see Section 2.9.1.

The field "server-kex-algs" MUST contain at least one SSH key exchange algorithm supported by the server. The key exchange algorithm which is used in the connection is the first algorithm sent in client's SSH_QUIC_INIT where: (1) the field "client-kex-alg-data" is non-empty, and (2) the algorithm is also present in "server-kex-algs". If there is no such key exchange algorithm, see Section 2.9.1.

There MUST be at least one "quic-tls-cipher-suite" field. Each of these specifies a TLS cipher suite ([RFC8446], Appendix B.4) which is supported by the server, and which can be used with a version of QUIC ([QUIC], [QUIC-TLS]) supported by the server. The TLS cipher suite which is used for the connection is the first suite sent in the client's SSH_QUIC_INIT where: (1) the cipher suite is supported by the negotiated QUIC protocol version, and (2) the cipher suite is present in the server's SSH_QUIC_REPLY. If there is no such cipher suite, see Section 2.9.1.

The server MAY send any number of extensions, encoded as a pair of "ext-pair-name" and "ext-pair-data" fields. Some extensions are defined for use with an Error Reply (see Section 2.9.1). Other extensions MAY be defined in the future; see Section 2.9.2.

The field "server-kex-alg-data" MUST be empty if the packet is an Error Reply. Otherwise, this field contains information for the SSH key exchange method: see Section 3. Generally, this includes the server's portion of key exchange inputs; the server's host key; and the server's signature of the calculated exchange hash.

2.9.1. Error Reply

If a server encounters an error which it is useful and appropriate to communicate to the client, the server MAY send an "Error Reply" version of SSH_QUIC_REPLY. Such a reply is created as follows:

- The server includes and populates all fields of SSH_QUIC_REPLY as it would normally, except that the fields "server-connection-id" and "server-kex-alg-data" MUST remain empty.

- In the extension pair fields, a "disc-reason" Extension Pair MUST be included. An "err-desc" Extension Pair MAY also be included. See Section 2.7.

- Extensibility considerations for SSH_QUIC_REPLY in Section 2.9.2 also apply to an Error Reply.

If the server does not support any of the QUIC protocol versions enumerated by the client, the server SHOULD send an Error Reply with the disconnect reason code SSH_DISCONNECT_PROTOCOL_VERSION_NOT_SUPPORTED.

In the following circumstances, the server SHOULD send an Error Reply with the disconnect reason code SSH_DISCONNECT_KEY_EXCHANGE_FAILED:

- If the server could have sent a successful SSH_QUIC_REPLY, but it would have been larger than the client's SSH_QUIC_INIT, even though the SSH_QUIC_INIT met or exceeded the minimum length.

*If the server possesses no server host key that can be used with a signature algorithm enumerated in the client's SSH_QUIC_INIT.

*If the server supports no key exchange algorithms matching the ones for which the client sent "client-kex-alg-data" in SSH_QUIC_INIT.

*If the server supports no TLS cipher suites enumerated in the client's SSH_QUIC_INIT.

Besides "disc-reason", an "err-desc" extension pair SHOULD be included to describe the specific error.

2.9.2. Extensibility

Implementations MUST allow room for future extensibility of SSH_QUIC_REPLY in the following manners:

1. By including algorithms in "server-sig-algs" which are unknown to or not supported by the client. Clients MUST tolerate the presence of such algorithms.

2. By including SSH key exchange algorithms which are unknown to or not supported by the client, with algorithm data in a format that's unknown to or not supported by the client. Clients MUST tolerate the presence of such algorithms and their data.

3. By including QUIC TLS cipher suites which are unknown to or not supported by the client. Clients MUST tolerate the presence of such cipher suites.

4. By including extensions which are unknown to or not supported by the client, with extension data in a format that's unknown to or not supported by the client. Clients MUST tolerate the presence of such extensions and their data.

Experience shows that any extensibility which is not actively exercised is lost due to implementations that lock down expectations incorrectly. Therefore, all servers MUST do at least one of the following, in each SSH_QUIC_REPLY packet, at random:

1. In the fields "server-quic-versions", include in a random position a version number of the form 0xFA?A?A?A, where ? indicates a random nibble. See [QUIC], section "Versions". Note the difference from the random version pattern in the client's SSH_QUIC_INIT. Due to the minimal amount of entropy provided by this rule, this MUST NOT be the only insertion of randomness made in a packet.

2. In the field "server-sig-algs", include in a random position one Random Name (Section 2.6).

3. In the field "server-kex-algs", include in a random position one Random Name (Section 2.6).

4. In the fields "quic-tls-cipher-suite", include in a random position one entry consisting of 16..64 Random Bytes.

5. In extension pairs, include in a random position one extension pair where the field "ext-pair-name" contains a Random Name, and the field "ext-pair-value" contains 0..100 Random Bytes.

2.10. SSH_QUIC_CANCEL

If a client cannot process the server's successful SSH_QUIC_REPLY, the client SHOULD report the error to the server using SSH_QUIC_CANCEL.

A client MUST NOT send an SSH_QUIC_CANCEL in response to an SSH_QUIC_REPLY which is itself an Error Reply. A client MUST assume that such a connection was already canceled by the server.

A client SHOULD send two or more copies of SSH_QUIC_CANCEL, in transmissions separated by a fraction of a second, to increase the likelihood of successful delivery. The server sends no acknowledgment to SSH_QUIC_CANCEL. After the server has received SSH_QUIC_CANCEL, it MUST ignore subsequent copies of SSH_QUIC_CANCEL for the same connection.

SSH_QUIC_CANCEL is an obfuscated datagram (Section 2.3) where "obfs-payload" encrypts the following:

```
byte        SSH_QUIC_CANCEL = 3
short-str   server-connection-id

byte e = nr-ext-pairs             (see Extensibility)
the following 2 fields repeated e times:
   short-str   ext-pair-name      (MUST NOT be empty)
   string      ext-pair-data      (MAY be empty)
```

Figure 8

The "server-connection-id" field MUST equal the "server-connection-id" field in the server's SSH_QUIC_REPLY.

In the extension pair fields, a "disc-reason" Extension Pair MUST be included. An "err-desc" Extension Pair MAY also be included. See Section 2.7.

2.10.1. Extensibility

Extensibility considerations also apply to SSH_QUIC_CANCEL:

- Clients MAY include extensions which are unknown to or not supported by the server, with extension data in a format that's unknown to or not supported by the server.

- Servers MUST tolerate the presence of such extensions and their data.

- Clients SHOULD include, in a random position, at least one extension pair where the field "ext-pair-name" contains a Random Name, and the field "ext-pair-value" contains 0..300 Random Bytes.

3. Key Exchange Methods

Clients and servers MAY use any key exchange method which is defined for SSH over TCP, whether it is assigned or private, as long as it meets all of the following criteria:

1. The algorithm requires exactly one message from the client to the server, for example SSH_MSG_KEX_ECDH_INIT. We call this message KEXMSG_CLIENT.

2. The algorithm requires exactly one reply from the server to the client, for example SSH_MSG_KEX_ECDH_REPLY. We call this message KEXMSG_SERVER.

3. The algorithm specifies a hash function HASH, for example SHA-256, SHA-384, or SHA-512.

4. The algorithm specifies calculation of an exchange hash H by applying HASH to a concatenation of encoded fields.

5. The algorithm uses a server host key to sign H.

6. The algorithm includes the server's public host key, and the signature of H, in its KEXMSG_SERVER message to the client.

7. The algorithm produces a shared secret K, represented as a signed (positive or negative) multi-precision integer.

Any such algorithm is modified for use in SSH over QUIC as follows:

1. The field "client-kex-alg-data" in SSH_QUIC_INIT encodes the same fields, in the same order, as KEXMSG_CLIENT, including the leading byte for the SSH packet type.

2. The field "server-kex-alg-data" in SSH_QUIC_REPLY encodes the same fields, in the same order, as KEXMSG_SERVER, including the leading byte for the SSH packet type.

3. The calculation of H specified by the algorithm is not performed. Instead, H is calculated by applying the hash function HASH to a concatenation of the following:

```
string   Unencrypted "obfs-payload" content of SSH_QUIC_INIT

string   Unencrypted "obfs-payload" content of SSH_QUIC_REPLY,
         excluding the entire field "server-kex-alg-data"

The fields of "server-kex-alg-data", excluding signature field

mpint    K
```

Figure 9

When a field is excluded as above, the entire encoding of the field is omitted: both the encoding of the content and the encoding of the length.

The SSH packet type byte is included:

* To ensure there are at least two fields in the encoded content. This avoids situations where an outer string (the field "client-kex-alg-data") would contain a single inner string (from KEXMSG_CLIENT). This could confuse implementers to incorrectly encode a single string only.

* For future consistency. The packet type byte may be useful for multiple-roundtrip key exchange methods, for example those using GSS-API [RFC4462]. Such key exchange methods are not currently defined for SSH/QUIC, but can be.

3.1. Required Key Exchange Methods

Clients and servers are REQUIRED to implement the key exchange method "curve25519-sha256" [RFC8731]. All other key exchange methods are optional.

Clients and servers MAY permit the user to disable a required key exchange method. However, required methods MUST be enabled by default.

The requirement to implement any particular key exchange method expires on the 5-year anniversary of the publishing of this memo. At that point, implementers SHOULD consult any new standards documents

if available, or survey the practical use of SSH/QUIC for implementation guidance.

3.2. Example 1: "curve25519-sha256"

When using the SSH key exchange method "curve25519-sha256", the SSH_QUIC_INIT field "client-kex-alg-data" is derived from SSH_MSG_KEX_ECDH_INIT ([RFC5656], Section 4) and contains the following:

```
byte     SSH_MSG_KEX_ECDH_INIT = 30
string   Q_C, client's ephemeral public key octet string
```

Figure 10

The SSH_QUIC_REPLY field "server-kex-alg-data" is derived from SSH_MSG_KEX_ECDH_REPLY and contains the following:

```
byte     SSH_MSG_KEX_ECDH_REPLY = 31
string   K_S, server's public host key
string   Q_S, server's ephemeral public key octet string
string   the signature on the exchange hash
```

Figure 11

The shared secret K is calculated as in [RFC8731]. Then the exchange hash H is calculated by applying SHA-256 to a concatenation of the following:

```
string   Content of SSH_QUIC_INIT
string   Content of SSH_QUIC_REPLY, except "server-kex-alg-data"
byte     SSH_MSG_KEX_ECDH_REPLY = 31
string   K_S, server's public host key
string   Q_S, server's ephemeral public key octet string
mpint    K
```

Figure 12

3.3. Example 2: "diffie-hellman-group14-sha256"

When using the SSH key exchange method "diffie-hellman-group14-sha256" [RFC8268], the SSH_QUIC_INIT field "client-kex-alg-data" is derived from SSH_MSG_KEXDH_INIT ([RFC4253], Section 8) and contains the following:

```
byte     SSH_MSG_KEXDH_INIT = 30
mpint    e
```

Figure 13

The SSH_QUIC_REPLY field "server-kex-alg-data" is derived from SSH_MSG_KEXDH_REPLY and contains the following:

```
byte     SSH_MSG_KEXDH_REPLY = 31
string   server public host key and certificates (K_S)
mpint    f
string   signature of H
```

Figure 14

The shared secret K is calculated as in [RFC4253]. Then the exchange hash H is calculated by applying SHA-256 to a concatenation of the following:

```
string   Content of SSH_QUIC_INIT
string   Content of SSH_QUIC_REPLY, except "server-kex-alg-data"
byte     SSH_MSG_KEXDH_REPLY = 31
string   server public host key and certificates (K_S)
mpint    f
mpint    K
```

Figure 15

4. SSH_MSG_EXT_INFO and the SSH Version String

A common user complaint to SSH application authors is that SSH over TCP sends the application version in plain text. The application version cannot be omitted, otherwise implementations cannot support a number of behaviors which other software versions implement incorrectly.

A prominent example is the order of arguments in the SFTP request SSH_FXP_SYMLINK. To send a request that will have the desired effects, the client MUST consult the server's version string to know whether the server uses the standard order of fields, or a reverse order used by OpenSSH.

SSH over QUIC removes the version string from the SSH key exchange. Instead, all clients and servers are REQUIRED to send and accept SSH_MSG_EXT_INFO [RFC8308], and to include the "ssh-version" extension defined here.

Clients MUST send SSH_MSG_EXT_INFO as the very first SSH packet over QUIC stream 0. The client MUST include the "ssh-version" extension in this SSH_MSG_EXT_INFO.

Servers MUST send SSH_MSG_EXT_INFO either:

1. as the very first SSH packet over QUIC stream 0, and/or

2. immediately preceding the server's SSH_MSG_USERAUTH_SUCCESS.

A server MUST include the "ssh-version" extension in at least one of its SSH_MSG_EXT_INFO. If the server sends SSH_MSG_EXT_INFO at both opportunities, it MAY omit "ssh-version" at the first opportunity, but only if it will send it in the second opportunity. The second SSH_MSG_EXT_INFO sent by the server MAY change a previously sent "ssh-version" extension value to include more specific detail. For example, the server MAY send a more accurate server software version when the client has authenticated. The client MUST use the "ssh-version" value which was most recently received from the server.

4.1. "ssh-version"

The "ssh-version" extension is encoded in SSH_MSG_EXT_INFO as follows:

```
string "ssh-version"
string ssh-version-string
```

Figure 16

The extension value, "ssh-version-string", contains the same SSH version string as sent at the start of SSH over TCP ([RFC4253], Section 4.2), but stripping the prefix "SSH-2.0-". Examples inspired by version strings used in practice:

```
GenericSoftware
Product_1.2.00
0.12 Library: Application 1.23p1
```

Figure 17

4.2. "no-flow-control"

The extension "no-flow-control" has no effect in SSH/QUIC. It SHOULD NOT be sent in SSH/QUIC and MUST be ignored by both parties.

4.3. "delay-compression"

Semantics of the "delay-compression" extension are modified as per Section 6.1.1.

5. QUIC Session Setup

When the server has sent its SSH_QUIC_REPLY, and when the client has received it, they each initialize the QUIC session [QUIC] [QUIC-TLS] as follows:

- The QUIC protocol version is set to the first version advertised in the client's SSH_QUIC_INIT which is also present in the server's SSH_QUIC_REPLY.

- Session state is set as if a TLS handshake had just completed.

- The TLS cipher suite is set to the first TLS cipher suite advertised in SSH_QUIC_INIT which is also present in SSH_QUIC_REPLY.

- The QUIC Key Phase bit is set to 0.

- The shared secrets that would have been obtained from the TLS handshake are instead generated from the SSH key exchange (Section 5.1).

Clients and servers MUST immediately begin to use QUIC Short Header Packets. Implementations MUST NOT send QUIC Long Header Packets, since they could be confused with the SSH/QUIC key exchange.

5.1. Shared Secrets

QUIC-TLS [QUIC-TLS] uses a client secret and a server secret from which it generates an AEAD key, an IV, and a header protection key for each sending direction.

An SSH key exchange produces a shared secret K, represented as an SSH multi-precision integer, and an exchange digest H, represented as binary data [RFC4253]. An SSH key exchange is parameterized with a hash function we call HASH. Note that HASH can be a different hash function, producing a different hash length, than the hash function used by the negotiated TLS cipher suite.

To compute the initial QUIC client and server secrets, the client and server encode the following binary data, which we call "secret_data":

```
mpint    K
string   H
```

Figure 18

The client and server secrets are then calculated as follows:

```
client_secret = HMAC-HASH("ssh/quic client", secret_data)
server_secret = HMAC-HASH("ssh/quic server", secret_data)
```

Figure 19

The HMAC construct is as specified in [RFC2104], instantiated using the SSH key exchange hash function, HASH.

QUIC keys and IVs are derived from these secrets using the regular QUIC-TLS key derivation process [QUIC-TLS]. Keys generated from these secrets are considered 1-RTT keys.

Clients and servers MUST implement QUIC key updates using the regular QUIC-TLS key update process [QUIC-TLS], respecting the QUIC-TLS minimum key update frequencies.

6. Adaptation of SSH to QUIC Streams

6.1. SSH/QUIC Packet Format

Each side serializes its SSH packets for sending over QUIC as follows:

```
uint32    n = payload-len, high bit set if compressed
byte[n]   payload (compressed or uncompressed)
```

Figure 20

Since security is provided by QUIC-TLS [QUIC-TLS], MAC and random padding are omitted at this stage.

The "payload-len" field has its high bit set if the "payload" field is compressed. See Section 6.1.1.

The "payload" field contains the same packet information as the "payload" field in the Binary Packet Protocol defined in [RFC4253].

6.1.1. Compression

Compression MAY be negotiated using the "delay-compression" extension in [RFC8308]. If "delay-compression" was negotiated, then:

- If compression is enabled for the server-to-client direction, the server MAY compress packets on any stream after it has sent SSH_MSG_USERAUTH_SUCCESS.

- If compression is enabled for the client-to-server direction, the client MAY compress packets on any stream after it has received SSH_MSG_USERAUTH_SUCCESS.

Due to multiple streams in SSH/QUIC, the packet SSH_MSG_NEWCOMPRESS is not an effective mechanism to signal the start of compression and MUST NOT be sent. It is replaced by the high bit in "payload-len".

6.2. Use of QUIC Streams

To avoid an unnecessary layer of flow control which has performance and complexity impacts in SSH over TCP, SSH/QUIC uses QUIC streams for SSH channels and dispenses with flow control on the level of SSH channels. This simplifies future SSH/QUIC implementations which might not implement SSH over TCP.

Conducting SSH channels over QUIC streams requires modifications of the SSH Connection Protocol [RFC4254]. The following sections describe these modifications.

6.3. Packet Sequence Numbers

In SSH over TCP, every SSH packet has an implicit sequence number which is unique for the direction of sending (to server vs. to client). The packet type SSH_MSG_UNIMPLEMENTED makes reference to this sequence number.

In SSH/QUIC, sequence numbers are separate for each sending direction, as well as each QUIC stream. This requires modification of SSH_MSG_UNIMPLEMENTED. This packet type is changed as follows:

```
byte      SSH_MSG_UNIMPLEMENTED
uint64    QUIC stream ID on which the packet was received
uint32    packet sequence number in stream, first packet = 0
```

Figure 21

6.4. Channel IDs

SSH over TCP uses 32-bit channel IDs which can be reused in the same session and do not have to be used sequentially. Conflicts in channel IDs are avoided by identifying each channel with two separate channel IDs: one designated by the sender and one by the recipient. [RFC4254]

QUIC streams use 62-bit channel IDs which cannot be reused and MUST be used sequentially. Both sides use the same stream ID. Conflicts in stream IDs are avoided by using the least significant bit to indicate whether the stream was opened by the client or by the server. [QUIC]

SSH/QUIC uses QUIC stream IDs. This requires modification of SSH channel-related packets. See Section 6.8.

6.5. Disconnection

The SSH packet type SSH_MSG_DISCONNECT is replaced by sending the QUIC frame CONNECTION_CLOSE of type 0x1d. The "Error Code" field in CONNECTION_CLOSE contains the value that would have been sent in the "reason code" in SSH_MSG_DISCONNECT. The "Reason Phrase" field in CONNECTION_CLOSE contains the value that would have been sent in "description" in SSH_MSG_DISCONNECT. The "language tag" field of SSH_MSG_DISCONNECT is not sent.

6.6. Prohibited SSH Packets

In SSH/QUIC, the following SSH packet types MUST NOT be sent:

SSH_MSG_DISCONNECT	1
SSH_MSG_NEWCOMPRESS	8
SSH_MSG_KEXINIT	20
SSH_MSG_NEWKEYS	21
key exchange packets	30-49
SSH_MSG_CHANNEL_WINDOW_ADJUST	93
SSH_MSG_CHANNEL_CLOSE	97

Figure 22

If they receive packets of these types, clients and servers MAY disconnect with SSH_DISCONNECT_PROTOCOL_ERROR (Section 6.5). Alternately, the receiver MAY send SSH_MSG_UNIMPLEMENTED (Section 6.3).

6.7. Global SSH Packets

In SSH/QUIC, the following SSH packet types MUST be sent on QUIC stream 0. With the exception of SSH_MSG_UNIMPLEMENTED (Section 6.3), these packets use the same encoded formats as in SSH over TCP:

```
SSH_MSG_IGNORE                      2
SSH_MSG_UNIMPLEMENTED               3   (Changed format!)
SSH_MSG_DEBUG                       4
SSH_MSG_SERVICE_REQUEST             5
SSH_MSG_SERVICE_ACCEPT              6
SSH_MSG_EXT_INFO                    7

SSH_MSG_USERAUTH_REQUEST           50
SSH_MSG_USERAUTH_FAILURE           51
SSH_MSG_USERAUTH_SUCCESS           52
SSH_MSG_USERAUTH_BANNER            53
SSH_MSG_USERAUTH_INFO_REQUEST      60
SSH_MSG_USERAUTH_INFO_RESPONSE     61

SSH_MSG_GLOBAL_REQUEST             80
SSH_MSG_REQUEST_SUCCESS            81
SSH_MSG_REQUEST_FAILURE            82
```

Figure 23

6.8. SSH Channel Packets

All SSH/QUIC channels MUST be opened as bidirectional QUIC streams. This means QUIC stream IDs where the least significant bits are 10 or 11 MUST NOT be used in SSH/QUIC. Implementations that receive such stream IDs MUST disconnect with SSH_DISCONNECT_PROTOCOL_ERROR (Section 6.5)

A client MUST NOT open a non-zero QUIC stream before the server has sent SSH_MSG_USERAUTH_SUCCESS. If a client does so, the server MUST disconnect with SSH_DISCONNECT_PROTOCOL_ERROR.

A server MUST NOT open a non-zero QUIC stream before it has sent SSH_MSG_USERAUTH_SUCCESS. However, a client MUST be prepared for the possibility that, due to network delays, a stream opened by the server can be received by the client before SSH_MSG_USERAUTH_SUCCESS. Therefore, if the client receives a server-initiated stream before SSH_MSG_USERAUTH_SUCCESS, it MUST assume that the server has also sent SSH_MSG_USERAUTH_SUCCESS. If the client then receives packets on QUIC stream 0 which invalidate this assumption, the client MUST disconnect with SSH_DISCONNECT_PROTOCOL_ERROR.

The initiator of any non-zero QUIC stream MUST send SSH_MSG_CHANNEL_OPEN as the first packet. If the receiver refuses the channel, it replies with SSH_MSG_CHANNEL_OPEN_FAILURE. Both sides then MUST close the QUIC stream as per Section 6.9. In this case, even though a QUIC stream was opened, an SSH channel was not. Therefore, other SSH_MSG_CHANNEL_xxxx packets MUST NOT be sent. This includes SSH_MSG_CHANNEL_EOF.

If the receiver accepts the channel, it replies with SSH_MSG_CHANNEL_OPEN_CONFIRMATION. Both sides then send SSH packets of types SSH_MSG_CHANNEL_xxxx. In SSH/QUIC, these packets have the following formats:

```
byte      SSH_MSG_CHANNEL_OPEN
string    channel type in US-ASCII only
uint32    maximum packet size
....      channel-type-specific data follows
```

Figure 24

```
byte      SSH_MSG_CHANNEL_OPEN_CONFIRMATION
uint32    maximum packet size
....      channel-type-specific data follows
```

Figure 25

```
byte      SSH_MSG_CHANNEL_OPEN_FAILURE
uint32    reason code
string    description in UTF-8
string    language tag
```

Figure 26

```
byte      SSH_MSG_CHANNEL_DATA
string    data
```

Figure 27

```
byte      SSH_MSG_CHANNEL_EXTENDED_DATA
uint32    data_type_code
string    data
```

Figure 28

```
byte      SSH_MSG_CHANNEL_EOF
```

Figure 29

```
byte      SSH_MSG_CHANNEL_REQUEST
string    request type in US-ASCII characters only
boolean   want reply
....      type-specific data follows
```

Figure 30

```
byte      SSH_MSG_CHANNEL_SUCCESS
```

Figure 31

```
byte      SSH_MSG_CHANNEL_FAILURE
```

Figure 32

6.9. Closing a Channel

The SSH packet type SSH_MSG_CHANNEL_CLOSE is replaced by QUIC stream state transitions [QUIC]. Each side considers a channel closed when the QUIC stream is both in a terminal sending state, and a terminal receiving state. This means:

- The QUIC sending stream state has become "Data Recvd" or "Reset Recvd".
- The QUIC receiving stream state has become "Data Read" or "Reset Read".

The SSH packet type SSH_MSG_CHANNEL_EOF continues to be used. This packet often does NOT correspond with the end of the stream in its direction. As in SSH over TCP, SSH channel requests MAY be sent after SSH_MSG_CHANNEL_EOF, and MUST be handled gracefully by receivers. A common example is the request "exit-status", which is sent by a server to communicate a process exit code to the SSH client, and is commonly sent after the end of output.

7. Acknowledgements

Paul Ebermann for first review and the encouragement to use QUIC streams.

Ilari Liusvaara for "server-name-indication" and value 1200 for SSH_QUIC_INIT padding target.

8. IANA Considerations

This document requests no changes to IANA registries.

9. Security Considerations

Clients and servers MUST insert into SSH_QUIC_INIT and SSH_QUIC_REPLY at least the minimum amount of cryptographically random data as specified in the section Random Elements. Compromising on this requirement reduces the security of any session created on the basis of such an SSH_QUIC_INIT or SSH_QUIC_REPLY.

10. References

10.1. Normative References

[QUIC]

Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", 2020, <https://tools.ietf.org/html/draft-ietf-quic-transport-29>.

[QUIC-TLS] Thomson, M. and S. Turner, "Using TLS to Secure QUIC", 2020, <https://tools.ietf.org/html/draft-ietf-quic-tls-29>.

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, DOI 10.17487/RFC2104, February 1997, <https://www.rfc-editor.org/info/rfc2104>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC4251] Ylonen, T. and C. Lonvick, Ed., "The Secure Shell (SSH) Protocol Architecture", RFC 4251, DOI 10.17487/RFC4251, January 2006, <https://www.rfc-editor.org/info/rfc4251>.

[RFC4253] Ylonen, T. and C. Lonvick, Ed., "The Secure Shell (SSH) Transport Layer Protocol", RFC 4253, DOI 10.17487/RFC4253, January 2006, <https://www.rfc-editor.org/info/rfc4253>.

[RFC5116] McGrew, D., "An Interface and Algorithms for Authenticated Encryption", RFC 5116, DOI 10.17487/RFC5116, January 2008, <https://www.rfc-editor.org/info/rfc5116>.

[RFC5656] Stebila, D. and J. Green, "Elliptic Curve Algorithm Integration in the Secure Shell Transport Layer", RFC

5656, DOI 10.17487/RFC5656, December 2009, <https://www.rfc-editor.org/info/rfc5656>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8308] Bider, D., "Extension Negotiation in the Secure Shell (SSH) Protocol", RFC 8308, DOI 10.17487/RFC8308, March 2018, <https://www.rfc-editor.org/info/rfc8308>.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/info/rfc8446>.

[RFC8731] Adamantiadis, A., Josefsson, S., and M. Baushke, "Secure Shell (SSH) Key Exchange Method Using Curve25519 and Curve448", RFC 8731, DOI 10.17487/RFC8731, February 2020, <https://www.rfc-editor.org/info/rfc8731>.

10.2. Informative References

[IANA-SSH] IANA, "Secure Shell (SSH) Protocol Parameters", , <https://www.iana.org/assignments/ssh-parameters/>.

[RFC4250] Lehtinen, S. and C. Lonvick, Ed., "The Secure Shell (SSH) Protocol Assigned Numbers", RFC 4250, DOI 10.17487/RFC4250, January 2006, <https://www.rfc-editor.org/info/rfc4250>.

[RFC4252] Ylonen, T. and C. Lonvick, Ed., "The Secure Shell (SSH) Authentication Protocol", RFC 4252, DOI 10.17487/RFC4252, January 2006, <https://www.rfc-editor.org/info/rfc4252>.

[RFC4254] Ylonen, T. and C. Lonvick, Ed., "The Secure Shell (SSH) Connection Protocol", RFC 4254, DOI 10.17487/RFC4254, January 2006, <https://www.rfc-editor.org/info/rfc4254>.

[RFC4462] Hutzelman, J., Salowey, J., Galbraith, J., and V. Welch, "Generic Security Service Application Program Interface (GSS-API) Authentication and Key Exchange for the Secure Shell (SSH) Protocol", RFC 4462, DOI 10.17487/RFC4462, May 2006, <https://www.rfc-editor.org/info/rfc4462>.

[RFC8268] Baushke, M., "More Modular Exponentiation (MODP) Diffie-Hellman (DH) Key Exchange (KEX) Groups for Secure Shell (SSH)", RFC 8268, DOI 10.17487/RFC8268, December 2017, <https://www.rfc-editor.org/info/rfc8268>.

Appendix A. Generating Random Lengths

The SSH/QUIC extensibility mechanism calls for generating random lengths such that values in the shorter end of the range are significantly more probable, but long lengths are still selected. The following C example shows a simple two-step process to prefer shorter lengths:

```
 int RandomIntBetweenZeroAnd(int maxValueInclusive);

 int RandomLen_PreferShort(int minLen, int maxLen)
 {
     int const SPAN_THRESHOLD = 7;
     int lenSpan = maxLen - minLen;

     if (lenSpan <= 0)
         return minLen;

     if (lenSpan > SPAN_THRESHOLD)
         if (0 != RandomIntBetweenZeroAnd(3))
             return minLen + RandomIntBetweenZeroAnd(SPAN_THRESHOLD);

     return minLen + RandomIntBetweenZeroAnd(lenSpan);
 }
```

Figure 33

Author's Address

denis bider
Bitvise Limited
4105 Lombardy Ct
Colleyville, TX 76034
United States

Email: ietf-draft@denisbider.com