Internet Engineering Task Force
Internet-Draft
Intended status: Informational
Expires: July 23, 2020

A. Malhotra
Boston University
A. Langley
Google
W. Ladd
Cloudflare

January 20, 2020

Roughtime
draft-roughtime-aanchal-04

Abstract

This document specifies Roughtime - a protocol that aims to achieve rough time synchronization while detecting servers that provide inaccurate time and providing cryptographic proof of their malfeasance.

Status of This Memo

This Internet-Draft will expire on July 23, 2020.

Copyright Notice

Table of Contents

1. Introduction

Time synchronization is essential to Internet security as many security protocols and other applications require synchronization [RFC7384] [MCBG]. Unfortunately widely deployed protocols such as the Network Time Protocol (NTP) [RFC5905] lack essential security features, and even newer protocols like Network Time Security (NTS) [I-D.ietf-ntp-using-nts-for-ntp] fail to ensure that the servers behave correctly. Authenticating time servers prevents network

adversaries from modifying time packets, but an authenticated time server still has full control over the contents of the time packet and may go rogue. The Roughtime protocol provides cryptographic proof of malfeasance, enabling clients to detect and prove to a third party a server's attempts to influence the time a client computes.

Protocol	Authenticated Server	Server Malfeasance Evidence
NTP, Chronos	N	N
NTP-MD5	Y*	N
NTP-Autokey	Y**	N
NTS	Y	N
Roughtime	Y	Y

Security Properties of current protocols

Table 1

Y* For security issues with symmetric-key based NTP-MD5 authentication, please refer to RFC 8573 [RFC8573].

Y** For security issues with Autokey Public Key Authentication, refer to [Autokey].

More specifically,

- If a server's timestamps do not fit into the time context of other servers' responses, then a Roughtime client can cryptographically prove this misbehavior to third parties. This helps detect "bad" servers.

- A Roughtime client can roughly detect (with no absolute guarantee) a delay attack [DelayAttacks] but can not cryptographically prove this to a third party. However, the absence of proof of malfeasance should not be considered a proof of absence of malfeasance. So Roughtime should not be used as a witness that a server is overall "good".

- Note that delay attacks cannot be detected/stopped by any protocol. Delay attacks can not, however, undermine the security guarantees provided by Roughtime.

- Although delay attacks cannot be prevented, they can be limited to a predetermined upper bound. This can be done by defining a maximal tolerable Round Trip Time (RTT) value, MAX-RTT, that a Roughtime client is willing to accept. A Roughtime client can

measure the RTT of every request-response handshake and compare it to MAX-RTT. If the RTT exceeds MAX-RTT, the corresponding server is assumed to be a falseticker. When this approach is used the maximal time error that can be caused by a delay attack is MAX-RTT/2. It should be noted that this approach assumes that the nature of the system is known to the client, including reasonable upper bounds on the RTT value.

2. Requirements Language

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.

3. Protocol Overview

Roughtime is a protocol for rough time synchronization that enables clients to provide cryptographic proof of server malfeasance. It does so by having responses from servers include a signature with a certificate rooted in a long-term public/private key pair over a value derived from a nonce provided by the client in its request. This provides cryptographic proof that the timestamp was issued after the server received the client's request. The derived value included in the server's response is the root of a Merkle tree which includes the hash of the client's nonce as the value of one of its leaf nodes. This enables the server to amortize the relatively costly signing operation over a number of client requests.

Single server mode: At its most basic level, Roughtime is a one round protocol in which a completely fresh client requests the current time and the server sends a signed response. The response includes a timestamp and a radius used to indicate the server's certainty about the reported time. For example, a radius of 1,000,000 microseconds means the server is absolutely confident that the true time is within one second of the reported time.

The server proves freshness of its response as follows: The client's request contains a nonce. The server incorporates the nonce into its signed response so that the client can verify the server's signatures covering the nonce issued by the client. Provided that the nonce has sufficient entropy, this proves that the signed response could only have been generated after the nonce.

Chaining multiple servers: For subsequent requests, the client generates a new nonce by hashing the reply from the previous server with a random value (a blind). This proves that the nonce was

created after the reply from the previous server. It sends the new nonce in a request to the next server and receives a response that includes a signature covering the nonce.

Cryptographic proof of misbehavior: If the time from the second server is before the first, then the client has proof that at least one of the servers is misbehaving; the reply from the second server implicitly shows that it was created later because of the way that the client constructed the nonce. If the time from the second server is too far in the future, the client can contact the first server again with a new nonce generated from the second server's response and get a signature that was provably created afterwards, but with an earlier timestamp.

With only two servers, the client can end up with proof that something is wrong, but no idea what the correct time is. But with half a dozen or more independent servers, the client will end up with chain of proof of any server's misbehavior, signed by several others, and (presumably) enough accurate replies to establish what the correct time is. Furthermore, this proof may be validated by third parties ultimately leading to a revocation of trust in the misbehaving server.

4. The guarantee

A Roughtime server guarantees that a response to a query sent at t_1, received at t_2, and with timestamp t_3 has been created between the transmission of the query and its reception. If t_3 is not within that interval, a server inconsistency may be detected and used to impeach the server. The propagation of such a guarantee and its use of type synchronization is discussed in Section 7. No delay attacker may affect this: they may only expand the interval between t_1 and t_2, or of course stop the measurement in the first place.

5. Message Format

Roughtime messages are maps consisting of one or more (tag, value) pairs. They start with a header, which contains the number of pairs, the tags, and value offsets. The header is followed by a message values section which contains the values associated with the tags in the header. Messages MUST be formatted according to Figure 1 as described in the following sections.

Messages may be recursive, i.e. the value of a tag can itself be a Roughtime message.

```
    0                   1                   2                   3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                   Number of pairs (uint32)                    |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                                                               |
   .                                                               .
   .                     N-1 offsets (uint32)                      .
   .                                                               .
   |                                                               |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                                                               |
   .                                                               .
   .                        N tags (uint32)                        .
   .                                                               .
   |                                                               |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                                                               |
   .                                                               .
   .                            Values                             .
   .                                                               .
   |                                                               |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 1: Roughtime Message Format

5.1. Data Types

5.1.1. uint32

A uint32 is a 32 bit unsigned integer. It is serialized with the least significant byte first.

5.1.2. uint64

A uint64 is a 64 bit unsigned integer. It is serialized with the least significant byte first.

5.1.3. Tag

Tags are used to identify values in Roughtime packets. A tag is a uint32 but may also be listed as a sequence of up to four ASCII characters [RFC0020]. ASCII strings shorter than four characters can be unambiguously converted to tags by padding them with zero bytes. For example, the ASCII string "NONC" would correspond to the tag 0x434e4f4e and "PAD" would correspond to 0x00444150.

5.1.4. Timestamp

A timestamp is a uint64 interpreted in the following way. The most significant 3 bytes contain the integer part of a Modified Julian Date (MJD). The least significant 5 bytes is a count of the number of Coordinated Universal Time (UTC) microseconds [ITU-R_TF.460-6] since midnight on that day.

The MJD is the number of UTC days since 17 November 1858 [ITU-R_TF.457-2].

Note that, unlike NTP, this representation does not use the full number of bits in the fractional part and that days with leap seconds will have more or fewer than the nominal 86,400,000,000 microseconds.

5.2. Header

All Roughtime messages start with a header. The first four bytes of the header is the uint32 number of tags N, and hence of (tag, value) pairs. The following 4*(N-1) bytes are offsets, each a uint32. The last 4*N bytes in the header are tags.

Offsets refer to the positions of the values in the message values section. All offsets MUST be multiples of four and placed in increasing order. The first post-header byte is at offset 0. The offset array is considered to have a not explicitly encoded value of 0 as its zeroth entry. The value associated with the ith tag begins at offset[i] and ends at offset[i+1]-1, with the exception of the last value which ends at the end of the packet. Values may have zero length.

Tags MUST be listed in the same order as the offsets of their values. A tag MUST NOT appear more than once in a header.

6. Protocol

Roughtime messages are sent between clients and servers as UDP packets, or over TCP. When transporting over TCP, the packets are prefixed with their length as a uint32. Currently no servers exist for the TCP version. As described in Section 3, clients initiate time synchronization by sending request packets containing a nonce to servers who send signed time responses in return.

6.1. Requests

A request is a Roughtime message with the tag NONC. The size of the request message SHOULD be at least 1024 bytes. To attain this size the PAD tag SHOULD be added to the message. Tags other than NONC

SHOULD be ignored by the server. Responding to requests shorter than 1024 bytes is OPTIONAL and servers MUST NOT send responses larger than the requests they are replying to.

The value of the NONC tag is a 64 byte nonce. It SHOULD be generated by hashing a previous Roughtime response message together with a blind as described in Section 8. If no previous responses are avaiable to the client, the nonce SHOULD be generated at random.

The PAD tag SHOULD be used by clients to ensure their request messages are at least 1024 bytes in size. Its value SHOULD be all zeros.

6.2. Responses

A response contains the tags SREP, SIG, CERT, INDX, and PATH. The SIG tag is a signature over the SREP value using the public key contained in CERT, as explained below.

The SREP tag contains a time response. Its value is a Roughtime message with the tags ROOT, MIDP, and RADI.

The ROOT tag contains a 32 byte value of a Merkle tree root as described in Section 6.3.

The MIDP tag value is a timestamp of the moment of processing.

The RADI tag value is a uint32 representing the server's estimate of the accuracy of MIDP in microseconds. Servers MUST ensure that the true time is within (MIDP-RADI, MIDP+RADI) at the time they compose the response packet.

The SIG tag value is a 64 byte Ed25519 signature [RFC8032] over a signature context concatenated with the entire value of a DELE or SREP tag. Signatures of DELE tags use the ASCII string "RoughTime v1 delegation signature--" and signatures of SREP tags use the ASCII string "RoughTime v1 response signature" as signature context. Both strings include a terminating zero byte.

The CERT tag contains a public-key certificate signed with the server's long-term key. Its value is a Roughtime message with the tags DELE and SIG, where SIG is a signature over the DELE value.

The DELE tag contains a delegated public-key certificate used by the server to sign the SREP tag. Its value is a Roughtime message with the tags MINT, MAXT, and PUBK. The purpose of the DELE tag is to enable separation of a long-term public key from keys on devices exposed to the public Internet.

The MINT tag is the minimum timestamp for which the key in PUBK is trusted to sign responses. MIDP MUST be more than or equal to MINT for a response to be considered valid.

The MAXT tag is the maximum timestamp for which the key in PUBK is trusted to sign responses. MIDP MUST be less than or equal to MAXT for a response to be considered valid.

The PUBK tag contains a temporary 32 byte Ed25519 public key which is used to sign the SREP tag.

The INDX tag value is a uint32 determining the position of NONC in the Merkle tree used to generate the ROOT value as described in Section 6.3.

The PATH tag value is a multiple of 32 bytes long and represents a path of 32 byte hash values in the Merkle tree used to generate the ROOT value as described in Section 6.3. In the case where a response is prepared for a single request and the Merkle tree contains only the root node, the size of PATH is zero.

6.3. The Merkle Tree

A Merkle tree is a binary tree where the value of each non-leaf node is a hash value derived from its two children. The root of the tree is thus dependent on all leaf nodes.

In Roughtime, each leaf node in the Merkle tree represents the nonce of one request that a response message is sent in reply to. Leaf nodes are indexed left to right, beginning with zero.

The values of all nodes are calculated from the leaf nodes and up towards the root node using the first 32 bytes of the output of the SHA-512 hash algorithm [RFC6234]. For leaf nodes, the byte 0x00 is prepended to the nonce before applying the hash function. For all other nodes, the byte 0x01 is concatenated with first the left and then the right child node value before applying the hash function.

The value of the Merkle tree's root node is included in the ROOT tag of the response.

The index of a request's nonce node is included in the INDX tag of the response.

The values of all sibling nodes in the path between a request's nonce node and the root node is stored in the PATH tag so that the client can reconstruct and validate the value in the ROOT tag using its nonce.

6.3.1. Root value validity check algorithm

One starts by computing the hash of the NONC value from the request, with 0x00 prepended. Then one walks from the least significant bit of INDX to the most significant bit, and also walks towards the end of PATH.

If PATH ends then the remaining bits of the INDX MUST be all zero. This indicates the termination of the walk, and the current value MUST equal ROOT if the response is valid.

If the current bit is 0, one hashes 0x01, the current hash, and the value from PATH to derive the next current value.

If the current bit is 1 one hashes 0x01, the value from PATH, and the current hash to derive the next current value.

6.4. Validity of response

A client MUST check the following properties when it receives a response. We assume the long-term server public key is known to the client through other means.

- The signature in CERT was made with the long-term key of the server.

- The DELE timestamps and the MIDP value are consistent.

- The INDX and PATH values prove NONC was included in the Merkle tree with value ROOT using the algorithm in Section 6.3.1.

- The signature of SREP in SIG validates with the public key in DELE.

A response that passes these checks is said to be valid. Validity of a response does not prove the time is correct, but merely that the server signed it, and thus guarantees that it began to compute the signature at a time in the interval (MIDP-RADI, MIDP+RADI).

7. Integration into ntp

We assume that there is a bound PHI on the frequency error in the clock on the machine. Given a measurement taken at a local time t1, we know the true time is in [t1-delta-sigma, t1-delta+sigma]. After d seconds have elapsed we know the true time is within [t1-delta-sigma-d*PHI, t1-delta+sigma+d*PHI]. A simple and effective way to mix with NTP or PTP discipline of the clock is to trim the observed intervals in NTP to fit entirely within this window or

reject measurements that fall to far outside. This assumes time has not been stepped. If the NTP process decides to step the time, it MUST use roughtime to ensure the new truetime estimate that will be stepped to is consistent with the true time.

Should this window become too large, another roughtime measurement is called for. The definition of "too large" is implementation defined.

Implementations MAY use other, more sophisticated means of adjusting the clock respecting roughtime information.

8. Cheater Detection

A chain of responses is a series of responses where the SHA-512 hash of the preceding response H, is concatenated with a 64 byte blind X, and then SHA-512(H, X) is the nonce used in the subsequent response. These may be represented as an array of objects in JavaScript Object Notation (JSON) format [RFC8259] where each object may have keys "blind" and "response_packet". Packet has the Base64 [RFC4648] encoded bytes of the packet and blind is the Base64 encoded blind used for the next nonce. The last packet needs no blind.

A pair of responses (r_1, r_2) is invalid if MIDP_1-RADI_1 > MIDP_2+RADI_2. A chain of longer length is invalid if for any i, j such that i < j, (r_i, r_j) is an invalid pair.

Invalidity of a chain is proof that causality has been violated if all servers were reporting correct time. An invalid chain where all individual responses are valid is cryptographic proof of malfeasance of at least one server: if all servers had the correct time in the chain, causality would imply that MIDP_1-RADI_1 < MIDP_2+RADI_2.

In conducting the comparison of timestamps one must know the length of a day and hence have historical leap second data for the days in question. However if violations are greater then a second the loss of leap second data doesn't impede their detection.

9. Grease

Servers MAY send back a fraction of responses that are syntactically invalid or contain invalid signatures as well as incorrect times. Clients MUST properly reject such responses. Servers MUST NOT send back responses with incorrect times and valid signatures. Either signature MAY be invalid for this application.

10. Roughtime Servers

The below list contains a list of servers with their public keys in Base64 format. These servers may implement older versions of this specification.

address: roughtime.cloudflare.com
port: 2002
long-term key: gD63hSj3ScS+wuOeGrubXlq35N1c5Lby/S+T7MNTjxo=

address: roughtime.int08h.com
port: 2002
long-term key: AW5uAoTSTDfG5NfY1bTh08GUnOqlRb+HVhbJ3ODJvsE=

address: roughtime.sandbox.google.com
port: 2002
long-term key: etPaaIxcBMY1oUeGpwvPMCJMwlRVNxv51KK/tktoJTQ=

address: roughtime.se
port: 2002
long-term key: S3AzfZJ5CjSdkJ21ZJGbxqdYP/SoE8fXKY0+aicsehI=

11. Trust anchors and policies

A trust anchor is any distributor of a list of trusted servers. It is RECOMMENDED that trust anchors subscribe to a common public forum where evidence of malfeasance may be shared and discussed. Trust anchors SHOULD subscribe to a zero-tolerance policy: any generation of incorrect timestamps will result in removal. To enable this trust anchors SHOULD list a wide variety of servers so the removal of a server does not result in operational issues for clients. Clients SHOULD attempt to detect malfeasance and have a way to report it to trust anchors.

Because only a single roughtime server is required for successful synchronization, Roughtime does not have the incentive problems that have prevented effective enforcement of discipline on the web PKI. We expect that some clients will aggressively monitor server behavior.

12. Acknowledgements

Thomas Peterson corrected multiple nits. Marcus Dansarie, Peter Loethberg (Lothberg), Tal Mizrahi, Ragnar Sundblad, Kristof Teichel, and the other members of the NTP working group contributed comments and suggestions.

13. IANA Considerations

13.1. Service Name and Transport Protocol Port Number Registry

IANA is requested to allocate the following entry in the Service Name and Transport Protocol Port Number Registry [RFC6335]:

Service Name: Roughtime

Transport Protocol: udp

Assignee: IESG <iesg@ietf.org>

Contact: IETF Chair <chair@ietf.org>

Description: Roughtime time synchronization

Reference: [[this memo]]

Port Number: [[TBD1]], selected by IANA from the User Port range

13.2. Roughtime Tag Registry

IANA is requested to create a new registry entitled "Roughtime Tag Registry". Entries SHALL have the following fields:

Tag (REQUIRED): A 32-bit unsigned integer in hexadecimal format.

ASCII Representation (OPTIONAL): The ASCII representation of the tag in accordance with Section 5.1.3 of this memo, if applicable.

Reference (REQUIRED): A reference to a relevant specification document.

The policy for allocation of new entries in this registry SHOULD be: Specification Required.

The initial contents of this registry SHALL be as follows:

Tag	ASCII Representation	Reference
0x00444150	PAD	[[this memo]]
0x00474953	SIG	[[this memo]]
0x434e4f48	NONC	[[this memo]]
0x454c4544	DELE	[[this memo]]
0x48544150	PATH	[[this memo]]
0x49444152	RADI	[[this memo]]
0x4b425550	PUBK	[[this memo]]
0x5044494d	MIDP	[[this memo]]
0x50455253	SREP	[[this memo]]
0x544e494d	MINT	[[this memo]]
0x544f4f52	ROOT	[[this memo]]
0x54524543	CERT	[[this memo]]
0x5458414d	MAXT	[[this memo]]
0x58444e49	INDX	[[this memo]]

14. Security Considerations

Since the only supported signature scheme, Ed25519, is not quantum resistant, this protocol will not survive the advent of quantum computers.

Maintaining a list of trusted servers and adjudicating violations of the rules by servers is not discussed in this document and is essential for security. Roughtime clients MUST update their view of which servers are trustworthy in order to benefit from the detection of misbehavior.

Validating timestamps made on different dates requires knowledge of leap seconds in order to calculate time intervals correctly.

Servers carry out a significant amount of computation in response to clients, and thus may experience vulnerability to denial of service attacks.

This protocol does not provide any confidentiality, and given the nature of timestamps such impact is minor.

The compromise of a PUBK's private key, even past MAXT, is a problem as the private key can be used to sign invalid times that are in the range MINT to MAXT, and thus violate the good behavior guarantee of the server.

Servers MUST NOT send response packets larger than the request packets sent by clients, in order to prevent amplification attacks.

15. Privacy Considerations

This protocol is designed to obscure all client identifiers. Servers necessarily have persistent long-term identities essential to enforcing correct behavior. Generating nonces from previous responses without using a blind can enable tracking of clients as they move between networks.

16. References

16.1. Normative References

[ITU-R_TF.457-2]
ITU-R, "Use of the Modified Julian Date by the Standard-Frequency and Time-Signal Services", ITU-R Recommendation TF.457-2, October 1997.

[ITU-R_TF.460-6]
ITU-R, "Standard-Frequency and Time-Signal Emissions", ITU-R Recommendation TF.460-6, February 2002.

[RFC0020] Cerf, V., "ASCII format for network interchange", STD 80, RFC 20, DOI 10.17487/RFC0020, October 1969, <https://www.rfc-editor.org/info/rfc20>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <https://www.rfc-editor.org/info/rfc4648>.

[RFC6234] Eastlake 3rd, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234, DOI 10.17487/RFC6234, May 2011, <https://www.rfc-editor.org/info/rfc6234>.

[RFC6335] Cotton, M., Eggert, L., Touch, J., Westerlund, M., and S. Cheshire, "Internet Assigned Numbers Authority (IANA) Procedures for the Management of the Service Name and Transport Protocol Port Number Registry", BCP 165, RFC 6335, DOI 10.17487/RFC6335, August 2011, <https://www.rfc-editor.org/info/rfc6335>.

[RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, DOI 10.17487/RFC8032, January 2017, <https://www.rfc-editor.org/info/rfc8032>.

[RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <https://www.rfc-editor.org/info/rfc8259>.

16.2. Informative References

[Autokey] Rottger, S., "Analysis of the NTP Autokey Procedures", 2012, <https://zero-entropy.de/autokey_analysis.pdf>.

[DelayAttacks] Mizrahi, T., "A Game Theoretic Analysis of Delay Attacks Against Time Synchronization Protocols", DOI 10.1109/ISPCS.2012.6336612, 2012, <https://ieeexplore.ieee.org/document/6336612>.

[I-D.ietf-ntp-using-nts-for-ntp] Franke, D., Sibold, D., Teichel, K., Dansarie, M., and R. Sundblad, "Network Time Security for the Network Time Protocol", draft-ietf-ntp-using-nts-for-ntp-20 (work in progress), July 2019.

[MCBG] Malhotra, A., Cohen, I., Brakke, E., and S. Goldberg, "Attacking the Network Time Protocol", 2015, <https://eprint.iacr.org/2015/1020>.

[RFC0768] Postel, J., "User Datagram Protocol", STD 6, RFC 768, DOI 10.17487/RFC0768, August 1980, <https://www.rfc-editor.org/info/rfc768>.

[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>.

[RFC5905] Mills, D., Martin, J., Ed., Burbank, J., and W. Kasch, "Network Time Protocol Version 4: Protocol and Algorithms Specification", RFC 5905, DOI 10.17487/RFC5905, June 2010, <https://www.rfc-editor.org/info/rfc5905>.

[RFC7384] Mizrahi, T., "Security Requirements of Time Protocols in Packet Switched Networks", RFC 7384, DOI 10.17487/RFC7384, October 2014, <https://www.rfc-editor.org/info/rfc7384>.

[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>.

[RFC8573] Malhotra, A. and S. Goldberg, "Message Authentication Code for the Network Time Protocol", RFC 8573, DOI 10.17487/RFC8573, June 2019, <https://www.rfc-editor.org/info/rfc8573>.

Appendix A. Terms and Abbreviations

ASCII	American Standard Code for Information Interchange
IANA	Internet Assigned Numbers Authority
JSON	JavaScript Object Notation [RFC8259]
MJD	Modified Julian Date
NTP	Network Time Protocol [RFC5905]
NTS	Network Time Security [I-D.ietf-ntp-using-nts-for-ntp]
UDP	User Datagram Protocol [RFC0768]
UTC	Coordinated Universal Time [ITU-R_TF.460-6]

Authors' Addresses

Aanchal Malhotra
Boston University
111 Cummington Mall
Boston 02215
USA

Email: aanchal4@bu.edu

Adam Langley
Google

Email:
agl@google.com

Watson Ladd
Cloudflare
101 Townsend St
San Francisco
USA

Email: watsonbladd@gmail.com