
Stream: Internet Engineering Task Force (IETF)
RFC: [10049](#)
Category: Experimental
Published: October 2026
ISSN: 2070-1721
Authors: W. Ladd M. Dansarie
Akamai Technologies Netnod

RFC 10049

Roughtime: A Protocol for Rough Time Synchronization

Abstract

This document describes Roughtime, an experimental protocol that aims to achieve two things: secure, rough time synchronization (even for clients without any idea of what time it is) and a format for clients to report any inconsistencies they observe between timeservers. This document specifies the on-wire protocol required for these goals and discusses aspects of the ecosystem needed for it to work.

Status of This Memo

This document is not an Internet Standards Track specification; it is published for examination, experimental implementation, and evaluation.

This document defines an Experimental Protocol for the Internet community. This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Not all documents approved by the IESG are candidates for any level of Internet Standard; see Section 2 of RFC 7841.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at <https://www.rfc-editor.org/info/rfc10049>.

Copyright Notice

Copyright (c) 2026 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 (<https://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 Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction	4
2. Conventions	5
3. Protocol Overview	5
3.1. Single Server Mode	5
3.2. Multi-server Mode	5
4. Message Format	6
4.1. Data Types	6
4.1.1. uint32	6
4.1.2. uint64	6
4.1.3. Tag	7
4.1.4. Timestamp	7
4.2. Header	7
5. Protocol Details	7
5.1. Requests	9
5.1.1. VER	9
5.1.2. NONC	9
5.1.3. TYPE	9
5.1.4. SRV	9
5.1.5. ZZZZ	10
5.2. Responses	10
5.2.1. SIG	11
5.2.2. NONC	11
5.2.3. TYPE	11
5.2.4. PATH	11
5.2.5. SREP	11
5.2.6. CERT	12

5.2.7. INDX	12
5.3. The Merkle Tree	12
5.3.1. Root Value Validity Check Algorithm	13
5.4. Validity of Response	13
6. Integration into NTP	13
7. Grease	14
8. RoughTime Clients	14
8.1. Necessary Configuration	14
8.2. Measurement Sequence	14
8.3. Server Lists	15
8.4. Malfeasance Reporting	16
8.4.1. Malfeasance Report Format	16
8.4.2. Reporting	16
9. Security Considerations	17
9.1. Confidentiality	17
9.2. Integrity and Authenticity	17
9.3. Generating Private Keys	17
9.4. Private Key Compromise	17
9.5. Quantum Resistance	17
9.6. Maintaining Lists of Servers	17
9.7. Amplification Attacks	18
10. Privacy Considerations	18
11. Operational Considerations	18
12. IANA Considerations	18
12.1. Service Name and Transport Protocol Port Number Registry	18
12.2. RoughTime Versions Registry	19
12.3. RoughTime Tags Registry	19
12.4. Media Type Registry	20
12.4.1. Media Type for RoughTime Server List	20
12.4.2. Media Type for RoughTime Malfeasance	21

13. References	22
13.1. Normative References	22
13.2. Informative References	23
Appendix A. Example Server List	24
Appendix B. Example Malfeasance Report	25
Acknowledgments	27
Authors' Addresses	28

1. Introduction

Time synchronization is essential to Internet security as many security protocols and other applications require it [RFC0738]. 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) [RFC8915] lack mechanisms to observe that the servers behave correctly. Furthermore, clients may lack even a basic idea of the time, creating bootstrapping problems as time is required for X.509 certificate validation.

The primary design goal of RoughTime is to permit devices to obtain a rough idea of the current time from a fairly static configuration and to enable them to report any inconsistencies they observe between servers. The configuration consists of a list of servers and their associated long-term keys, which ideally remain unchanged throughout a server's lifetime. This makes the long-term public keys the roots of trust in RoughTime. With a sufficiently long list of trusted servers and keys, a client will be able to acquire authenticated time with high probability, even after long periods of inactivity. Proofs of malfeasance constructed by chaining together responses from different trusted servers can be used to prove misbehavior by a server and, after analysis, can result in revoking trust in that particular key.

Unlike Khronos [RFC9523], RoughTime produces external evidence that time servers are reporting incompatible times. This requires changes to the format of the timestamps and hence cannot be a mere extension to NTP.

Operational experience is needed to evaluate the viability of using RoughTime for secure time bootstrapping in Internet-connected systems. This includes the need for experience with maintaining a RoughTime ecosystem with services that maintain and distribute lists of trusted servers and process malfeasance reports. To facilitate the experiments necessary to gain that experience, this document is limited to describing the RoughTime on-wire protocol. Apart from describing the server list and malfeasance report formats, this document does not describe the ecosystem, nor the means by which the server list is maintained and distributed or the policies to apply to such a list.

2. Conventions

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 authenticated, rough time synchronization that enables clients to provide cryptographic proof of server malfeasance. It does so by having responses from servers include a signature over a value derived from the client's request, which includes a nonce. This provides cryptographic proof that the response 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 [Merkle] that includes the hash value of the client's request as the value of one of its leaf nodes. This tree enables the server to amortize the relatively costly signing operation over a number of client requests.

3.1. 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.

The client's request contains a nonce that the server incorporates into its signed response. The client can verify the server's signatures and -- provided that the nonce has sufficient entropy -- this proves that the signed response could only have been generated after the nonce.

3.2. Multi-server Mode

When using multiple servers, a client can detect, cryptographically prove, and report inconsistencies between different servers.

A Roughtime server guarantees that the timestamp included in the response to a request is generated after the reception of the request and prior to the transmission of the associated response. If the time response from a server is not consistent with time responses from other servers, this indicates server error or intentional malfeasance that can be reported and potentially used to impeach the server.

Proofs of malfeasance are constructed by chaining requests to different Roughtime servers. Details on proofs and malfeasance reporting are provided in [Section 8](#). For the reporting to result in impeachment, an additional mechanism is required that provides a review and impeachment process. Defining such a mechanism is beyond the scope of this document. A simple option could be an online forum where a court of human observers evaluate cases after reviewing input reports.

4. 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 value offsets, and the tags. The header is followed by a message values section, which contains the values associated with the tags in the header. Messages are formatted according to [Figure 1](#) as described in the following subsections.

In some cases, messages are recursive, i.e., the value of a tag can itself be a Roughtime message.

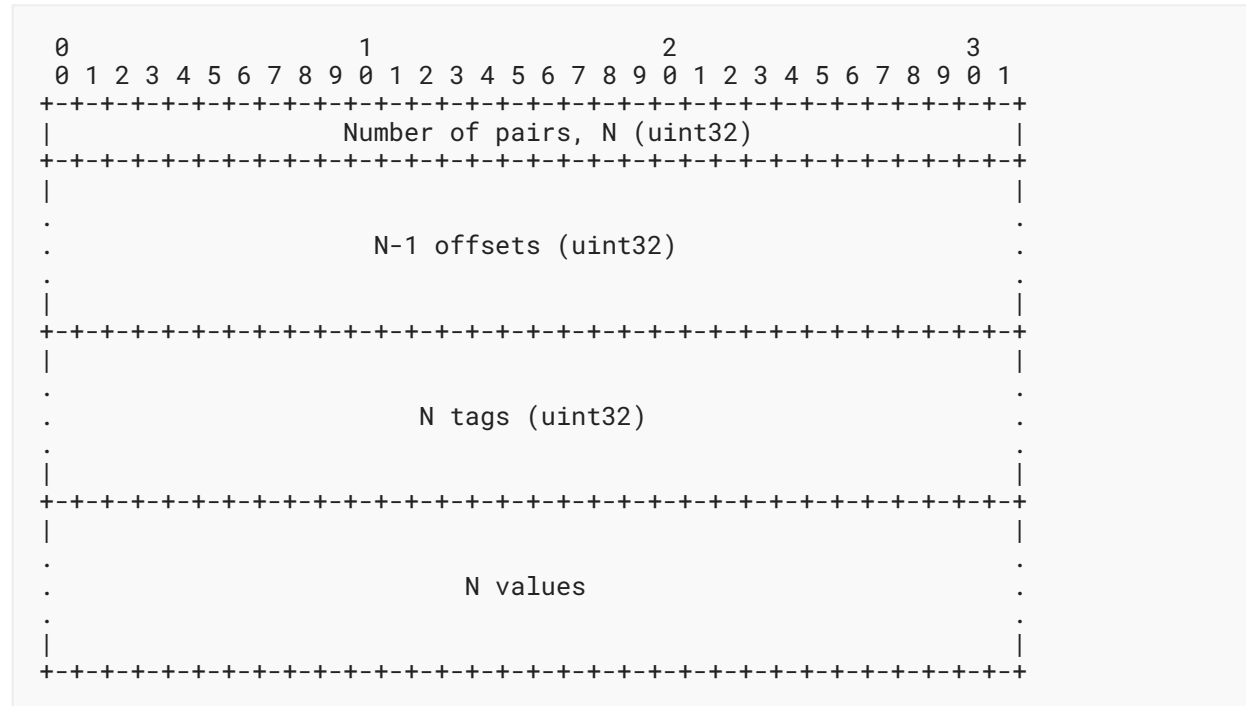


Figure 1: Roughtime Message

4.1. Data Types

4.1.1. uint32

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

4.1.2. uint64

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

4.1.3. Tag

Tags are used to identify values in Roughtime messages. A tag is a sequence of four octets. Each tag sequence starts with one to four capital ASCII letters (A-Z) [RFC0020] followed by zero to three padding zero octets. Throughout this document, tags are referred to by their ASCII string representation. However, they are registered and sorted as uint32 values, where the least significant byte is the first octet in the sequence.

For example, the ASCII string "NONC" would correspond to the uint32 0x434e4f4e, which is serialized as {0x4e, 0x4f, 0x4e, 0x43}. "VER" would correspond to 0x00524556 and be serialized as {0x56, 0x45, 0x52, 0x00}.

4.1.4. Timestamp

A timestamp is a representation of UTC time as a uint64 count of seconds since 00:00:00 on 1 January 1970 (the Unix epoch), assuming every day has 86400 seconds. This is a constant offset from the NTP timestamp in seconds. Leap seconds do not have an unambiguous representation in a timestamp, and this has implications for the attainable accuracy and setting of the RADI tag (see [Section 5.2.5](#)).

4.2. Header

As illustrated in [Figure 1](#), 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, and the last $4*N$ bytes in the header are tags.

The offsets array is considered to have an implicitly encoded value of 0 as its zeroth entry. Its members refer to the positions of the tag values in the message values section. All offsets are multiples of four.

The members of the offsets and tags arrays, as well as the message values section are sorted in ascending order by the tag's uint32 value. As a consequence, the offset array is also sorted in ascending order. A tag **MUST NOT** appear more than once in a header.

The first post-header byte, i.e., the first byte of the message values section, is at offset 0. The value associated with the i th tag begins at $\text{offset}[i]$ and ends at $\text{offset}[i+1]-1$, with the exception of the last value, which ends at the end of the message. Values **MAY** have zero length. All lengths and offsets are in bytes.

5. Protocol Details

As described in [Section 3](#), clients initiate time synchronization by sending requests containing a nonce to servers who send signed time responses in return. Roughtime packets can be sent between clients and servers either as UDP datagrams or via TCP streams. Servers **SHOULD** support both the UDP and TCP transport modes.

RoughTime packets are formatted according to [Figure 2](#) and as described here. The first field is a uint64 with the value 0x4d49544847554f52 ("ROUGHTIM" in ASCII). The second field is a uint32 and contains the length of the third field. The third and last field contains a RoughTime message as specified in [Section 4](#).

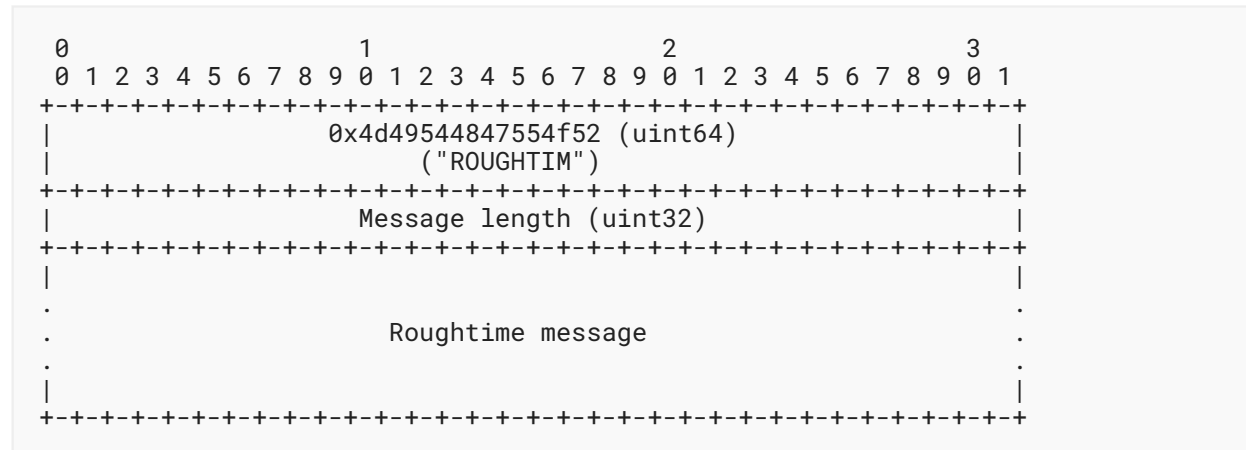


Figure 2: RoughTime Packet

RoughTime request and response packets **MUST** be transmitted in a single datagram when the UDP transport mode is used. Setting the packet's Don't Fragment bit [[RFC0791](#)] is **OPTIONAL** in IPv4 networks. Setting it may cause packets to get dropped, but not setting it could lead to long delays due to reconstruction and dropped fragments.

A RoughTime packet could exceed the maximum deliverable length of a packet on a particular path, making RoughTime queries over UDP impossible on that path. A client **SHOULD** attempt to use the TCP transport mode for RoughTime queries to a server if it does not receive responses to its UDP queries.

Clients **MUST** implement exponential backoff in establishing TCP connections and making requests over UDP. It is **RECOMMENDED** that clients use an initial retry interval of 1 second, a maximum interval of 24 hours, and a base of 1.5. Therefore, the minimum interval, in seconds, before retrying after n failures is $\min(1.5^{(n-1)}, 86400)$. Guidance for implementers considering other values can be found in [Section 3.1.3](#) of [[RFC8085](#)].

Clients **MUST NOT** reset the retry interval until they receive a properly signed response.

Multiple requests and responses can be exchanged over an established TCP connection. Clients **MAY** send multiple outstanding requests and servers **MAY** send responses out of order. The connection **SHOULD** be closed by the client when it has no more requests to send and has received all expected responses. Either side **SHOULD** close the connection in response to synchronization, format, implementation-defined timeouts, or other errors.

All requests and responses contain the VER tag. It contains a list of one or more uint32 version numbers. The version of RoughTime specified by this document has version number 1.

5.1. Requests

A request contains the tags VER, NONC, and TYPE. It **SHOULD** include the tag SRV. Unknown tags **MUST** be ignored by the server. Requests not containing the three mandatory tags **MUST** be ignored. A future version of this protocol may mandate additional tags in the message and assign them semantic meaning.

The size of the request message **SHOULD** be at least 1024 bytes when the UDP transport mode is used. To attain this size, the ZZZZ tag is added to the message. A reason for sending request messages smaller could be to use the UDP transport mode over paths with low maximum deliverable length. However, responding to request messages shorter than 1024 bytes is **OPTIONAL** and servers **MUST NOT** send responses larger than the request messages they are replying to; see [Section 9.7](#).

5.1.1. VER

In a request, the VER tag contains a list of uint32 version numbers. The VER tag **MUST** include at least one RoughTime version supported by the client and **MUST NOT** contain more than 32 version numbers. The version numbers and tags included in the request **MUST** be compatible with each other and the packet contents.

The version numbers **MUST NOT** repeat and **MUST** be sorted in ascending numerical order.

Servers **MUST** ignore any unknown version numbers in the list supplied by the client. If the list contains no version numbers supported by the server, it **MAY** respond with another version or ignore the request entirely; see [Section 5.2.5](#).

5.1.2. NONC

The value of the NONC tag is a 32-byte nonce. It **SHOULD** be generated in a manner indistinguishable from random. RFC 4086 [\[BCP106\]](#) contains specific guidelines regarding this. [Section 8.2](#) describes how to securely generate nonces when querying multiple servers in sequence.

5.1.3. TYPE

The TYPE tag is used to unambiguously distinguish between request and response messages. In a request, it **MUST** contain a uint32 with value 0. Requests containing a TYPE tag with any other value **MUST** be ignored by servers.

5.1.4. SRV

The SRV tag is used by the client to indicate which long-term public key it expects to verify the response with. The value of the SRV tag is $H(0xff \ || \ public_key)$ where `public_key` is the server's long-term, 32-byte Ed25519 public key, and `H` is SHA-512 truncated to the first 32 bytes.

5.1.5. ZZZZ

The ZZZZ tag is used to expand the request to the minimum required length. Its value is all zero bytes.

5.2. Responses

The server begins the request handling process with a set of long-term keys. It resolves which long-term key to use with the following procedure:

1. If the request contains a SRV tag, then the server looks up the long-term key indicated by the SRV value. If no such key exists, then the server **MUST** ignore the request.
2. If the request contains no SRV tag, but the server has just one long-term key, it **SHOULD** select that key. Otherwise, if the server has multiple long-term keys, then it **MUST** ignore the request.

A response contains the tags SIG, NONC, TYPE, PATH, SREP, CERT, and INDX. The structure of a response message is illustrated in [Figure 3](#).

```
|--SIG
|--NONC
|--TYPE
|--PATH
|--SREP
|  |--VER
|  |--RADI
|  |--MIDP
|  |--VERS
|  |--ROOT
|--CERT
|  |--SIG
|  |--DELE
|    |--PUBK
|    |--MINT
|    |--MAXT
|--INDX
```

Figure 3: Roughtime Response Message Structure

No mechanism for reporting errors -- such as wrong request format, unsupported version, or unknown SRV value -- back to the client is provided. This is based on experience from the NTP protocol, where Kiss-o'-Death packets (see [Section 7.4](#) of [\[RFC5905\]](#)) are used to indicate errors. The existence of this unauthenticated protocol feature in NTP makes it possible for on-path attackers to make a client stop using authenticated modes or certain servers altogether (see [Section 5.4](#) of [\[RFC8633\]](#) and [Sections 8.3](#) and [8.7](#) of [\[RFC8915\]](#)). Considering the protocol's dependence on multiple independent servers for security, error reporting functionality has been excluded from this version of Roughtime.

5.2.1. SIG

In general, a SIG tag value is a 64-byte Ed25519 signature [RFC8032] over a concatenation of a context string and the entire value of a tag. All context strings include a terminating zero byte.

The SIG tag in the root of a response is a signature over the SREP value using the public key contained in CERT and the context string "Roughtime v1 response signature".

5.2.2. NONC

The NONC tag contains the nonce of the message being responded to.

5.2.3. TYPE

In a response, the TYPE tag **MUST** contain a uint32 with value 1. Responses containing a TYPE tag with any other value **MUST** be ignored by clients.

5.2.4. PATH

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 5.3. 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.

The PATH **MUST NOT** contain more than 32 hash values. The maximum length of PATH is normally limited by the maximum size of the response message; see Sections 5.1 and 9.7. Server implementations **MUST** select a maximum Merkle tree height (see Section 5.3) that ensures this.

5.2.5. SREP

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

The VER tag, when used in a response, contains a single uint32 version number. It **SHOULD** be one of the version numbers supplied by the client in its request; see Section 5.1.1. The server **MUST** ensure that the version number corresponds with the rest of the packet contents.

The RADI tag value is a uint32 representing the server's estimate of the accuracy of MIDP in seconds. Servers **MUST** ensure that the true time is within ($\text{MIDP} - \text{RADI}$, $\text{MIDP} + \text{RADI}$) at the moment of processing. The value of RADI **MUST NOT** be zero. Since leap seconds cannot be unambiguously represented by Roughtime timestamps, servers **MUST** take this into account when setting the RADI value during leap second events. Servers that do not have any leap second information **SHOULD** set the value of RADI to at least 3. Failure to do so will impact the observed correctness of Roughtime servers and can lead to malfeasance reports.

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

The VERS tag value contains a list of uint32 version numbers supported by the server, sorted in ascending numerical order. It **MUST** contain the version number specified in the VER tag. It **MUST NOT** contain more than 32 version numbers.

The ROOT tag contains a 32-byte value of a Merkle tree root as described in [Section 5.3](#).

5.2.6. CERT

The CERT tag contains a public key certificate signed with the server's private long-term key. Its value is a Roughtime message with the tags SIG and DELE, where SIG is a signature over the DELE value with the context string "Roughtime v1 delegation signature".

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 PUBK, MINT, and MAXT. 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 PUBK tag contains a temporary 32-byte Ed25519 public key that is used to sign the SREP tag.

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.

5.2.7. INDX

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 5.3](#).

5.3. The Merkle Tree

A Merkle tree [[Merkle](#)] 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 one request. 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 full value of the client's request packet, including the "ROUGHTIM" header, 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 leaf node is included in the INDX tag of the response.

The values of all sibling nodes in the path between a request leaf node and the root node are stored in the PATH tag so that the client can reconstruct and validate the value in the ROOT tag using its request packet. These values are each 32 bytes and are stored one after the other with no additional padding or structure. The order in which they are stored is described in the following subsection.

5.3.1. Root Value Validity Check Algorithm

This section describes how to compute the value of the root of the Merkle tree from the values in the tags PATH, INDX, and NONC. The bits of INDX are ordered from least to most significant. $H(x)$ denotes the first 32 bytes of the SHA-512 hash digest of x , and $||$ denotes concatenation.

The algorithm maintains a current value h . At initialization, h is set to $H(0x00 || \text{request_packet})$. For each step of the algorithm, let node be the next 32 bytes in PATH. If the current bit in INDX is 0 then $h = H(0x01 || h || \text{node})$, else $h = H(0x01 || \text{node} || h)$. When no more entries remain in PATH, h is compared to the value of the root of the Merkle tree contained in ROOT. If they are equal, the algorithm succeeds. If they are not, or if any of the remaining bits of INDX is non-zero, the algorithm fails.

5.4. 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 MIDP timestamp lies in the interval specified by the MINT and MAXT timestamps.

The INDX and PATH values prove a hash value derived from the request packet was included in the Merkle tree with value ROOT using the algorithm in [Section 5.3.1](#).

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 that the timestamp's value in the response is correct, but merely that the server guarantees that it signed the timestamp and computed its signature during the time interval ($\text{MIDP}-\text{RADI}$, $\text{MIDP}+\text{RADI}$).

6. Integration into NTP

We assume that there is a bound ϕ on the frequency error in the clock on the machine. Let δ be the time difference between the clock on the client and the clock on the server, and let σ represent the error in the measured value of δ introduced by the measurement process. Given a measurement taken at a local time t , we know the true time is in $(t-\delta-\sigma, t-\delta+\sigma)$. After d seconds have elapsed we know the true time is within $(t-\delta-\sigma-d\phi, t-\delta+\sigma+d\phi)$.

This bound can be used as a simple and effective means to limit the error an attacker can introduce into NTP or Precision Time Protocol (PTP) measurements. For example, an NTP client can ensure that its observation intervals fall entirely within this range or can reject measurements that fall outside.

An application that needs to verify X.509 certificates (which requires knowledge of the current time) but lacks an accurate and trusted time source can use RoughTime to obtain a time estimate. In particular, securely establishing NTS-protected NTP time synchronization requires verification of the NTS-KE server's certificate, which is not possible if the client has no idea of the current time (see [Section 8.5](#) of [\[RFC8915\]](#)). In that case, a RoughTime time estimate can be used for certificate validation.

If an NTP server uses a RoughTime server as a time source for synchronization (and not only for filtering its NTP measurements), the root dispersion **SHOULD** include the server's RADI value, and root delay **SHOULD** include the interval between sending the RoughTime request and receiving the response.

7. Grease

The primary purpose of grease is to prevent protocol ossification, which could prohibit future protocol extensions and development [\[RFC9170\]](#). In RoughTime, grease is also intended to ensure that clients validate signatures. To grease the RoughTime protocol, servers **SHOULD** send back a fraction of responses with any of the following: lack of mandatory tags, version numbers not in the request, undefined tags, or invalid signatures together with incorrect times. Clients **MUST** properly ignore undefined tags and reject invalid responses. Servers **MUST NOT** send back responses with incorrect times and valid signatures. Either signature in the response (i.e., of the SREP or DELE tag) **MAY** be invalid for this application.

8. RoughTime Clients

8.1. Necessary Configuration

To carry out a RoughTime measurement, a client needs a list of servers, a minimum of three of which are operational and not run by the same parties. RoughTime clients **SHOULD** regularly update their view of which servers are trustworthy in order to benefit from the detection of misbehavior (see [Section 8.3](#)). Clients **SHOULD** also have a means of reporting to the provider of such a list, such as an operating system or software vendor, a malfeasance report as described in [Section 8.4](#).

8.2. Measurement Sequence

The client randomly selects at least three servers from the list, and sequentially queries them. To ensure that all possible inconsistencies can be detected, it is necessary for clients to repeat the query sequence twice with the servers in the same order.

The first probe uses a nonce that is randomly generated. The second query uses $H(\text{resp} \parallel \text{rand})$ where rand is a random 32-byte value and resp is the entire response to the first probe, including the "ROUGHTIM" header. Each subsequent query uses $H(\text{resp} \parallel \text{rand})$ for the previous response and a different 32-byte rand value. $H(x)$ and $\parallel$ are defined as in [Section 5.3.1](#).

For each pair of responses (i, j) , where i was received before j , the client **MUST** check that $MIDP_i - RADI_i$ is less than or equal to $MIDP_j + RADI_j$. If these checks pass, the times are consistent with causal ordering. The measurement succeeds if the validity checks described in [Section 5.4](#) are successful, the times reported are consistent with causal ordering, and the delay between request and response is within an implementation-dependent maximum value.

If the validity checks are successful, but at least one of the responses is not consistent with causal ordering, there has been a malfeasance. In case of detected malfeasance, clients **SHOULD**, if it is technically possible, generate a malfeasance report (see [Section 8.4](#)), alert the user, and make another measurement. See [Section 5](#) for guidance on backoff when making repeated measurements.

8.3. Server Lists

To facilitate regular updates of lists of trusted servers, a common server list format is specified here. Support for the common server list format is **OPTIONAL** and clients **MAY** instead implement their own mechanisms for configuring server lists.

A server list is a JSON object [[RFC8259](#)] that contains the key "servers". Server list objects **MAY** also contain the keys "sources" and "reports". Appendix A contains an example server list in the format described here.

Server lists have the "application/roughTime-server+json" media type.

The value of the "servers" key is a list of server objects, each containing the keys "name", "version", "publicKeyType", "publicKey", and "addresses".

The value of "name" is a string that contains a server name suitable for display to a user.

The value of "version" is an integer that indicates the highest RoughTime version number supported by the server.

The value of "publicKeyType" is a string indicating the signature scheme used by the server. The value for servers supporting version 1 of RoughTime is "ed25519".

The value of "publicKey" is a string that is base64-encoded [[RFC4648](#)] and represents the long-term public key of the server in a format consistent with the value of "publicKeyType".

The value of "addresses" is a list of address objects. An address object contains the keys "protocol" and "address". The value of "protocol" is either "tcp" or "udp", indicating the transport mode to use. The value of "address" is a string indicating a host and a port number, separated by a colon character, for example "roughTime.example.com:5319". The host part is either an IPv4 address, an IPv6 address, or a fully qualified domain name (FQDN). IPv4 addresses are specified in dotted decimal notation. IPv6 addresses **MUST** conform to the "Text Representation of Addresses" ([Section 2.2](#) of [[RFC4291](#)]) and **MUST NOT** include zone identifiers [[RFC9844](#)]. To disambiguate IPv6 addresses from ports when zero compression happens, IPv6 addresses are encapsulated within []. The port part is a decimal integer representing a valid port number, i.e., in the range 0-65535.

The value of "sources", if present, is a list of strings indicating where updated versions of the list may be acquired using the HTTP GET method [RFC9110]. Each string is a URL [RFC3986] pointing to a list in the format specified here. The URI scheme **MUST** be HTTPS [RFC9110].

The value of "reports", if present, is a string indicating a URL [RFC3986] where malfeasance reports can be sent by clients using the HTTP POST method. The URI scheme **MUST** be HTTPS [RFC9110].

8.4. Malfeasance Reporting

A malfeasance report is cryptographic proof that a sequence of responses arrived in that order. It can be used to demonstrate that at least one server sent the wrong time.

8.4.1. Malfeasance Report Format

A malfeasance report is a JSON object [RFC8259] that contains the key "responses". Its value is a list of response objects, sorted in the order received. Each response object contains the keys "rand", "publicKey", "request", and "response". The values of all four keys are represented as strings that are base64-encoded [RFC4648]. Appendix B contains an example malfeasance report in the format described here.

Malfeasance reports have the "application/roughTime-malfeasance+json" media type.

The "rand" key **MAY** be omitted from the first response object in the list. In all other cases, its value is the 32-byte value used to generate the request nonce value from the previous response packet.

The value of "publicKey" is the long-term key that the server was expected to use for deriving the response signature.

The value of "request" is the transmitted request packet, including the "ROUGHTIM" header.

The value of "response" is the received response packet, including the "ROUGHTIM" header.

8.4.2. Reporting

When the client's list of servers has an associated URL for malfeasance reports, it **SHOULD** send a malfeasance report to that URL when malfeasance is detected (see Section 8.2) and it is technically feasible to do so. Malfeasance reports are sent using the HTTP POST method [RFC9110].

Since the failure of a popular RoughTime server can cause numerous clients to send malfeasance reports at the same time, clients **MUST** use exponential backoff to prevent overloading the server receiving the reports. It is **RECOMMENDED** that clients use an initial retry interval of 10 seconds, a maximum interval of 24 hours, and a base of 1.5. Therefore, the minimum interval, in seconds, before retrying after n failures is $\min(10 * 1.5^{(n-1)}, 86400)$.

Clients **MUST NOT** send malfeasance reports in response to signature verification failures or any other protocol errors.

As described in [Section 1](#), the operational rules for acceptance or rejection of a particular malfeasance report are beyond the scope of this document.

9. Security Considerations

9.1. Confidentiality

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

9.2. Integrity and Authenticity

The RoughTime protocol only provides integrity and authenticity protection for data contained in the SREP tag. Accordingly, new tags **SHOULD** be added to the SREP tag whenever possible.

9.3. Generating Private Keys

Although any random 256-bit string can be used as a private Ed25519 key, it has a high risk of being vulnerable to small-subgroup attacks and timing side-channel leaks. For this reason, all private keys used in RoughTime **MUST** be generated following the procedure described in [Section 5.1.5](#) of [\[RFC8032\]](#).

9.4. Private Key Compromise

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. To protect against this, it is necessary for clients to query multiple servers in accordance with the procedure described in [Section 8.2](#).

9.5. Quantum Resistance

Since the only supported signature scheme, Ed25519, is not quantum resistant, the RoughTime version described in this document will not survive the advent of quantum computers. A later version will have to be devised and implemented before then. The use of a single version number as the negotiation point rather than defining a suite of acceptable signatures is intended to prevent fragmentation and misconfiguration.

9.6. Maintaining Lists of Servers

The infrastructure and procedures for 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.

9.7. Amplification Attacks

UDP protocols that send responses significantly larger than requests, such as NTP, have previously been leveraged for amplification attacks. To prevent RoughTime from being used for such attacks, servers **MUST NOT** send response packets larger than the request packets sent by clients.

10. 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 in a nonrandom manner can cause leaks of private data or enable tracking of clients as they move between networks.

11. Operational Considerations

It is expected that clients identify a server by its long-term public key. In multi-tenancy environments, where multiple servers may be listening on the same IP or port space, the protocol is designed so that the client indicates which server it expects to respond. This is done with the SRV tag. Additional recommendations for clients are listed in [Section 8](#).

12. IANA Considerations

12.1. Service Name and Transport Protocol Port Number Registry

IANA has allocated the following entry in the "Service Name and Transport Protocol Port Number Registry":

Service Name: roughTime

Transport Protocol: tcp, udp

Assignee: IESG <iesg@ietf.org>

Contact: IETF Chair <chair@ietf.org>

Description: RoughTime time synchronization

Reference: RFC 10049

Port Number: 5319

12.2. RoughTime Versions Registry

IANA has created a new registry titled "RoughTime Versions" in a new "RoughTime" registry group. Entries have the following fields:

Version ID (**REQUIRED**): A 32-bit unsigned integer

Version Name (**REQUIRED**): A short text string naming the version being identified

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

The policy for allocation of new entries is IETF Review [[RFC8126](#)].

The initial contents of this registry are specified below.

Version ID	Version Name	Reference
0x0	Reserved	RFC 10049
0x1	RoughTime version 1	RFC 10049
0x2-0x7fffffff	Unassigned	
0x80000000-0xbfffffff	Reserved for Experimental Use	RFC 10049
0xc0000000-0xffffffff	Reserved for Private Use	RFC 10049

Table 1: Initial Contents of the RoughTime Versions Registry

Private and Experimental Use are defined in [[RFC8126](#)]. The experimental range is intended for testing and evaluating new versions of the RoughTime protocol. Such tests may be conducted over the open Internet.

12.3. RoughTime Tags Registry

IANA has created a new registry titled "RoughTime Tags" in a new "RoughTime" registry group. Entries have the following fields:

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

ASCII Representation (**REQUIRED**): The ASCII representation of the tag in accordance with [Section 4.1.3](#) of this document.

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

The policy for allocation of new entries in this registry is Specification Required [[RFC8126](#)].

The initial contents of this registry are specified below.

Tag	ASCII Representation	Reference
0x00474953	SIG	RFC 10049
0x00524556	VER	RFC 10049
0x00565253	SRV	RFC 10049
0x434e4f4e	NONC	RFC 10049
0x454c4544	DELE	RFC 10049
0x45505954	TYPE	RFC 10049
0x48544150	PATH	RFC 10049
0x49444152	RADI	RFC 10049
0x4b425550	PUBK	RFC 10049
0x5044494d	MIDP	RFC 10049
0x50455253	SREP	RFC 10049
0x53524556	VERS	RFC 10049
0x544e494d	MINT	RFC 10049
0x544f4f52	ROOT	RFC 10049
0x54524543	CERT	RFC 10049
0x5458414d	MAXT	RFC 10049
0x58444e49	INDX	RFC 10049
0x5a5a5a5a	ZZZZ	RFC 10049

Table 2: Initial Contents of the Roughtime Tags Registry

12.4. Media Type Registry

12.4.1. Media Type for Roughtime Server List

IANA has allocated the following entry in the "Media Types" registry.

Type name: application

Subtype name: roughtime-server+json

Required parameters: N/A

Optional parameters: N/A

Encoding considerations: Encoding considerations are identical to those specified for the "application/json" media type, see [\[RFC8259\]](#).

Security considerations: [Section 9](#) of RFC 10049.

Interoperability considerations: N/A

Published specification: [Section 8.3](#) of RFC 10049.

Applications that use this media type: RoughTime clients (RFC 10049) that update their lists of RoughTime servers.

Fragment identifier considerations: N/A

Additional information:

Deprecated alias names for this type: N/A

Magic number(s): N/A

File extension(s): N/A

Macintosh file type code(s): N/A

Person & email address to contact for further information: See Authors' Addresses section of RFC 10049.

Intended usage: COMMON

Restrictions on usage: N/A

Author: See Authors' Addresses section of RFC 10049.

Change controller: Internet Engineering Task Force

12.4.2. Media Type for RoughTime Malfeasance

IANA has allocated the following entry in the "Media Types" registry.

Type name: application

Subtype name: roughTime-malfeasance+json

Required parameters: N/A

Optional parameters: N/A

Encoding considerations: Encoding considerations are identical to those specified for the "application/json" media type, see [RFC8259].

Security considerations: Section 9 of RFC 10049.

Interoperability considerations: N/A

Published specification: Section 8.4.1 of RFC 10049.

Applications that use this media type: RoughTime clients (RFC 10049) use this media type to report cryptographic proof that a RoughTime server has sent the wrong time.

Fragment identifier considerations: N/A

Additional information:

Deprecated alias names for this type: N/A

Magic number(s): N/A

File extension(s): N/A

Macintosh file type code(s): N/A

Person & email address to contact for further information: See Authors' Addresses section of RFC 10049.

Intended usage: COMMON

Restrictions on usage: N/A

Author: See Authors' Addresses section of RFC 10049.

Change controller: Internet Engineering Task Force

13. References

13.1. Normative References

[BCP106] Best Current Practice 106, <<https://www.rfc-editor.org/info/bcp106>>.

At the time of writing, this BCP comprises the following:

Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, DOI 10.17487/RFC4086, June 2005, <<https://www.rfc-editor.org/info/rfc4086>>.

[RFC0020] Cerf, V., "ASCII format for network interchange", STD 80, RFC 20, DOI 10.17487/RFC20, October 1969, <<https://www.rfc-editor.org/info/rfc20>>.

- [RFC0791] Postel, J., "Internet Protocol", STD 5, RFC 791, DOI 10.17487/RFC791, September 1981, <<https://www.rfc-editor.org/info/rfc791>>.
- [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>>.
- [RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <<https://www.rfc-editor.org/info/rfc3986>>.
- [RFC4291] Hinden, R. and S. Deering, "IP Version 6 Addressing Architecture", RFC 4291, DOI 10.17487/RFC4291, February 2006, <<https://www.rfc-editor.org/info/rfc4291>>.
- [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>>.
- [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>>.
- [RFC8085] Eggert, L., Fairhurst, G., and G. Shepherd, "UDP Usage Guidelines", BCP 145, RFC 8085, DOI 10.17487/RFC8085, March 2017, <<https://www.rfc-editor.org/info/rfc8085>>.
- [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>>.
- [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>>.
- [RFC9110] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP Semantics", STD 97, RFC 9110, DOI 10.17487/RFC9110, June 2022, <<https://www.rfc-editor.org/info/rfc9110>>.

13.2. Informative References

- [Merkle] Merkle, R. C., "A Digital Signature Based on a Conventional Encryption Function", Advances in Cryptology - CRYPTO '87, Lecture Notes in Computer Science, vol. 293, pp. 369-378, DOI 10.1007/3-540-48184-2_32, 1988, <https://doi.org/10.1007/3-540-48184-2_32>.
- [RFC0738] Harrenstien, K., "Time server", RFC 738, DOI 10.17487/RFC738, October 1977, <<https://www.rfc-editor.org/info/rfc738>>.

- [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>>.
- [RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <<https://www.rfc-editor.org/info/rfc8126>>.
- [RFC8633] Reilly, D., Stenn, H., and D. Sibold, "Network Time Protocol Best Current Practices", BCP 223, RFC 8633, DOI 10.17487/RFC8633, July 2019, <<https://www.rfc-editor.org/info/rfc8633>>.
- [RFC8915] Franke, D., Sibold, D., Teichel, K., Dansarie, M., and R. Sundblad, "Network Time Security for the Network Time Protocol", RFC 8915, DOI 10.17487/RFC8915, September 2020, <<https://www.rfc-editor.org/info/rfc8915>>.
- [RFC9170] Thomson, M. and T. Pauly, "Long-Term Viability of Protocol Extension Mechanisms", RFC 9170, DOI 10.17487/RFC9170, December 2021, <<https://www.rfc-editor.org/info/rfc9170>>.
- [RFC9523] Rozen-Schiff, N., Dolev, D., Mizrahi, T., and M. Schapira, "A Secure Selection and Filtering Mechanism for the Network Time Protocol with Khronos", RFC 9523, DOI 10.17487/RFC9523, February 2024, <<https://www.rfc-editor.org/info/rfc9523>>.
- [RFC9844] Carpenter, B. and R. Hinden, "Entering IPv6 Zone Identifiers in User Interfaces", RFC 9844, DOI 10.17487/RFC9844, August 2025, <<https://www.rfc-editor.org/info/rfc9844>>.

Appendix A. Example Server List

This appendix presents an example RoughTime server list in the format described by [Section 8.3](#).

```
{
  "servers": [
    {
      "name": "example.com RoughTime server",
      "version": 1,
      "publicKeyType": "ed25519",
      "publicKey": "203mkkheDExCuhG+ZNIoWm0/IdCdLzADgUn8SnC4hME=",
      "addresses": [
        {
          "protocol": "udp",
          "address": "roughTime.example.com:5319"
        },
        {
          "protocol": "tcp",
          "address": "roughTime.example.com:5319"
        }
      ]
    },
    {
      "name": "A UDP-only server specified with IP addresses",
      "version": 1,
      "publicKeyType": "ed25519",
      "publicKey": "ZYfeGa94YuG1IZrV3kR9+8/nmZ2lX2XyHmiSb+wI00Y=",
      "addresses": [
        {
          "protocol": "udp",
          "address": "192.0.2.33:5319"
        },
        {
          "protocol": "udp",
          "address": "[2001:db8::2:33]:5319"
        }
      ]
    }
  ],
  "sources": [
    "https://www.example.net/roughTime/ecosystem.json",
    "https://www.example.org/roughTime/ecosystem.json"
  ],
  "reports": "https://www.example.net/roughTime/malfeasance"
}
```

Appendix B. Example Malfeasance Report

This appendix presents an example RoughTime malfeasance report in the format described by [Section 8.4.1](#). The report provides sufficient information to prove that the server with the public key `1RhHag6fn2wZQ6idy10ChgpRgks3gvdMM2hWNeJNgXg=` responded with a time that is inconsistent with the times reported by the two other servers.

Page 26

[illegible]

Acknowledgments

Aanchal Malhotra and Adam Langley authored early draft versions of this document. Daniel Franke, Sarah Grant, Erik Kline, Martin Langer, Ben Laurie, Peter Löthberg, Michael McCourt, Hal Murray, Tal Mizrahi, Ruben Nijveld, Christopher Patton, Thomas Peterson, Rich Salz, Dieter Sibold, Ragnar Sundblad, Kristof Teichel, Luke Valenta, David Venhoek, Ulrich Windl, and the

other members of the NTP Working Group contributed comments and suggestions as well as pointed out errors. We also acknowledge the helpful comments and suggestions provided by the Last Call reviewers and members of the IESG.

Authors' Addresses

Watson Ladd

Akamai Technologies

Email: watsonbladd@gmail.com

Marcus Dansarie

Netnod

Email: marcus@dansarie.se

URI: <https://orcid.org/0000-0001-9246-0263>