Network Time Protocols
Internet-Draft
Intended status: Experimental
Expires: 18 September 2026

W. Ladd
Akamai Technologies
M. Dansarie
Netnod

17 March 2026

Roughtime
draft-ietf-ntp-roughtime-19

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 give clients a format for reporting 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.

About This Document

This note is to be removed before publishing as an RFC.

Status information for this document may be found at https://datatracker.ietf.org/doc/draft-ietf-ntp-roughtime/.

Source for this draft and an issue tracker can be found at https://github.com/ietf-wg-ntp/draft-roughtime.

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 18 September 2026.

Table of Contents

1. Introduction

Time synchronization is essential to Internet security as many security protocols and other applications require it [RFC738]. 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, result in revoking trust in that particular key.

Unlike Khronos [RFC9523], Roughtime produces external evidence that timeservers 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 and Definitions

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] which includes the hash value of the client's request 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.

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 which 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 sections.

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

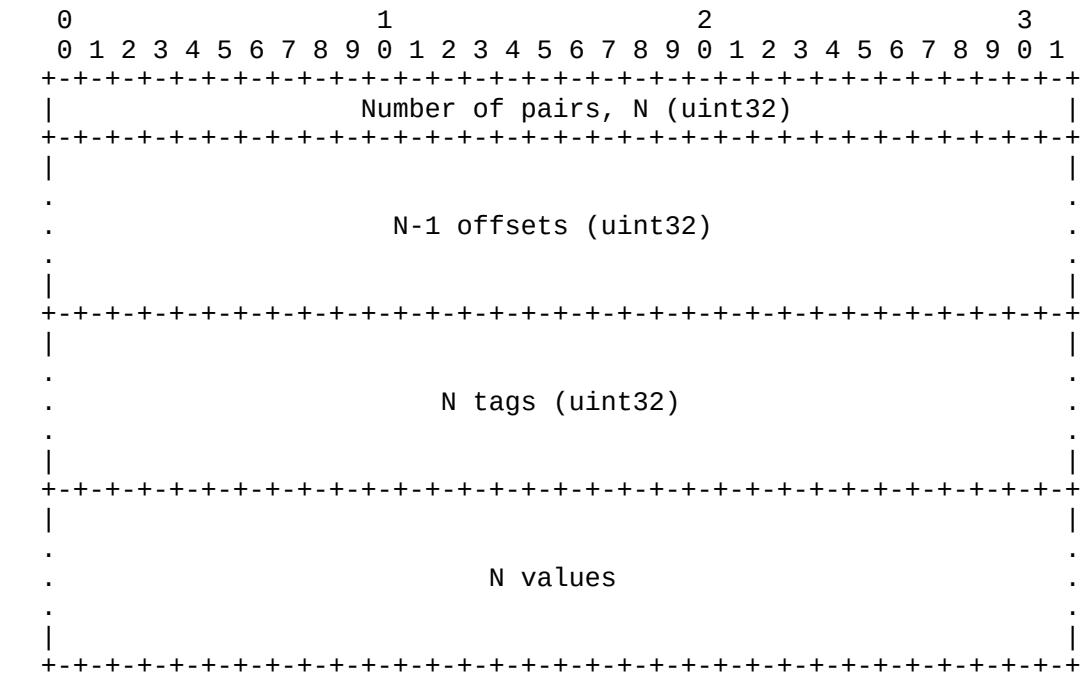

Figure 1: Roughtime Message

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) [RFC20] 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 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 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.

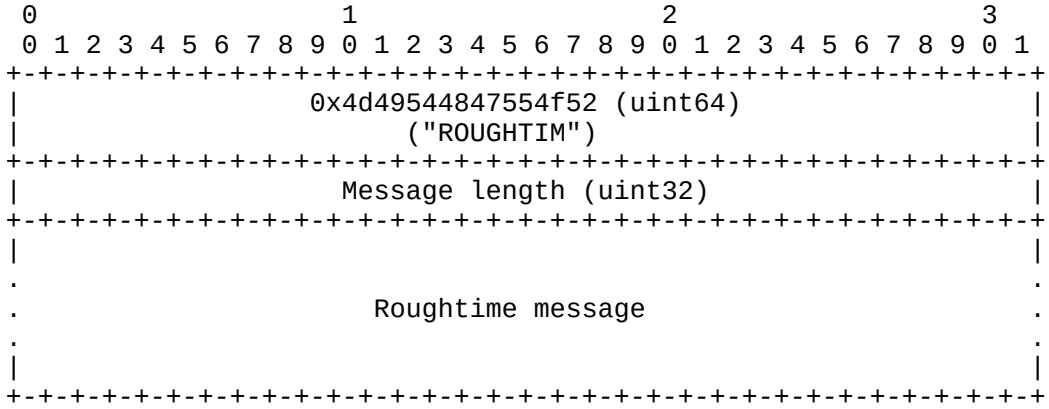

Figure 2: Roughtime packet

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 [RFC791] 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.

NOTE TO RFC EDITOR: remove this paragraph before publication. For testing this draft of the document, a version number of 0x8000000c is used.

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. BCP 106 [RFC4086] 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.

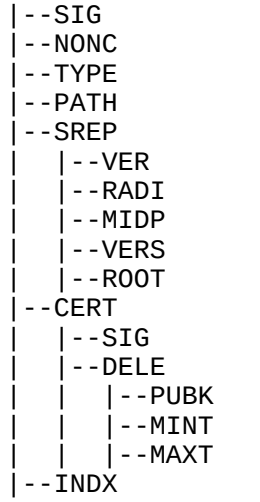

Figure 3: Roughtime response message structure.

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 signature context ASCII 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 Section 5.1 and Section 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 (MIDP-RADI, MIDP+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 which 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 next section.

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 || 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 || node), else h = H(0x01 || 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 (MIDP-RADI, MIDP+RADI).

6. Integration into NTP

We assume that there is a bound phi on the frequency error in the clock on the machine. Let delta be the time difference between the clock on the client and the clock on the server and let sigma represent the error in the measured value of delta 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 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 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(resp || 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(resp || rand) for the previous response and a different 32-byte rand value. H(x) and || 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 [RFC8259] object 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.

NOTE TO RFC EDITOR: remove this paragraph before publication. To indicate compatibility with drafts of this document, a decimal representation of the version number indicated in Section 5 SHOULD be used. For indicating compatibility with pre-IETF specifications of Roughtime, the version number 3000600613 SHOULD be used.

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 base64-encoded [RFC4648] string representing 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:2002". 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" [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 [RFC8259] object 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 base64-encoded [RFC4648] strings. 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 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 is requested to allocate 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 (mailto:iesg@ietf.org)

Contact: IETF Chair chair@ietf.org (mailto:chair@ietf.org)

Description: Roughtime time synchronization

Reference: [[this document]]

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

12.2. Roughtime Versions Registry

IANA is requested to create a new registry titled "Roughtime Versions" in a new "Roughtime" registry group. Entries will 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 in Table 1.

Version ID	Version name	Reference
0x0	Reserved	[[this document]]
0x1	Roughtime version 1	[[this document]]
0x2-0x7fffffff	Unassigned	
0x80000000-0xbfffffff	Reserved for experimental use	[[this document]]
0xc0000000-0xffffffff	Reserved for private use	[[this document]]

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 is requested to create a new registry titled "Roughtime Tags" in a new "Roughtime" registry group. Entries will 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 in Table 2.

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

Table 2: Initial contents of the Roughtime Tags registry.

12.4. Media Type Registry

12.4.1. Roughtime Server List MIME type

IANA is requested to allocate the following entry in the Media Type 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 [[this document]].

Interoperability considerations: N/A

Published specification: Section 8.3 of [[this document]].

Applications that use this media type: Roughtime clients [[this document]] 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 [[this document]].

Intended usage: COMMON

Restrictions on usage: N/A

Author: See Authors' Addresses section of [[this document]].

Change controller: Internet Engineering Task Force

12.4.2. Roughtime Malfeasance MIME type

IANA is requested to allocate the following entry in the Media Type 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 [[this document]].

Interoperability considerations: N/A

Published specification: Section 8.4.1 of [[this document]].

Applications that use this media type: Roughtime clients [[this document]] 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 [[this document]].

Intended usage: COMMON

Restrictions on usage: N/A

Author: See Authors' Addresses section of [[this document]].

Change controller: Internet Engineering Task Force

13. References

13.1. Normative References

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

[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/rfc/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/rfc/rfc3986>.

[RFC4086] 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/rfc/rfc4086>.

[RFC4291] Hinden, R. and S. Deering, "IP Version 6 Addressing Architecture", RFC 4291, DOI 10.17487/RFC4291, February 2006, <https://www.rfc-editor.org/rfc/rfc4291>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <https://www.rfc-editor.org/rfc/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/rfc/rfc6234>.

[RFC791] Postel, J., "Internet Protocol", STD 5, RFC 791, DOI 10.17487/RFC0791, September 1981, <https://www.rfc-editor.org/rfc/rfc791>.

[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/rfc/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/rfc/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/rfc/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/rfc/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/rfc/rfc9110>.

13.2. Informative References

[Merkle] Merkle, R. C., "A Digital Signature Based on a Conventional Encryption Function", in Pomerance, C. (eds) Advances in Cryptology, Lecture Notes in Computer Science vol 293, DOI 10.1007/3-540-48184-2_32, 1988, <https://doi.org/10.1007/3-540-48184-2_32>.

[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/rfc/rfc5905>.

[RFC738] Harrenstien, K., "Time server", RFC 738, DOI 10.17487/RFC0738, October 1977, <https://www.rfc-editor.org/rfc/rfc738>.

[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/rfc/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/rfc/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/rfc/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/rfc/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/rfc/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/rfc/rfc9844>.

Acknowledgments

Aanchal Malhotra and Adam Langley authored early drafts of this document. Daniel Franke, Sarah Grant, Erik Kline, Martin Langer, Ben Laurie, Peter Lthberg, 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.

Appendix A. Example Server List

This appendix presents an example Roughtime server list in the format described by Section 8.3.

NOTE TO RFC EDITOR: replace all occurrences of the port number 2002 below with the port number assigned for Roughtime by IANA.

```
{
  "servers": [
    {
      "name": "example.com Roughtime server",
      "version": 1,
      "publicKeyType": "ed25519",
      "publicKey": "2O3mkkheDExCuhG+ZNIoWmO/IdCdLzADgUn8SnC4hME=",
      "addresses": [
        {
          "protocol": "udp",
          "address": "roughtime.example.com:2002"
        },
        {
          "protocol": "tcp",
          "address": "roughtime.example.com:2002"
        }
      ]
    },
    {
      "name": "A UDP-only server specified with IP addresses",
      "version": 1,
      "publicKeyType": "ed25519",
      "publicKey": "ZYfeGa94YuG1IZrV3kR9+8/nmZ2lX2XyHmiSb+wI0OY=",
      "addresses": [
        {
          "protocol": "udp",
          "address": "192.0.2.33:2002"
        },
        {
          "protocol": "udp",
          "address": "[2001:db8::2:33]:2002"
        }
      ]
    }
  ],
  "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 lRhHag6fn2wZQ6idy10ChgpRgks3gvdMM2hWNeJNgXg= responded with a time that is inconsistent with the times reported by the two other servers.


```
{
  "responses": [
    {
      "publicKey": "FnDyLV/68ephhLdFJbdEGCdkVvpXDaVe5PYvRDdlOOY=",
      "request": "Uk9VR0hUSU0ABAAABQAAAAQAAAAkAAAARAAAAEgAAABWRVIAU1JW
AE5PTkNUWVBFWlpaWgEAAACf4gKLPdPfiNTv93lrhNqYgyehDgMyHFmA1BrAhM1QEDBh9l
BlN6LUye6zghiqSWMwyNm0IucxQxW3zTMrwj4dAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
==",
      "response": "Uk9VR0hUSU2UAQAABwAAAEAAAABgAAAAZAAAAGQAAADAAAAAWAE
AAFNJRwBOT05DVFlQRVBBVEhTUkVQQ0VSVElORFhBWL64CToGs4v/4UtfN/80HLFiA09vG
IDRP/zTjcTj8/1DlZWCsVja6RlfwaYnc1wfJqThfhcuSDonrTGyKngBMGH2UGU3otTJ7rO
CGKpJYzDI2bQi5zFDFbfNMyvCPh0BAAAABQAAAAQAAAAIAAAAEAAAABQAAABWRVIAUkFES
U1JRFBWRVJTUk9PVAEAAAADAAAAQ0u4aQAAAAABAAAAc86AWYB/O3Kxzsx4d5P5cbSOftJ
UA8bWVtVrQ3tc+b0CAAAAQAAAAFNJRwBERUxFI2B5tbj5ePjVKYE0PAL1NmgZOAsqh/E2f
rom9Ol5BAnVcLje0C6exbXY8hE3dRvYV01Alru8Ocle+jOZT5r8AwMAAAAgAAAAKAAAAFB
VQktNSU5UTUFYVKqljhhqi4A54vW20e+slwViPyxybNnqKXzimIiIUHQMaBCvaQAAAADYy
d9pAAAAAAAAAAA="
    },
    {
      "publicKey": "l9cdSuR8dFxtG9aJo9pWzUXaX8pftNG4UDC45Qk3znc=",
      "rand": "v/DirVBRQLGtictYD7mN3px02UlMT4J3haTRomt1NNM=",
      "request": "Uk9VR0hUSU0ABAAABQAAAAQAAAAkAAAARAAAAEgAAABWRVIAU1JW
```

```
AE5PTkNUWVBFWlpaWgEAAABFCvadUKPN/U9OXqFalVFnv/EhsAuPPpBYfo5MDULZKv335k
bNTZcsYLM96o77yLbvvRk1fm2TQB4yKxbP1jzeAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
==",
      "response": "Uk9VR0hUSU2UAQAABwAAAEAAAABgAAAAZAAAAGQAAADAAAAAWAE
AAFNJRwBOT05DVFlQRVBBVEhTUkVQQ0VSVElORFj+2I02cCBAfDYzP+8znW6bICqVrAF23
xNLfM+Qycmkpfp1+BQSb4l/6mRll66l2VIfPSQzigl2V5OJgQzGEBcG/ffmRs1Nlyxgsz3
qjvvItu+9GTV+bZNAHjIrFs/WPN4BAAAABQAAAAQAAAAIAAAAEAAAABQAAABWRVIAUkFES
U1JRFBWRVJTUk9PVAEAAAADAAAAw/m2aQAAAAABAAAAS8ROJoXIPSlO9yN+uREb+/UiOJJ
qCjx3VWZ/vD2FqBMCAAAAQAAAAFNJRwBERUxFw0MFQMiDoHySot8rlnV83Vqaa3qTrAY15
W1TzqNuAurOvreNw08BfUwxF7BQ/b/JCwCQcqtD5uRYsvikvuyLCwMAAAAgAAAAKAAAAFB
VQktNSU5UTUFYVCQs/eCxtWjVrXyse0SeojyZdNFkOwe3B3nLHtHJZK/9gRCvaQAAAADxy
d9pAAAAAAAAAAA="
    },
    {
      "publicKey": "lRhHag6fn2wZQ6idy10ChgpRgks3gvdMM2hWNeJNgXg=",
      "rand": "lMvMVoLsakxc5ZmMzEFQ8hh1FaDo2gCXXIX/L4QPSxQ=",
      "request": "Uk9VR0hUSU0ABAAABQAAAAQAAAAkAAAARAAAAEgAAABWRVIAU1JW
AE5PTkNUWVBFWlpaWgEAAADmAab6GlTSr3xRmwwwvRjW6EKRhtyWYYWZHZIGnF8w2b4DuI
ICvoCWTwCaupBfcOEwV+RqBWB6L1JUGn+ot8+9AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

```
  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
  ==",
        "response": "Uk9VR0hUSU2UAQAABwAAAEAAAABgAAAAZAAAAGQAAADAAAAAWAE
  AAFNJRwBOT05DVFlQRVBBVEhTUkVQQ0VSVElORFg8MlhdCJeQ2qzN6b9q646W9kB+XmAdS
  g7ToU1gj3AD8AP8eCHfduMBE0E8j4/LqWIr72zWQx8Y1U/1uxs97yMAvgO4ggK+gJZPAJq
  6kF9w4TBX5GoFYHovUlQaf6i3z70BAAAABQAAAAQAAAAIAAAAEAAAABQAAABWRVIAUkFES
  U1JRFBWRVJTUk9PVAEAAAADAAAAw/m2aQAAAAABAAAA6ho4+0Cml+VJbqU7hsF717uusV2
  HYvRbU9CdjJE/Zn0CAAAAQAAAAFNJRwBERUxFM9Fvq8T9kOrxcS7jviCPHe44HX/75Je1h
  afPJ0f8NoASy29EgD7C0c/LMxXiXxEwyuxYTPN9oseAr9XIt68uDwMAAAAgAAAAKAAAAFB
  VQktNSU5UTUFYVMxBDiNG247IO4onsGcjFHsA3vP+arl+s0lBLhXw0c1RlBCvaQAAAAAEy
  t9pAAAAAAAAAAA="
      }
    ]
  }
```

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