Stream: Independent Submission
RFC: 0000
Category: Informational
Published: October 2023
ISSN: 2070-1721
Authors: M. Schanzenbach, *Fraunhofer AISEC*; C. Grothoff, *Berner Fachhochschule*; B. Fix, *GNUnet e.V.*

RFC 0000
The GNU Name System

Abstract

This document provides the GNU Name System (GNS) technical specification. GNS is a decentralized and censorship-resistant domain name resolution protocol that provides a privacy-enhancing alternative to the Domain Name System (DNS) protocols.

This document defines the normative wire format of resource records, resolution processes, cryptographic routines, and security and privacy considerations for use by implementers.

This specification was developed outside the IETF and does not have IETF consensus. It is published here to inform readers about the function of GNS, guide future GNS implementations, and ensure interoperability among implementations including with the pre-existing GNUnet implementation.

Status of This Memo

This document is not an Internet Standards Track specification; it is published for informational purposes.

This is a contribution to the RFC Series, independently of any other RFC stream. The RFC Editor has chosen to publish this document at its discretion and makes no statement about its value for implementation or deployment. Documents approved for publication by the RFC Editor are not 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/rfc0000.

Copyright Notice

Table of Contents

1. Introduction

This specification describes the GNU Name System (GNS), a censorship-resistant, privacy-preserving, and decentralized domain name resolution protocol. GNS cryptographically secures the binding of names to arbitrary tokens, enabling it to double in some respects as an alternative to some of today's public key infrastructures.

Per Domain Name System (DNS) terminology [RFC1035], GNS roughly follows the idea of a local root zone deployment (see [RFC8806]), with the difference that the design encourages alternative roots and does not expect all deployments to use the same or any specific root zone. In the GNS reference implementation, users can autonomously and freely delegate control of names to zones through their local configurations. GNS expects each user to be in control of their setup. By following the guidelines in Section 9.10, users should manage to avoid any confusion as to how names are resolved.

Name resolution and zone dissemination are based on the principle of a petname system where users can assign local names to zones. The GNS has its roots in ideas from the Simple Distributed Security Infrastructure [SDSI], enabling the decentralized mapping of secure identifiers to memorable names. One of the first academic descriptions of the cryptographic ideas behind GNS can be found in [GNS].

This document defines the normative wire format of resource records, resolution processes, cryptographic routines, and security and privacy considerations for use by implementers.

This specification was developed outside the IETF and does not have IETF consensus. It is published here to guide implementers of GNS and to ensure interoperability among implementations.

1.1. Requirements Notation

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

2. Terminology

Apex Label: This type of label is used to publish resource records in a zone that can be resolved without providing a specific label. It is the GNS method for providing what is called the "zone apex" in DNS [RFC4033]. The apex label is represented using the character U+0040 ("@" without the quotes).

Application: An application is a component that uses a GNS implementation to resolve names into records and processes its contents.

Blinded Zone Key: The blinded zone key is a key derived from a zone key and a label. The zone key and any blinded zone key derived from it are unlinkable without knowledge of the specific label used for the derivation.

Extension Label: This type of label is used to refer to the authoritative zone that the record is in. The primary use for the extension label is in redirections where the redirection target is defined relative to the authoritative zone of the redirection record (see Section 5.2). The extension label is represented using the character U+002B ("+" without the quotes).

Label Separator: Labels in a name are separated using the label separator U+002E ("." without the quotes). In GNS, except for zone Top-Level Domains (zTLDs) (see below) and boxed records (see Section 5.3.3), every label separator in a name indicates delegation to another zone.

Label: A GNS label is a label as defined in [RFC8499]. Labels are UTF-8 strings in Unicode Normalization Form C (NFC) [Unicode-UAX15]. The apex label and the extension label have special purposes in the resolution protocol that are defined in the rest of this document. Zone administrators **MAY** disallow certain labels that might be easily confused with other labels through registration policies (see also Section 9.4).

Name: A name in GNS is a domain name as defined in [RFC8499]: names are UTF-8 strings [RFC3629] consisting of an ordered list of labels concatenated with a label separator. Names are resolved starting from the rightmost label. GNS does not impose length restrictions on names or labels. However, applications **MAY** ensure that name and label lengths are compatible with DNS and, in particular, Internationalized Domain Names for Applications (IDNA) [RFC5890]. In the spirit of [RFC5895], applications **MAY** preprocess names and labels to ensure compatibility with DNS or support specific user expectations -- for example, according to [Unicode-UTS46]. A GNS name may be indistinguishable from a DNS name, and care must be taken by applications and implementers when handling GNS names (see Section 9.10). In order to avoid misinterpretation of example domains with (reserved) DNS domains, this document uses the suffix ".gns.alt" in examples which is also registered in the GANA ".alt Subdomains" registry [GANA] (see also [RFC9476]).

Resolver: In this document, a resolver is the component of a GNS implementation that provides the recursive name resolution logic defined in Section 7.

Resource Record: A GNS resource record is the information associated with a label in a GNS zone. A GNS resource record contains information as defined by its resource record type.

Start Zone: In order to resolve any given GNS name, an initial start zone must be determined for this name. The start zone can be explicitly defined as part of the name using a zTLD. Otherwise, it is determined through a local suffix-to-zone mapping (see Section 7.1).

Top-Level Domain (TLD): The rightmost part of a GNS name is a GNS TLD. A GNS TLD can consist of one or more labels. Unlike DNS TLDs (defined in [RFC8499]), GNS does not expect all users to use the same global root zone. Instead, with the exception of zTLDs (see Section 4.1), GNS TLDs are typically part of the configuration of the local resolver (see Section 7.1) and thus might not be globally unique.

Zone: A GNS zone contains authoritative information (resource records). A zone is uniquely identified by its zone key. Unlike DNS zones, a GNS zone does not need to have an SOA record under the apex label.

Zone Key: The zone key is a key that uniquely identifies a zone. It is usually a public key of an asymmetric key pair. However, the established technical term "public key" is misleading, as in GNS a zone key may be a shared secret that should not be disclosed to unauthorized parties.

Zone Key Derivation Function: The zone key derivation function (ZKDF) blinds a zone key using a label.

Zone Master: The zone master is the component of a GNS implementation that provides local zone management and publication as defined in Section 6.

Zone Owner: The zone owner is the holder of the secret (typically a private key), which (together with a label and a value to sign) allows the creation of zone signatures that can be validated against the respective blinded zone key.

Zone Top-Level Domain (zTLD): A GNS zTLD is a sequence of GNS labels at the end of a GNS name. The zTLD encodes a zone type and zone key of a zone (see Section 4.1). Due to the statistical uniqueness of zone keys, zTLDs are also globally unique. A zTLD label sequence can only be distinguished from ordinary TLD label sequences by attempting to decode the labels into a zone type and zone key.

Zone Type: The type of a GNS zone determines the cipher system and binary encoding format of the zone key, blinded zone keys, and cryptographic signatures.

3. Overview

GNS exhibits the three properties that are commonly used to describe a petname system:

Global names through the concept of zTLDs:
As zones can be uniquely identified by their zone keys and are statistically unique, zTLDs are globally unique mappings to zones. Consequently, GNS domain names with a zTLD suffix are also globally unique. Names with zTLD suffixes are not human readable.

Memorable petnames for zones:
Users can configure local, human-readable references to zones. Such petnames serve as zTLD monikers that provide convenient names for zones to the local operator. The petnames may also be published as suggestions for other users searching for a good label to use when referencing the respective zone.

A secure mapping from names to records:
GNS allows zone owners to map labels to resource records or to delegate authority of names in the subdomain induced by a label to other zones. Zone owners may choose to publish this information to make it available to other users. Mappings are encrypted and signed using keys derived from the respective label before being published in remote storage. When names are resolved, signatures on resource records including delegations are verified by the recursive resolver.

In the remainder of this document, the "implementer" refers to the developer building a GNS implementation including the resolver, zone master, and supporting configuration such as start zones (see Section 7.1).

3.1. Names and Zones

It follows from the above that GNS does not support names that are simultaneously global, secure, and human readable. Instead, names are either global and not human readable or not globally unique and human readable. An example for a global name pointing to the record "example" in a zone is as follows:

```
example.000G006K2TJNMD9VTCYRX7BRVV3HAEPS15E6NHDXKPJA1KAJJEG9AFF884
```

Now consider the case where a user locally configured the petname "pet.gns.alt" for the zone with the "example" record of the name above. The name "example.pet.gns.alt" would then point to the same record as the globally unique name above, but name resolution would only work on the local system where the "pet.gns.alt" petname is configured.

The delegation of petnames and subsequent resolution of delegation build on ideas from the Simple Distributed Security Infrastructure [SDSI]. In GNS, any user can create and manage any number of zones (see Section 4) if their system provides a zone master implementation. For each zone, the zone type determines the respective set of cryptographic operations and the wire formats for encrypted data, public keys, and signatures. A zone can be populated with mappings from labels to resource records (see Section 5) by its owner. A label can be mapped to a delegation record; this results in the corresponding subdomain being delegated to another zone. Circular delegations are explicitly allowed, including delegating a subdomain to its immediate parent zone. In order to support (legacy) applications as well as to facilitate the use of petnames, GNS defines auxiliary record types in addition to supporting existing DNS records.

3.2. Publishing Binding Information

Zone contents are encrypted and signed before being published in remote key-value storage (see Section 6), as illustrated in Figure 1. In this process, unique zone identification is hidden from the network through the use of key blinding. Key blinding allows the creation of signatures for zone contents using a blinded public/private key pair. This blinding is realized using a deterministic key derivation from the original zone key and corresponding private key using record label values as inputs from which blinding factors are derived. Specifically, the zone owner can derive blinded private keys for each record set published under a label, and a resolver can derive the corresponding blinded public keys. It is expected that GNS implementations use decentralized remote storage entities, such as distributed hash tables (DHTs), in order to facilitate availability within a network without the need for dedicated infrastructure. The specification of such a distributed or decentralized storage entity is out of scope for this document, but possible existing implementations include those based on [RFC7363], [Kademlia], or [R5N].

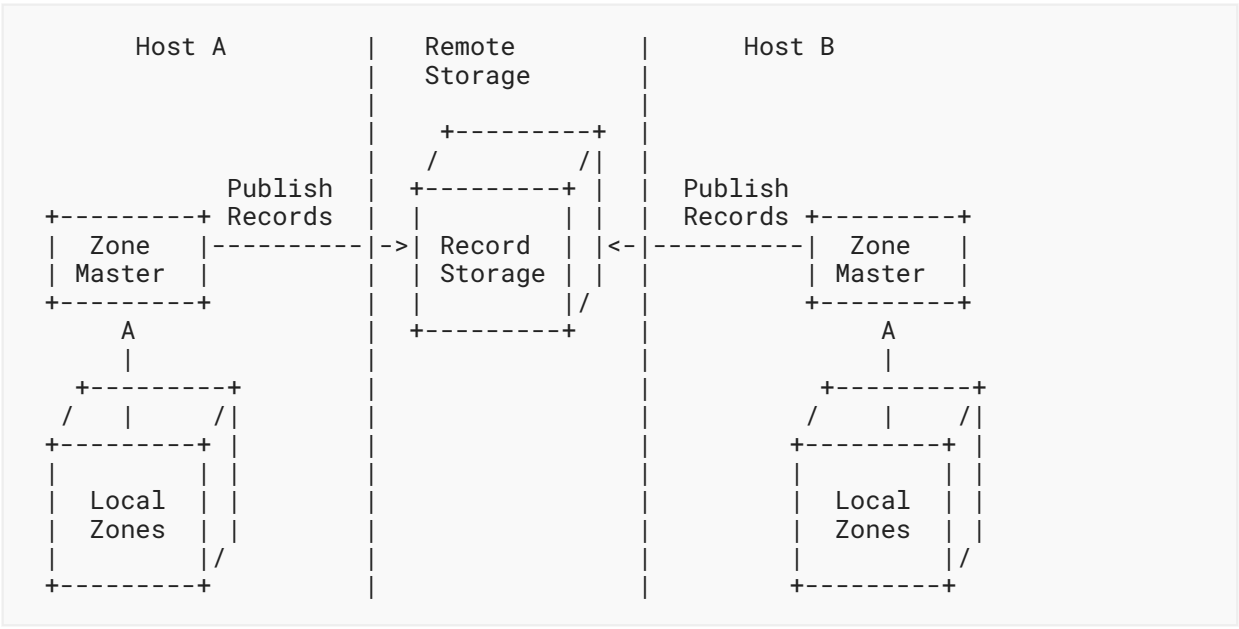

Figure 1: An Example Diagram of Two Hosts Publishing GNS Zones

A zone master implementation **SHOULD** be provided as part of a GNS implementation to enable users to create and manage zones. If this functionality is not implemented, names can still be resolved if zone keys for the initial step in the name resolution have been configured (see Section 7) or if the names end with a zTLD suffix.

3.3. Resolving Names

Applications use the resolver to look up GNS names. Starting from a configurable start zone, names are resolved by following zone delegations recursively, as illustrated in Figure 2. For each label in a name, the recursive GNS resolver fetches the respective record set from the storage layer (see Section 7). Without knowledge of the label values and the zone keys, the different derived keys are unlinkable to both the original zone key and each other. This prevents zone enumeration (except via expensive online brute-force attacks): to confirm the affiliation of a query or the corresponding encrypted record set with a specific zone requires knowledge of both the zone key and the label, neither of which is disclosed to remote storage by the protocol. At the same time, the blinded zone key and digital signatures associated with each encrypted record set allow resolvers and oblivious remote storage to verify the integrity of the published information without disclosing anything about the originating zone or the record sets.

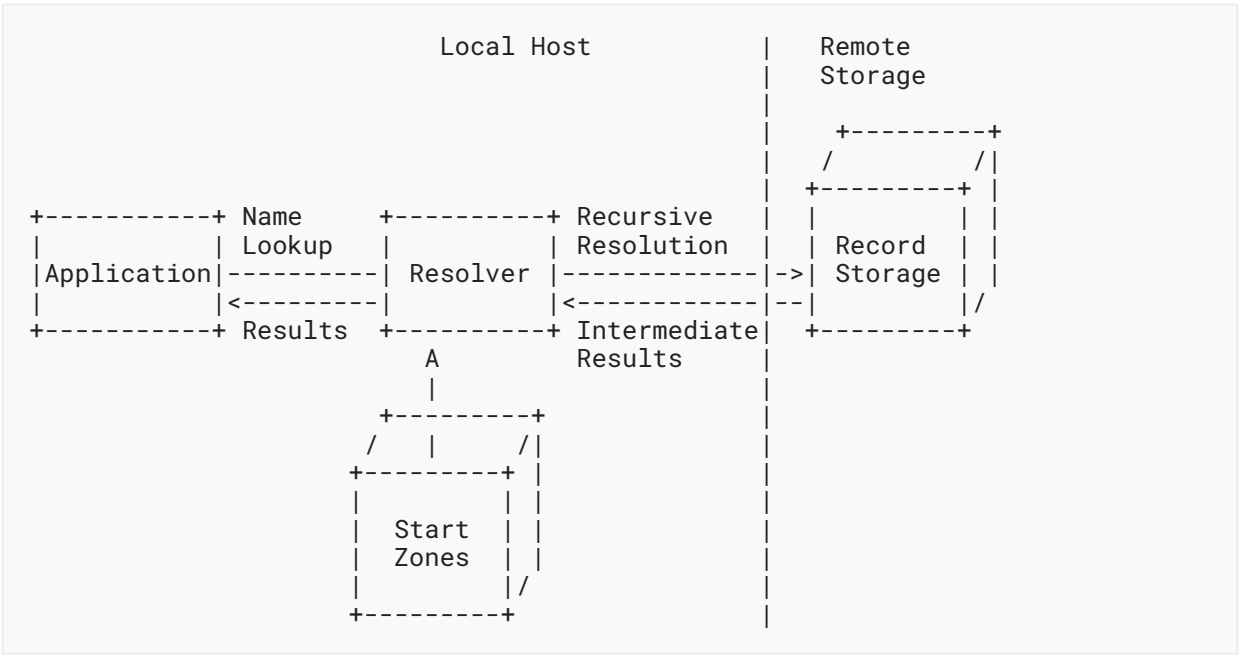

Figure 2: High-Level View of the GNS Resolution Process

4. Zones

A zone in GNS is uniquely identified by its zone type (ztype) and zone key. Each zone can be referenced by its zTLD (see Section 4.1), which is a string that encodes the zone type and zone key. The ztype is a unique 32-bit number that corresponds to a resource record type number identifying a delegation record type in the GANA "GNS Record Types" registry [GANA]. The ztype is a unique identifier for the set cryptographic functions of the zone and the format of the delegation record type. Any ztype registration **MUST** define the following set of cryptographic functions:

KeyGen() -> d, zk
: A function for generating a new private key d and the corresponding public zone key zk.

ZKDF(zk,label) -> zk'
: A ZKDF that blinds a zone key zk using a label. zk and zk' must be unlinkable. Furthermore, blinding zk with different values for the label must result in different, unlinkable zk' values.

S-Encrypt(zk,label,expiration,plaintext) -> ciphertext
: A symmetric encryption function that encrypts the plaintext to derive ciphertext based on key material derived from the zone key zk, a label, and an expiration timestamp. In order to leverage performance-enhancing caching features of certain underlying storage entities -- in particular, DHTs -- a deterministic encryption scheme is recommended.

S-Decrypt(zk,label,expiration,ciphertext) -> plaintext
: A symmetric decryption function that decrypts the ciphertext into plaintext based on key material derived from the zone key, a label, and an expiration timestamp.

Sign(d,message) -> signature
: A function for signing a message using the private key d, yielding an unforgeable cryptographic signature. In order to leverage performance-enhancing caching features of certain underlying storage entities -- in particular, DHTs -- a deterministic signature scheme is recommended.

Verify(zk,message,signature) -> boolean
: A function for verifying that the signature was created using the private key d corresponding to the zone key zk where d,zk := Keygen(). The function returns a boolean value of "TRUE" if the signature is valid and "FALSE" otherwise.

SignDerived(d,label,message) -> signature
: A function for signing a message (typically encrypted record data) that can be verified using the derived zone key zk' := ZKDF(zk,label). In order to leverage performance-enhancing caching features of certain underlying storage entities -- in particular, DHTs -- a deterministic signature scheme is recommended.

VerifyDerived(zk,label,message,signature) -> boolean
: A function for verifying the signature using the derived zone key zk' := ZKDF(zk,label). The function returns a boolean value of "TRUE" if the signature is valid and "FALSE" otherwise.

The cryptographic functions of the default ztypes are specified with their corresponding delegation records as discussed in Section 5.1. In order to support cryptographic agility, additional ztypes **MAY** be defined in the future that replace or update the default ztypes defined in this document. All ztypes **MUST** be registered as dedicated zone delegation record types in the GANA "GNS Record Types" registry (see [GANA]). When defining new record types, the cryptographic security considerations of this document -- in particular, Section 9.3 -- apply.

4.1. Zone Top-Level Domain (zTLD)

A zTLD is a string that encodes the zone type and zone key into a domain name suffix. A zTLD is used as a globally unique reference to a zone in the process of name resolution. It is created by encoding a binary concatenation of the zone type and zone key (see Figure 3). The used encoding is a variation of the Crockford Base32 encoding [CrockfordB32] called Base32GNS. The encoding and decoding symbols for Base32GNS including this modification are defined in Table 4 (Appendix C). The functions for encoding and decoding based on Table 4 are called Base32GNS-Encode and Base32GNS-Decode, respectively.

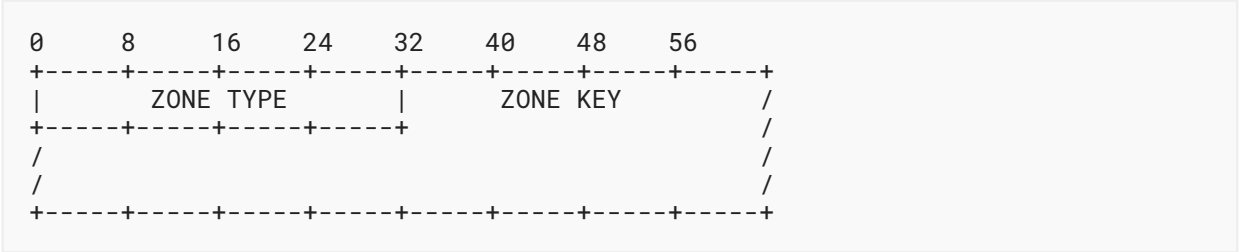

Figure 3: The Binary Representation of the zTLD

The ZONE TYPE must be encoded in network byte order. The format of the ZONE KEY depends entirely on the ZONE TYPE.

Consequently, a zTLD is encoded and decoded as follows:

```
zTLD := Base32GNS-Encode(ztype||zkey)
ztype||zkey := Base32GNS-Decode(zTLD)
```

where "||" is the concatenation operator.

The zTLD can be used "as is" as a rightmost label in a GNS name. If an application wants to ensure DNS compatibility of the name, it **MAY** also represent the zTLD as follows: if the zTLD is less than or equal to 63 characters, it can be used as a zTLD as is. If the zTLD is longer than 63 characters, the zTLD is divided into smaller labels separated by the label separator. Here, the most significant bytes of the "ztype||zkey" concatenation must be contained in the rightmost label of the resulting string and the least significant bytes in the leftmost label of the resulting string. This allows the resolver to determine the ztype and zTLD length from the rightmost label and to subsequently determine how many labels the zTLD should span. A GNS implementation **MUST** support the division of zTLDs in DNS-compatible label lengths. For example, assuming a zTLD of 130 characters, the division is as follows:

```
zTLD[126..129].zTLD[63..125].zTLD[0..62]
```

4.2. Zone Revocation

In order to revoke a zone key, a signed revocation message **MUST** be published. This message **MUST** be signed using the private key of the zone. The revocation message is broadcast to the network. The specification of the broadcast mechanism is out of scope for this document. A possible broadcast mechanism for efficient flooding in a distributed network is implemented in [GNUnet]. Alternatively, revocation messages could also be distributed via a distributed ledger or a trusted central server. To prevent flooding attacks, the revocation message **MUST** contain a proof of work (PoW). The revocation message, including the PoW, **MAY** be calculated ahead of time to support timely revocation.

For all occurrences below, "Argon2id" is the password-based key derivation function as defined in [RFC9106]. For the PoW calculations, the algorithm is instantiated with the following parameters:

S: The salt. Fixed 16-byte string: "GnsRevocationPow"

t: Number of iterations: 3

m: Memory size in KiB: 1024

T: Output length of hash in bytes: 64

p: Parallelization parameter: 1

v: Algorithm version: 0x13

y: Algorithm type (Argon2id): 2

X: Unused

K: Unused

Figure 4 illustrates the format of the data "P" on which the PoW is calculated.

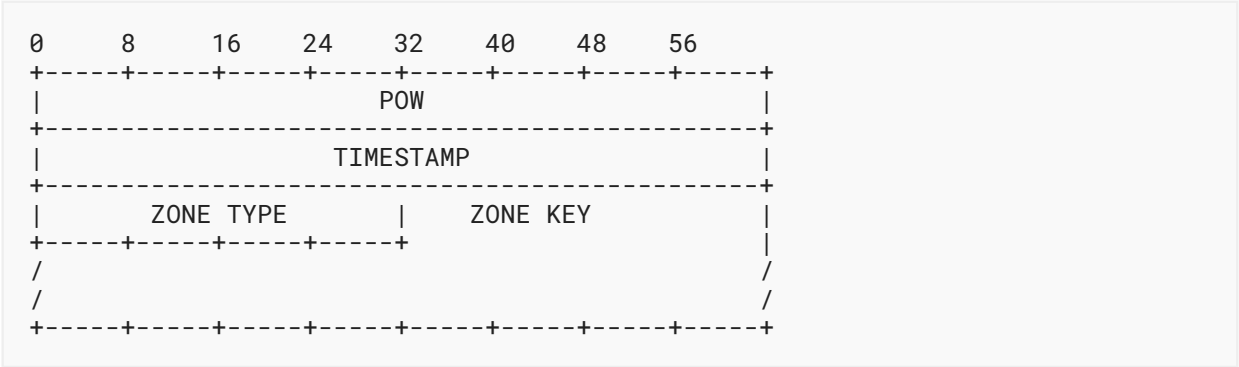

Figure 4: The Format of the PoW Data

POW: A 64-bit value that is a solution to the PoW. In network byte order.

TIMESTAMP: Denotes the absolute 64-bit date when the revocation was computed. In microseconds since midnight (0 hour), January 1, 1970 UTC in network byte order.

ZONE TYPE: The 32-bit zone type in network byte order.

ZONE KEY: The 256-bit public key zk of the zone that is being revoked. The wire format of this value is defined by the ZONE TYPE.

Usually, PoW schemes require that one POW value be found, such that a specific number of leading zeroes are found in the hash result. This number is then referred to as the difficulty of the PoW. In order to reduce the variance in time it takes to calculate the PoW, a valid GNS revocation requires that a number of different PoWs (Z, as defined below) must be found that on average have D leading zeroes.

Given an average difficulty of D, the proofs have an expiration time of EPOCH. Applications **MAY** calculate proofs with a difficulty that is higher than D by providing POW values where there are (on average) more than D bits of leading zeroes. With each additional bit of difficulty, the lifetime of the proof is prolonged by another EPOCH. Consequently, by calculating a more difficult PoW, the lifetime of the proof -- and thus the persistence of the revocation message -- can be increased on demand by the zone owner.

The parameters are defined as follows:

Z: The number of PoWs that are required. Its value is fixed at 32.

D: The lower limit of the average difficulty. Its value is fixed at 22.

EPOCH: A single epoch. Its value is fixed at 365 days in microseconds.

The revocation message wire format is illustrated in Figure 5.

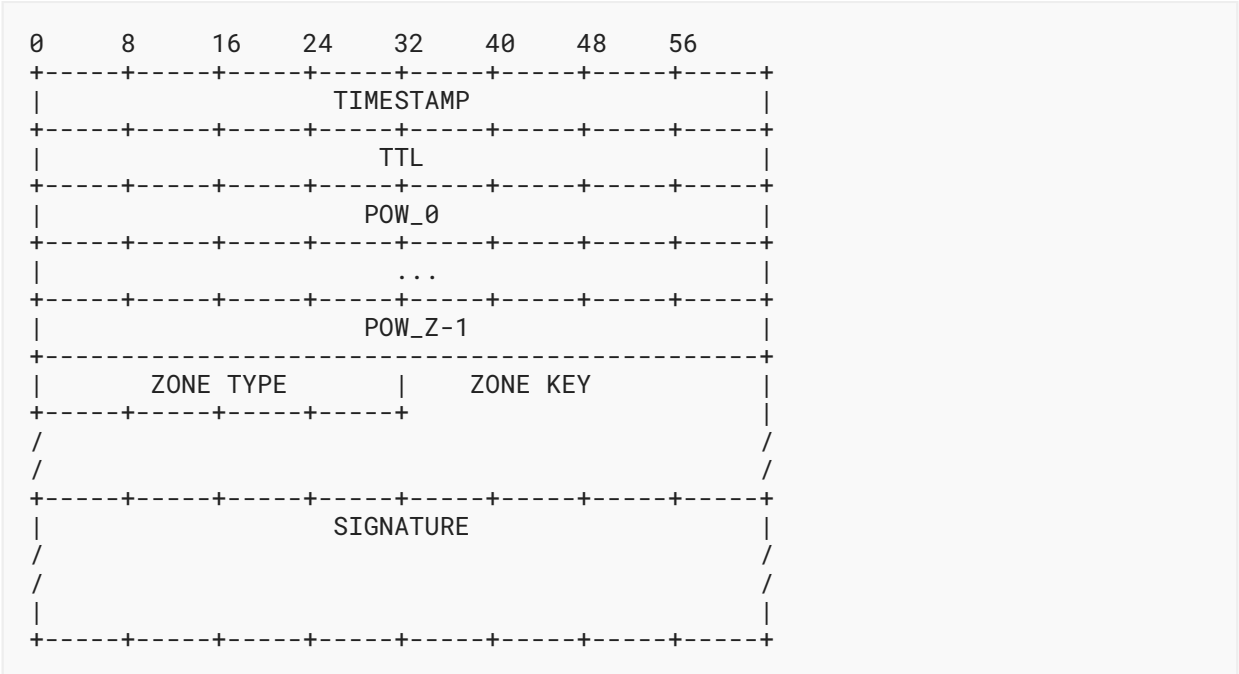

Figure 5: The Revocation Message Wire Format

TIMESTAMP:

Denotes the absolute 64-bit date when the revocation was computed. In microseconds since midnight (0 hour), January 1, 1970 UTC in network byte order. This is the same value as the timestamp used in the individual PoW calculations.

TTL: Denotes the relative 64-bit time to live of the record in microseconds in network byte order. The field **SHOULD** be set to EPOCH * 1.1. Given an average number of leading zeroes D', then the field value **MAY** be increased up to (D'-D+1) * EPOCH * 1.1. Validators **MAY** reject messages with lower or higher values when received.

POW_i: The values calculated as part of the PoW, in network byte order. Each POW_i **MUST** be unique in the set of POW values. To facilitate fast verification of uniqueness, the POW values must be given in strictly monotonically increasing order in the message.

ZONE TYPE: The 32-bit zone type corresponding to the zone key in network byte order.

ZONE KEY: The public key zk of the zone that is being revoked and the key to be used to verify SIGNATURE.

SIGNATURE: A signature over a timestamp and the zone zk of the zone that is revoked and corresponds to the key used in the PoW. The signature is created using the Sign() function of the cryptosystem of the zone and the private key (see Section 4).

The signature over the public key covers a 32-bit header prefixed to the timestamp and public key fields. The header includes the key length and signature purpose. The wire format is illustrated in Figure 6.

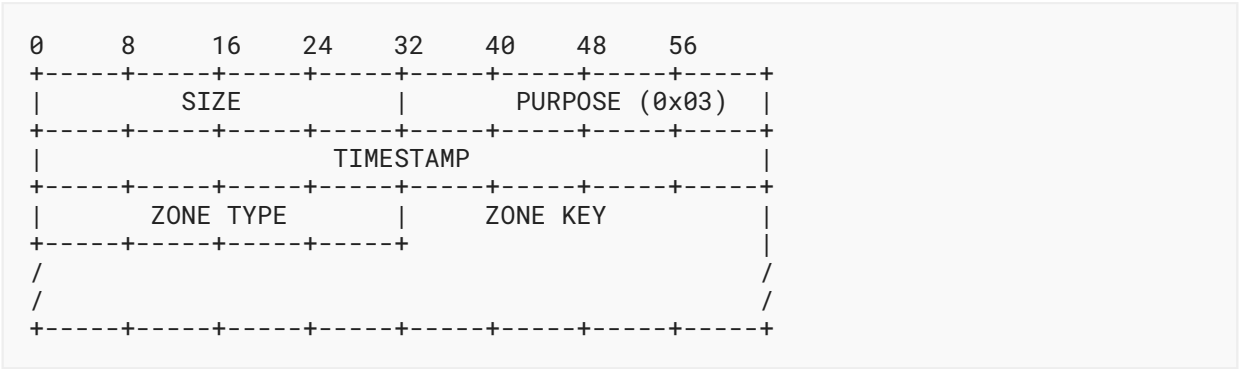

Figure 6: The Wire Format of the Revocation Data for Signing

SIZE: A 32-bit value containing the length of the signed data in bytes in network byte order.

PURPOSE: A 32-bit signature purpose flag. The value of this field **MUST** be 3. The value is encoded in network byte order. It defines the context in which the signature is created so that it cannot be reused in other parts of the protocol including possible future extensions. The value of this field corresponds to an entry in the GANA "GNUnet Signature Purposes" registry [GANA].

TIMESTAMP: Field as defined in the revocation message above.

ZONE TYPE: Field as defined in the revocation message above.

ZONE KEY: Field as defined in the revocation message above.

In order to validate a revocation, the following steps **MUST** be taken:

1. The signature **MUST** be verified against the zone key.
2. The set of POW values **MUST NOT** contain duplicates; this **MUST** be checked by verifying that the values are strictly monotonically increasing.
3. The average number of leading zeroes D' resulting from the provided POW values **MUST** be greater than or equal to D. Implementers **MUST NOT** use an integer data type to calculate or represent D'.

The TTL field in the revocation message is informational. A revocation **MAY** be discarded without checking the POW values or the signature if the TTL (in combination with TIMESTAMP) indicates that the revocation has already expired. The actual validity period of the revocation **MUST** be determined by examining the leading zeroes in the POW values.

The validity period of the revocation is calculated as (D'-D+1) * EPOCH * 1.1. The EPOCH is extended by 10% in order to deal with unsynchronized clocks. The validity period added on top of the TIMESTAMP yields the expiration date. If the current time is after the expiration date, the revocation is considered stale.

Verified revocations **MUST** be stored locally. The implementation **MAY** discard stale revocations and evict them from the local store at any time.

Implementations **MUST** broadcast received revocations if they are valid and not stale. Should the calculated validity period differ from the TTL field value, the calculated value **MUST** be used as the TTL field value when forwarding the revocation message. Systems might disagree on the current time, so implementations **MAY** use stale but otherwise valid revocations but **SHOULD NOT** broadcast them. Forwarded stale revocations **MAY** be discarded.

Any locally stored revocation **MUST** be considered during delegation record processing (see Section 7.3.4).

5. Resource Records

A GNS implementation **SHOULD** provide a mechanism for creating and managing local zones as well as a persistence mechanism (such as a local database) for resource records. A new local zone is established by selecting a zone type and creating a zone key pair. If this mechanism is not implemented, no zones can be published in storage (see Section 6) and name resolution is limited to non-local start zones (see Section 7.1).

A GNS resource record holds the data of a specific record in a zone. The resource record format is illustrated in Figure 7.

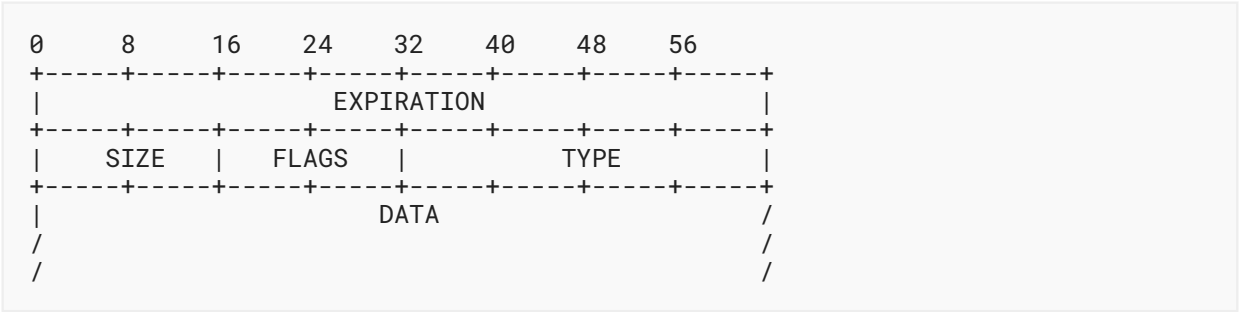

Figure 7: The Resource Record Wire Format

EXPIRATION: Denotes the absolute 64-bit expiration date of the record. In microseconds since midnight (0 hour), January 1, 1970 UTC in network byte order.

SIZE: Denotes the 16-bit size of the DATA field in bytes in network byte order.

FLAGS: A 16-bit bit field indicating special properties of the resource record. The semantics of the different bits are defined below.

TYPE: The 32-bit resource record type in network byte order. This type can be one of the GNS resource records as defined in Section 5, a DNS record type as defined in [RFC1035], or any of the complementary standardized DNS resource record types. Note that values below 2^16 are reserved for 16-bit DNS resource record types allocated by IANA [RFC6895]. Values above 2^16 are allocated by the GANA "GNS Record Types" registry [GANA].

DATA: The variable-length resource record data payload. The content is defined by the respective type of the resource record.

The FLAGS field is used to indicate special properties of the resource record. An application creating resource records **MUST** set all bits in FLAGS to 0 unless it specifically understands and wants to set the respective flag. As additional flags can be defined in future protocol versions, if an application or implementation encounters a flag that it does not recognize, the flag **MUST** be ignored. However, all implementations **MUST** understand the SHADOW and CRITICAL flags defined below. Any combination of the flags specified below is valid. Figure 8 illustrates the flag distribution in the 16-bit FLAGS field of a resource record:

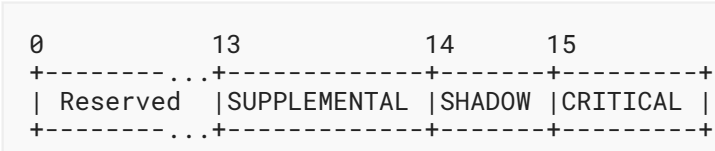

Figure 8: The Resource Record Flag Wire Format

CRITICAL:

If this flag is set, it indicates that processing is critical. Implementations that do not support the record type or are otherwise unable to process the record **MUST** abort resolution upon encountering the record in the resolution process.

SHADOW: If this flag is set, this record **MUST** be ignored by resolvers unless all (other) records of the same record type have expired. Used to allow zone publishers to facilitate good performance when records change by allowing them to put future values of records into storage. This way, future values can propagate and can be cached before the transition becomes active.

SUPPLEMENTAL: This is a supplemental record. It is provided in addition to the other records. This flag indicates that this record is not explicitly managed alongside the other records under the respective name but might be useful for the application.

5.1. Zone Delegation Records

This section defines the initial set of zone delegation record types. Any implementation **SHOULD** support all zone types defined here and **MAY** support any number of additional delegation records defined in the GANA "GNS Record Types" registry (see [GANA]). Not supporting some zone types will result in resolution failures if the respective zone type is encountered. This can be a valid choice if some zone delegation record types have been determined to be cryptographically insecure. Zone delegation records **MUST NOT** be stored and published under the apex label. A zone delegation record type value is the same as the respective ztype value. The ztype defines the cryptographic primitives for the zone that is being delegated to. A zone delegation record payload contains the public key of the zone to delegate to. A zone delegation record **MUST** have the CRITICAL flag set and **MUST** be the only non-supplemental record under a label. There **MAY** be inactive records of the same type that have the SHADOW flag set in order to facilitate smooth key rollovers.

In the following, "||" is the concatenation operator of two byte strings. The algorithm specification uses character strings such as GNS labels or constant values. When used in concatenations or as input to functions, the null-terminator of the character strings **MUST NOT** be included.

5.1.1. PKEY

In GNS, a delegation of a label to a zone of type "PKEY" is represented through a PKEY record. The PKEY DATA entry wire format is illustrated in Figure 9.

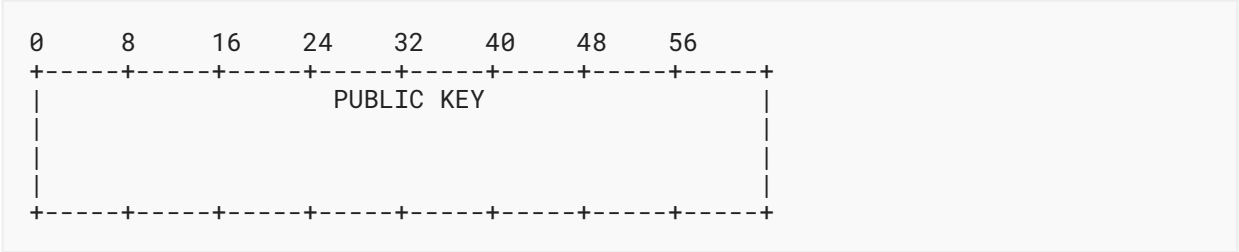

Figure 9: The PKEY Wire Format

PUBLIC KEY: A 256-bit Ed25519 public key.

For PKEY zones, the zone key material is derived using the curve parameters of the twisted Edwards representation of Curve25519 [RFC7748] (the reasoning behind choosing this curve can be found in Section 9.3) with the ECDSA scheme [RFC6979]. The following naming convention is used for the cryptographic primitives of PKEY zones:

d: A 256-bit Ed25519 private key (private scalar).

zk: The Ed25519 public zone key corresponding to d.

p: The prime of edwards25519 as defined in [RFC7748], i.e., 2^255 - 19.

G: The group generator (X(P),Y(P)). With X(P),Y(P) of edwards25519 as defined in [RFC7748].

L: The order of the prime-order subgroup of edwards25519 as defined in [RFC7748].

KeyGen(): The generation of the private scalar d and the curve point zk := d*G (where G is the group generator of the elliptic curve) as defined in Section 2.2 of [RFC6979] represents the KeyGen() function.

The zone type and zone key of a PKEY are 4 + 32 bytes in length. This means that a zTLD will always fit into a single label and does not need any further conversion. Given a label, the output zk' of the ZKDF(zk,label) function is calculated as follows for PKEY zones:

```
ZKDF(zk,label):
  PRK_h := HKDF-Extract ("key-derivation", zk)
  h := HKDF-Expand (PRK_h, label || "gns", 512 / 8)
  zk' := (h mod L) * zk
  return zk'
```

The PKEY cryptosystem uses an HMAC-based key derivation function (HKDF) as defined in [RFC5869], using SHA-512 [RFC6234] for the extraction phase and SHA-256 [RFC6234] for the expansion phase. PRK_h is key material retrieved using an HKDF using the string "key-derivation" as the salt and the zone key as the initial keying material. h is the 512-bit HKDF

expansion result and must be interpreted in network byte order. The expansion information input is a concatenation of the label and the string "gns". The multiplication of zk with h is a point multiplication, while the multiplication of d with h is a scalar multiplication.

The Sign() and Verify() functions for PKEY zones are implemented using 512-bit ECDSA deterministic signatures as specified in [RFC6979]. The same functions can be used for derived keys:

```
SignDerived(d,label,message):
  zk := d * G
  PRK_h := HKDF-Extract ("key-derivation", zk)
  h := HKDF-Expand (PRK_h, label || "gns", 512 / 8)
  d' := (h * d) mod L
  return Sign(d',message)
```

A signature (R,S) is valid if the following holds:

```
VerifyDerived(zk,label,message,signature):
  zk' := ZKDF(zk,label)
  return Verify(zk',message,signature)
```

The S-Encrypt() and S-Decrypt() functions use AES in counter mode as defined in [MODES] (CTR-AES256):

```
S-Encrypt(zk,label,expiration,plaintext):
  PRK_k := HKDF-Extract ("gns-aes-ctx-key", zk)
  PRK_n := HKDF-Extract ("gns-aes-ctx-iv", zk)
  K := HKDF-Expand (PRK_k, label, 256 / 8)
  NONCE := HKDF-Expand (PRK_n, label, 32 / 8)
  IV := NONCE || expiration || 0x0000000000000001
  return CTR-AES256(K, IV, plaintext)

S-Decrypt(zk,label,expiration,ciphertext):
  PRK_k := HKDF-Extract ("gns-aes-ctx-key", zk)
  PRK_n := HKDF-Extract ("gns-aes-ctx-iv", zk)
  K := HKDF-Expand (PRK_k, label, 256 / 8)
  NONCE := HKDF-Expand (PRK_n, label, 32 / 8)
  IV := NONCE || expiration || 0x0000000000000001
  return CTR-AES256(K, IV, ciphertext)
```

The key K and counter IV (Initialization Vector) are derived from the record label and the zone key zk, using an HKDF as defined in [RFC5869]. SHA-512 [RFC6234] is used for the extraction phase and SHA-256 [RFC6234] for the expansion phase. The output keying material is 32 bytes (256 bits) for the symmetric key and 4 bytes (32 bits) for the nonce. The symmetric key K is a 256-bit AES key [RFC3826].

The nonce is combined with a 64-bit IV and a 32-bit block counter as defined in [RFC3686]. The block counter begins with a value of 1, and it is incremented to generate subsequent portions of the key stream. The block counter is a 32-bit integer value in network byte order. The IV is the expiration time of the resource record block in network byte order. The resulting counter (IV) wire format is illustrated in Figure 10.

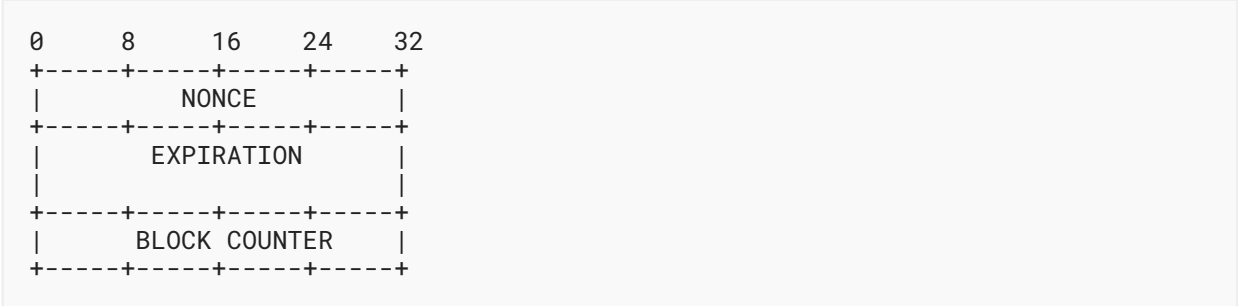

Figure 10: The Block Counter Wire Format

5.1.2. EDKEY

In GNS, a delegation of a label to a zone of type "EDKEY" is represented through an EDKEY record. The EDKEY DATA entry wire format is illustrated in Figure 11.

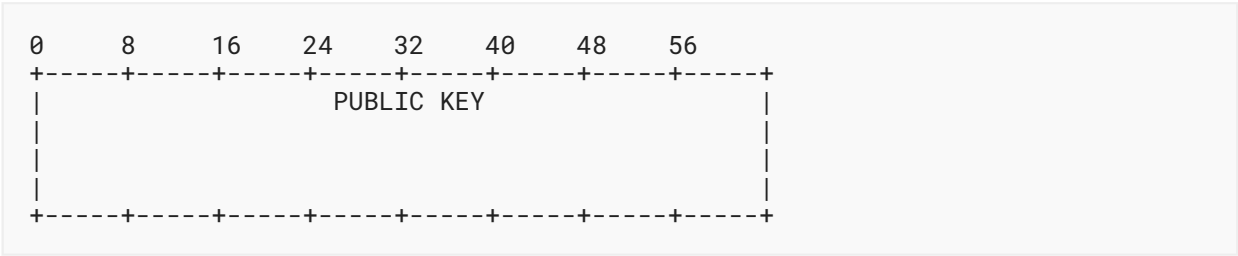

Figure 11: The EDKEY DATA Wire Format

PUBLIC KEY: A 256-bit EdDSA zone key.

For EDKEY zones, the zone key material is derived using the curve parameters of the twisted Edwards representation of Curve25519 [RFC7748] (a.k.a. Ed25519) with the Ed25519 scheme [ed25519] as specified in [RFC8032]. The following naming convention is used for the cryptographic primitives of EDKEY zones:

d: A 256-bit EdDSA private key.

a: An integer derived from d using the SHA-512 hash function as defined in [RFC8032].

zk: The EdDSA public key corresponding to d. It is defined as the curve point a*G where G is the group generator of the elliptic curve as defined in [RFC8032].

p: The prime of edwards25519 as defined in [RFC8032], i.e., 2^255 - 19.

G: The group generator (X(P),Y(P)). With X(P),Y(P) of edwards25519 as defined in [RFC8032].

L: The order of the prime-order subgroup of edwards25519 as defined in [RFC8032].

KeyGen(): The generation of the private key d and the associated public key zk := a*G where G is the group generator of the elliptic curve and a is an integer derived from d using the SHA-512 hash function as defined in Section 5.1.5 of [RFC8032] represents the KeyGen() function.

The zone type and zone key of an EDKEY are 4 + 32 bytes in length. This means that a zTLD will always fit into a single label and does not need any further conversion.

The "EDKEY" ZKDF instantiation is based on [Tor224]. The calculation of a is defined in Section 5.1.5 of [RFC8032]. Given a label, the output of the ZKDF function is calculated as follows:

```
ZKDF(zk,label):
  /* Calculate the blinding factor */
  PRK_h := HKDF-Extract ("key-derivation", zk)
  h := HKDF-Expand (PRK_h, label || "gns", 512 / 8)
  /* Ensure that h == h mod L */
  h[31] &= 7

  zk' := h * zk
  return zk'
```

Implementers **SHOULD** employ a constant-time scalar multiplication for the constructions above to protect against timing attacks. Otherwise, timing attacks could leak private key material if an attacker can predict when a system starts the publication process.

The EDKEY cryptosystem uses an HKDF as defined in [RFC5869], using SHA-512 [RFC6234] for the extraction phase and HMAC-SHA-256 [RFC6234] for the expansion phase. PRK_h is key material retrieved using an HKDF using the string "key-derivation" as the salt and the zone key as the initial keying material. The blinding factor h is the 512-bit HKDF expansion result. The expansion information input is a concatenation of the label and the string "gns". The result of the HKDF must be clamped and interpreted in network byte order. a is the 256-bit integer corresponding to the 256-bit private key d. The multiplication of zk with h is a point multiplication, while the division and multiplication of a and a1 with the cofactor are integer operations.

The Sign(d,message) and Verify(zk,message,signature) procedures **MUST** be implemented as defined in [RFC8032].

Signatures for EDKEY zones use a derived private scalar d'; this is not compliant with [RFC8032]. As the corresponding private key to the derived private scalar is not known, it is not possible to deterministically derive the signature part R according to [RFC8032]. Instead, signatures **MUST** be generated as follows for any given message and private zone key: a nonce is calculated from the highest 32 bytes of the expansion of the private key d and the blinding factor h. The nonce is then

hashed with the message to r. This way, the full derivation path is included in the calculation of the R value of the signature, ensuring that it is never reused for two different derivation paths or messages.

```
SignDerived(d,label,message):
  /* Key expansion */
  dh := SHA-512 (d)
  /* EdDSA clamping */
  a := dh[0..31]
  a[0] &= 248
  a[31] &= 127
  a[31] |= 64
  /* Calculate zk corresponding to d */
  zk := a * G

  /* Calculate blinding factor */
  PRK_h := HKDF-Extract ("key-derivation", zk)
  h := HKDF-Expand (PRK_h, label || "gns", 512 / 8)
  /* Ensure that h == h mod L */
  h[31] &= 7

  zk' := h * zk
  a1 := a >> 3
  a2 := (h * a1) mod L
  d' := a2 << 3
  nonce := SHA-256 (dh[32..63] || h)
  r := SHA-512 (nonce || message)
  R := r * G
  S := r + SHA-512(R || zk' || message) * d' mod L
  return (R,S)
```

A signature (R,S) is valid if the following holds:

```
VerifyDerived(zk,label,message,signature):
  zk' := ZKDF(zk,label)
  (R,S) := signature
  return S * G == R + SHA-512(R, zk', message) * zk'
```

The S-Encrypt() and S-Decrypt() functions use XSalsa20 as defined in [XSalsa20] (XSalsa20-Poly1305):

```
S-Encrypt(zk,label,expiration,plaintext):
  PRK_k := HKDF-Extract ("gns-xsalsa-ctx-key", zk)
  PRK_n := HKDF-Extract ("gns-xsalsa-ctx-iv", zk)
  K := HKDF-Expand (PRK_k, label, 256 / 8)
  NONCE := HKDF-Expand (PRK_n, label, 128 / 8)
  IV := NONCE || expiration
  return XSalsa20-Poly1305(K, IV, plaintext)

S-Decrypt(zk,label,expiration,ciphertext):
  PRK_k := HKDF-Extract ("gns-xsalsa-ctx-key", zk)
  PRK_n := HKDF-Extract ("gns-xsalsa-ctx-iv", zk)
  K := HKDF-Expand (PRK_k, label, 256 / 8)
  NONCE := HKDF-Expand (PRK_n, label, 128 / 8)
  IV := NONCE || expiration
  return XSalsa20-Poly1305(K, IV, ciphertext)
```

The result of the XSalsa20-Poly1305 encryption function is the encrypted ciphertext followed by the 128-bit authentication tag. Accordingly, the length of encrypted data equals the length of the data plus the 16 bytes of the authentication tag.

The key K and counter IV are derived from the record label and the zone key zk using an HKDF as defined in [RFC5869]. SHA-512 [RFC6234] is used for the extraction phase and SHA-256 [RFC6234] for the expansion phase. The output keying material is 32 bytes (256 bits) for the symmetric key and 16 bytes (128 bits) for the NONCE. The symmetric key K is a 256-bit XSalsa20 key [XSalsa20]. No additional authenticated data (AAD) is used.

The nonce is combined with an 8-byte IV. The IV is the expiration time of the resource record block in network byte order. The resulting counter (IV) wire format is illustrated in Figure 12.

Figure 12: The Counter Block Initialization Vector

5.2. Redirection Records

Redirect records are used to redirect resolution. Any implementation **SHOULD** support all redirection record types defined here and **MAY** support any number of additional redirection records defined in the GANA "GNS Record Types" registry [GANA]. Redirection records **MUST** have the CRITICAL flag set. Not supporting some record types can result in resolution failures.

This can be a valid choice if some redirection record types have been determined to be insecure, or if an application has reasons to not support redirection to DNS for reasons such as complexity or security. Redirection records **MUST NOT** be stored and published under the apex label.

5.2.1. REDIRECT

A REDIRECT record is the GNS equivalent of a CNAME record in DNS. A REDIRECT record **MUST** be the only non-supplemental record under a label. There **MAY** be inactive records of the same type that have the SHADOW flag set in order to facilitate smooth changes of redirection targets. No other records are allowed. Details on the processing of this record are provided in Section 7.3.1. A REDIRECT DATA entry is illustrated in Figure 13.

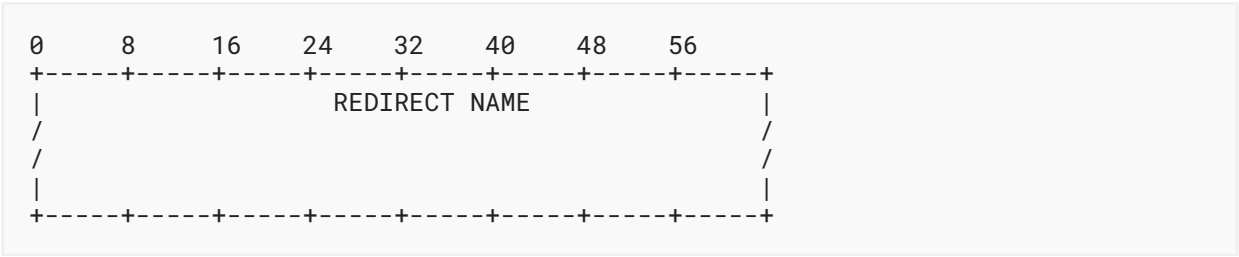

Figure 13: The REDIRECT DATA Wire Format

REDIRECT NAME: The name to continue with. The value of a redirect record can be a regular name or a relative name. Relative GNS names are indicated by an extension label (U+002B ("+")) as the rightmost label. The string is UTF-8 encoded and zero terminated.

5.2.2. GNS2DNS

A GNS2DNS record delegates resolution to DNS. The resource record contains a DNS name for the resolver to continue with in DNS followed by a DNS server. Both names are in the format defined in [RFC1034] for DNS names. There **MAY** be multiple GNS2DNS records under a label. There **MAY** also be DNSSEC DS records or any other records used to secure the connection with the DNS servers under the same label. There **MAY** be inactive records of the same type or types that have the SHADOW flag set in order to facilitate smooth changes of redirection targets. No other non-supplemental record types are allowed in the same record set. A GNS2DNS DATA entry is illustrated in Figure 14.

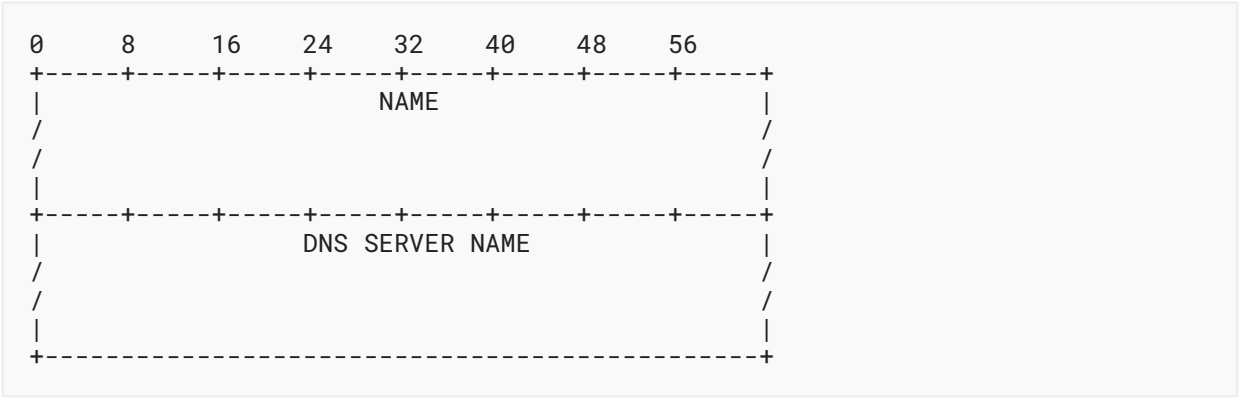

Figure 14: The GNS2DNS DATA Wire Format

NAME: The name to continue with in DNS. The value is UTF-8 encoded and zero terminated.

DNS SERVER NAME: The DNS server to use. This value can be an IPv4 address in dotted-decimal form, an IPv6 address in colon-hexadecimal form, or a DNS name. It can also be a relative GNS name ending with a "+" as the rightmost label. The implementation **MUST** check the string syntactically for an IP address in the respective notation before checking for a relative GNS name. If all three checks fail, the name **MUST** be treated as a DNS name. The value is UTF-8 encoded and zero terminated.

NOTE: If an application uses DNS names obtained from GNS2DNS records in a DNS request, they **MUST** first be converted to an IDNA-compliant representation [RFC5890].

5.3. Auxiliary Records

This section defines the initial set of auxiliary GNS record types. Any implementation **SHOULD** be able to process the specified record types according to Section 7.3.

5.3.1. LEHO

The LEHO (LEgacy HOstname) record is used to provide a hint for legacy hostnames: applications can use the GNS to look up IPv4 or IPv6 addresses of Internet services. However, connecting to such services sometimes not only requires the knowledge of an address and port but also requires the canonical DNS name of the service to be transmitted over the transport protocol. In GNS, legacy hostname records provide applications the DNS name that is required to establish a connection to such a service. The most common use case is HTTP virtual hosting and TLS Server Name Indication [RFC6066], where a DNS name must be supplied in the HTTP "Host"-header and the TLS handshake, respectively. Using a GNS name in those cases might not work, as it might not be globally unique. Furthermore, even if uniqueness is not an issue, the legacy service might not even be aware of GNS.

A LEHO resource record is expected to be found together in a single resource record with an IPv4 or IPv6 address. A LEHO DATA entry is illustrated in Figure 15.

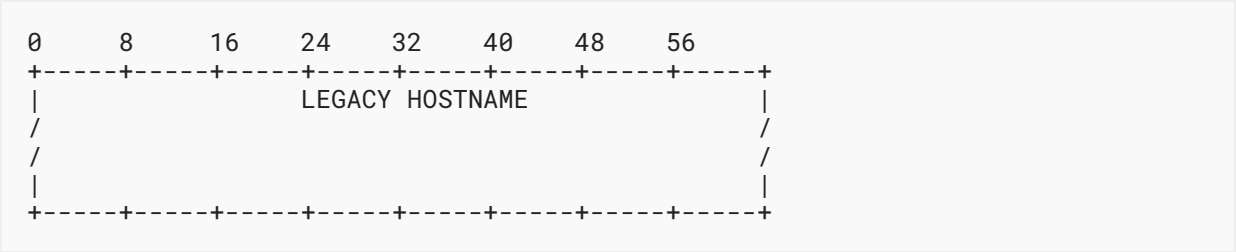

Figure 15: The LEHO DATA Wire Format

LEGACY HOSTNAME: A UTF-8 string (which is not zero terminated) representing the legacy hostname.

NOTE: If an application uses a LEHO value in an HTTP request header (e.g., a "Host:" header), it **MUST** be converted to an IDNA-compliant representation [RFC5890].

5.3.2. NICK

Nickname records can be used by zone administrators to publish a label that a zone prefers to have used when it is referred to. This is a suggestion for other zones regarding what label to use when creating a delegation record (Section 5.1) containing this zone key. This record **SHOULD** only be stored locally under the apex label "@" but **MAY** be returned with record sets under any label as a supplemental record. Section 7.3.5 details how a resolver must process supplemental and non-supplemental NICK records. A NICK DATA entry is illustrated in Figure 16.

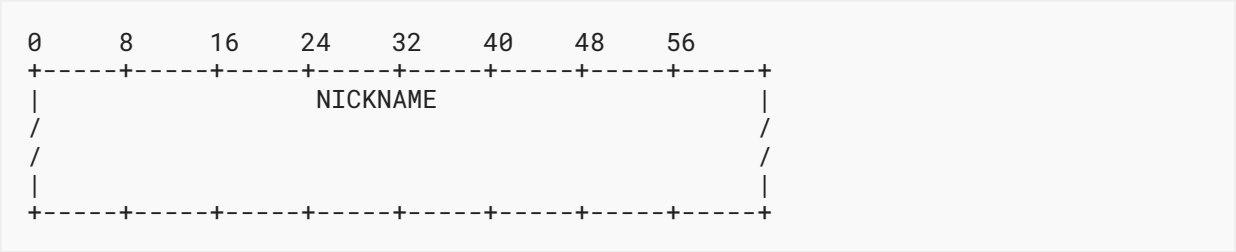

Figure 16: The NICK DATA Wire Format

NICKNAME: A UTF-8 string (which is not zero terminated) representing the preferred label of the zone. This string **MUST** be a valid GNS label.

5.3.3. BOX

GNS lookups are expected to return all of the required useful information in one record set. This avoids unnecessary additional lookups and cryptographically ties together information that belongs together, making it impossible for an adversarial storage entity to provide partial answers that might omit information critical for security.

This general strategy is incompatible with the special labels used by DNS for SRV and TLSA records. Thus, GNS defines the BOX record format to box up SRV and TLSA records and include them in the record set of the label they are associated with. For example, a TLSA record for "_https._tcp.example.org" will be stored in the record set of "example.org" as a BOX record with service (SVC) 443 (https), protocol (PROTO) 6 (tcp), and record TYPE "TLSA". For reference, see also [RFC2782]. A BOX DATA entry is illustrated in Figure 17.

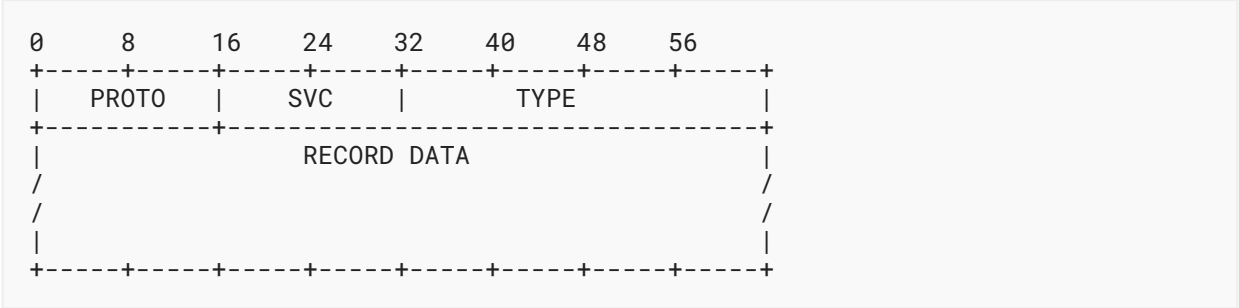

Figure 17: The BOX DATA Wire Format

PROTO: The 16-bit protocol number in network byte order. Values below 2^8 are reserved for 8-bit Internet Protocol numbers allocated by IANA [RFC5237] (e.g., 6 for TCP). Values above 2^8 are allocated by the GANA "GNUnet Overlay Protocols" registry [GANA].

SVC: The 16-bit service value of the boxed record in network byte order. In the case of TCP and UDP, it is the port number.

TYPE: The 32-bit record type of the boxed record in network byte order.

RECORD DATA: A variable-length field containing the "DATA" format of TYPE as defined for the respective TYPE. Thus, for TYPE values below 2^16, the format is the same as the respective record type's binary format in DNS.

6. Record Encoding for Remote Storage

Any API that allows storing a block under a 512-bit key and retrieving one or more blocks from a key can be used by an implementation for remote storage. To be useful, the API **MUST** permit storing at least 176 byte blocks to be able to support the defined zone delegation record encodings and **SHOULD** allow at least 1024 byte blocks. In the following, it is assumed that an implementation realizes two procedures on top of storage:

```
PUT(key,block)
GET(key) -> block
```

A GNS implementation publishes blocks in accordance with the properties and recommendations of the underlying remote storage. This can include a periodic refresh operation to preserve the availability of published blocks.

There is no mechanism for explicitly deleting individual blocks from remote storage. However, blocks include an EXPIRATION field, which guides remote storage implementations to decide when to delete blocks. Given multiple blocks for the same key, remote storage implementations **SHOULD** try to preserve and return the block with the largest EXPIRATION value.

All resource records from the same zone sharing the same label are encrypted and published together in a single resource records block (RRBLOCK) in the remote storage under a key q, as illustrated in Figure 18. A GNS implementation **MUST NOT** include expired resource records in blocks. An implementation **MUST** use the PUT storage procedure when record sets change to update the zone contents. Implementations **MUST** ensure that the EXPIRATION fields of RRBLOCKs increase strictly monotonically for every change, even if the smallest expiration time of records in the block does not.

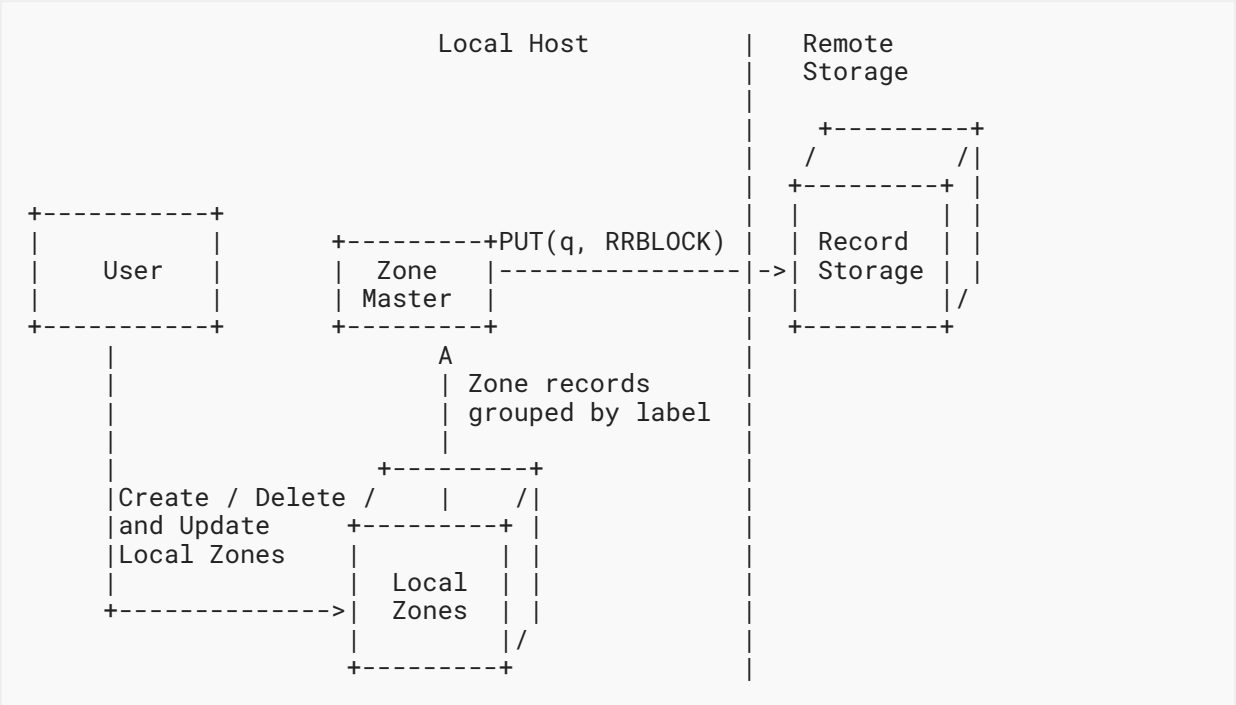

Figure 18: Management and Publication of Local Zones in Distributed Storage

Storage key derivation and records block creation are specified in the following sections and illustrated in Figure 19.

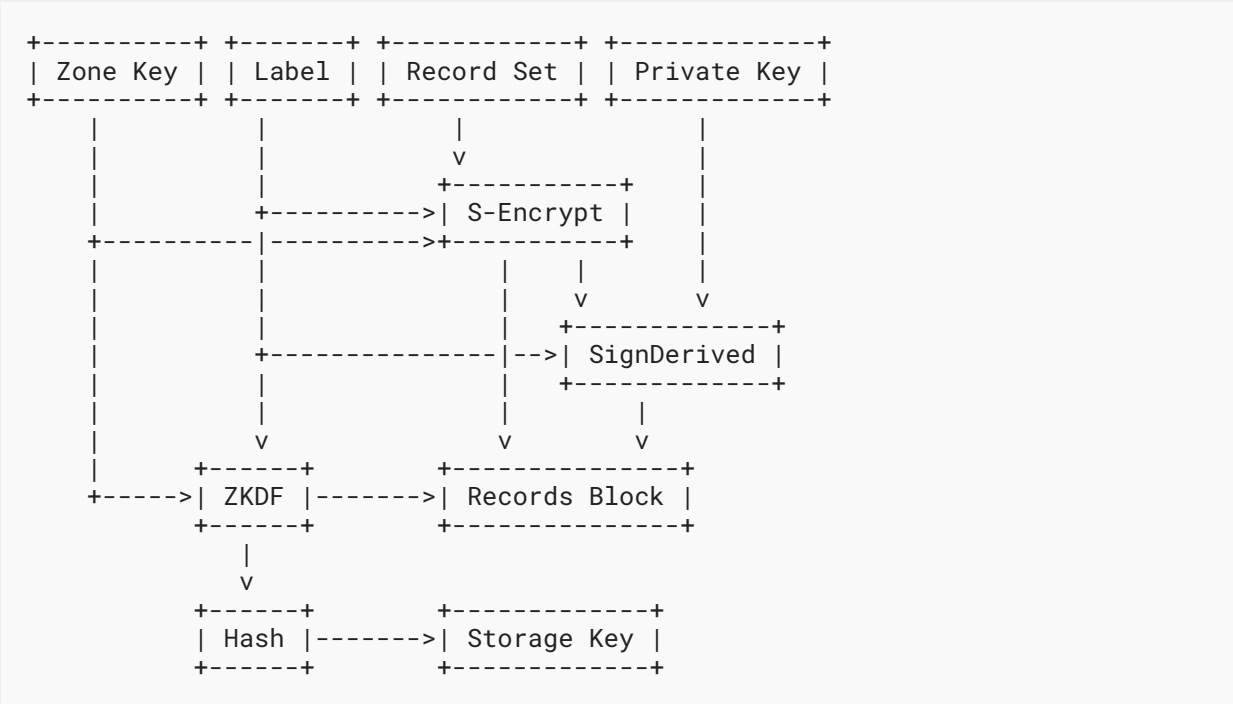

Figure 19: Storage Key and Records Block Creation Overview

6.1. The Storage Key

The storage key is derived from the zone key and the respective label of the contained records. The required knowledge of both the zone key and the label in combination with the similarly derived symmetric secret keys and blinded zone keys ensures query privacy (see [RFC8324], Section 3.5).

Given a label, the storage key q is derived as follows:

```
q := SHA-512 (ZKDF(zk, label))
```

label: A UTF-8 string under which the resource records are published.

zk: The zone key.

q: The 512-bit storage key under which the resource records block is published. It is the SHA-512 hash [RFC6234] over the derived zone key.

6.2. Plaintext Record Data (RDATA)

GNS records from a zone are grouped by their labels such that all records under the same label are published together as a single block in storage. Such grouped record sets **MAY** be paired with supplemental records.

Record data (RDATA) is the format used to encode such a group of GNS records. The binary format of RDATA is illustrated in Figure 20.

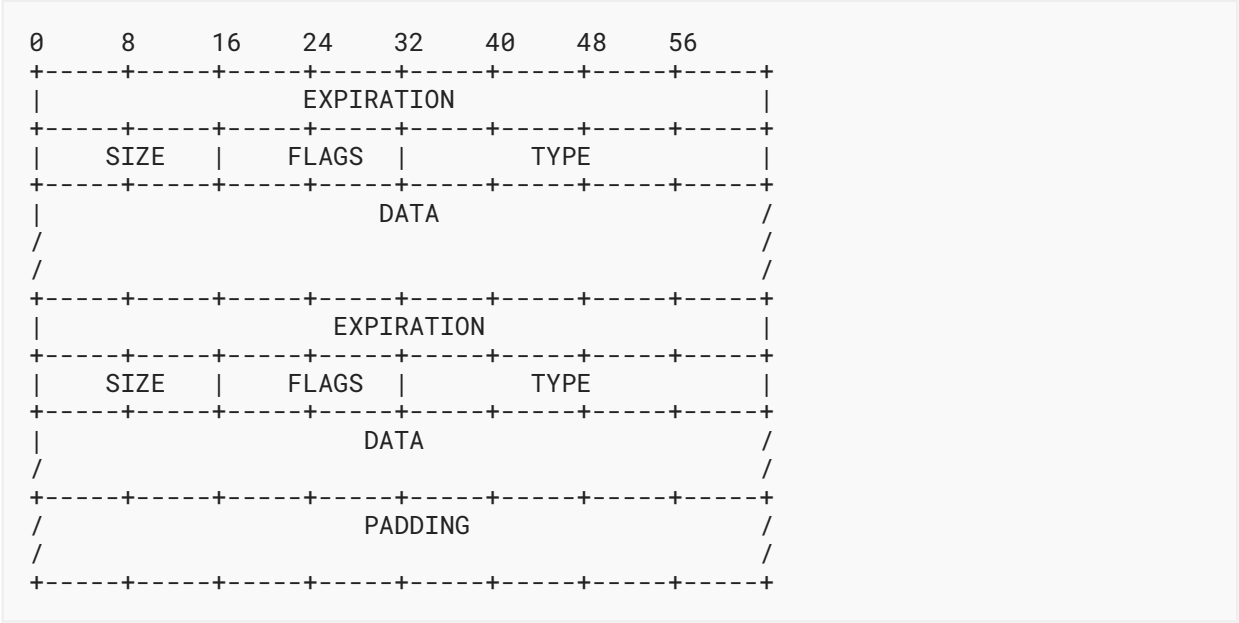

Figure 20: The RDATA Wire Format

EXPIRATION, SIZE, TYPE, FLAGS, and DATA: Definitions for these fields are provided below Figure 7 in Section 5.

PADDING: When serializing records into RDATA, a GNS implementation **MUST** ensure that the size of the RDATA is a power of two using this field. The field **MUST** be set to zero and **MUST** be ignored on receipt. As a special exception, record sets with (only) a zone delegation record type are never padded.

6.3. The Resource Records Block

The resource records grouped in an RDATA are encrypted using the S-Encrypt() function defined by the zone type of the zone to which the resource records belong and prefixed with metadata into a resource record block (RRBLOCK) for remote storage. The GNS RRBLOCK wire format is illustrated in Figure 21.

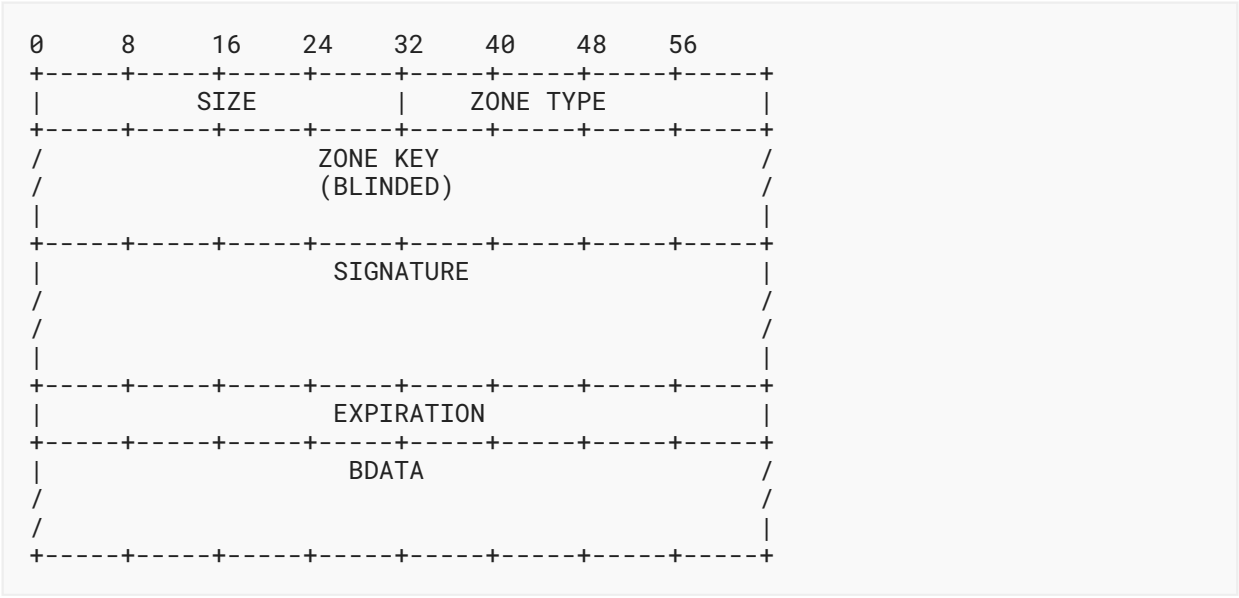

Figure 21: The RRBLOCK Wire Format

SIZE: A 32-bit value containing the length of the block in bytes in network byte order. Despite the message format's use of a 32-bit value, implementations **MAY** refuse to publish blocks beyond a certain size significantly below the theoretical block size limit of 4 GB.

ZONE TYPE: The 32-bit ztype in network byte order.

ZONE KEY (BLINDED): The blinded zone key "ZKDF(zk, label)" to be used to verify SIGNATURE. The length and format of the blinded public key depend on the ztype.

SIGNATURE: The signature is computed over the EXPIRATION and BDATA fields as shown in Figure 22. The length and format of the signature depend on the ztype. The signature is created using the SignDerived() function of the cryptosystem of the zone (see Section 4).

EXPIRATION: Specifies when the RRBLOCK expires and the encrypted block **SHOULD** be removed from storage and caches, as it is likely stale. However, applications **MAY** continue to use non-expired individual records until they expire. The value **MUST** be set to the maximum of the expiration time of the resource record contained within this block with the smallest expiration time and the previous EXPIRATION value (if any) plus one to ensure strict monotonicity (see Section 9.3). If the RDATA includes shadow records, then the maximum expiration time of all shadow records with matching type and the expiration times of the non-shadow records is considered. This is a 64-bit absolute date in microseconds since midnight (0 hour), January 1, 1970 UTC in network byte order.

BDATA: The encrypted RDATA computed using S-Encrypt() with the zone key, label, and expiration time as additional inputs. Its ultimate size and content are determined by the S-Encrypt() function of the ztype.

The signature over the public key covers a 32-bit pseudo header conceptually prefixed to the EXPIRATION and BDATA fields. The wire format is illustrated in Figure 22.

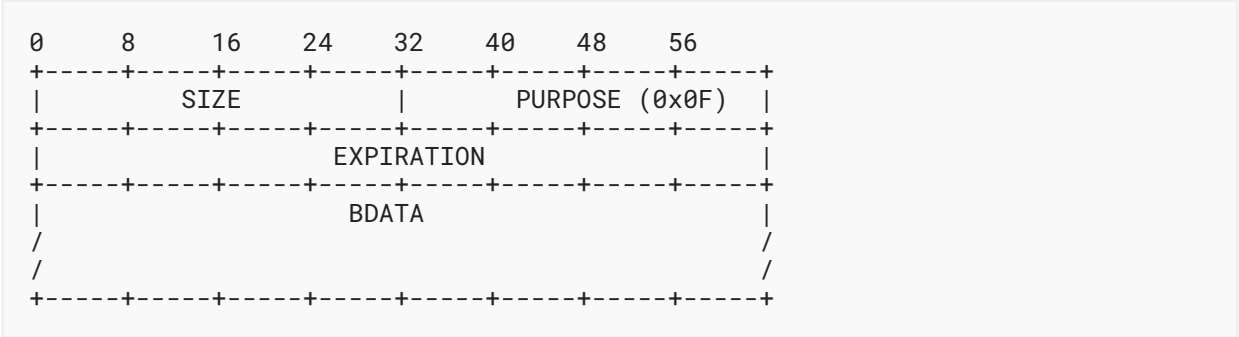

Figure 22: The Wire Format Used for Creating the Signature of the RRBLOCK

SIZE: A 32-bit value containing the length of the signed data in bytes in network byte order.

PURPOSE: A 32-bit signature purpose flag in network byte order. The value of this field **MUST** be 15. It defines the context in which the signature is created so that it cannot be reused in other parts of the protocol including possible future extensions. The value of this field corresponds to an entry in the GANA "GNUnet Signature Purposes" registry [GANA].

EXPIRATION: Field as defined in the RRBLOCK message above.

BDATA: Field as defined in the RRBLOCK message above.

7. Name Resolution

Names in GNS are resolved by recursively querying the record storage. Recursive in this context means that a resolver does not provide intermediate results for a query to the application. Instead, it **MUST** respond to a resolution request with either the requested resource record or an error message if resolution fails. Figure 23 illustrates how an application requests the lookup of a GNS name (1). The application **MAY** provide a desired record type to the resolver. Subsequently, a Start Zone is determined (2) and the recursive resolution process started. This is where the desired record type is used to guide processing. For example, if a zone delegation record type is requested, the resolution of the apex label in that zone must be skipped, as the desired record is already found. Details on how the resolution process is initiated and each iterative result (3a,3b) in the resolution is processed are provided in the sections below. The results of the lookup are eventually returned to the application (4). The implementation **MUST NOT** filter the returned resource record sets according to the desired record type. Filtering of record sets is typically done by the application.

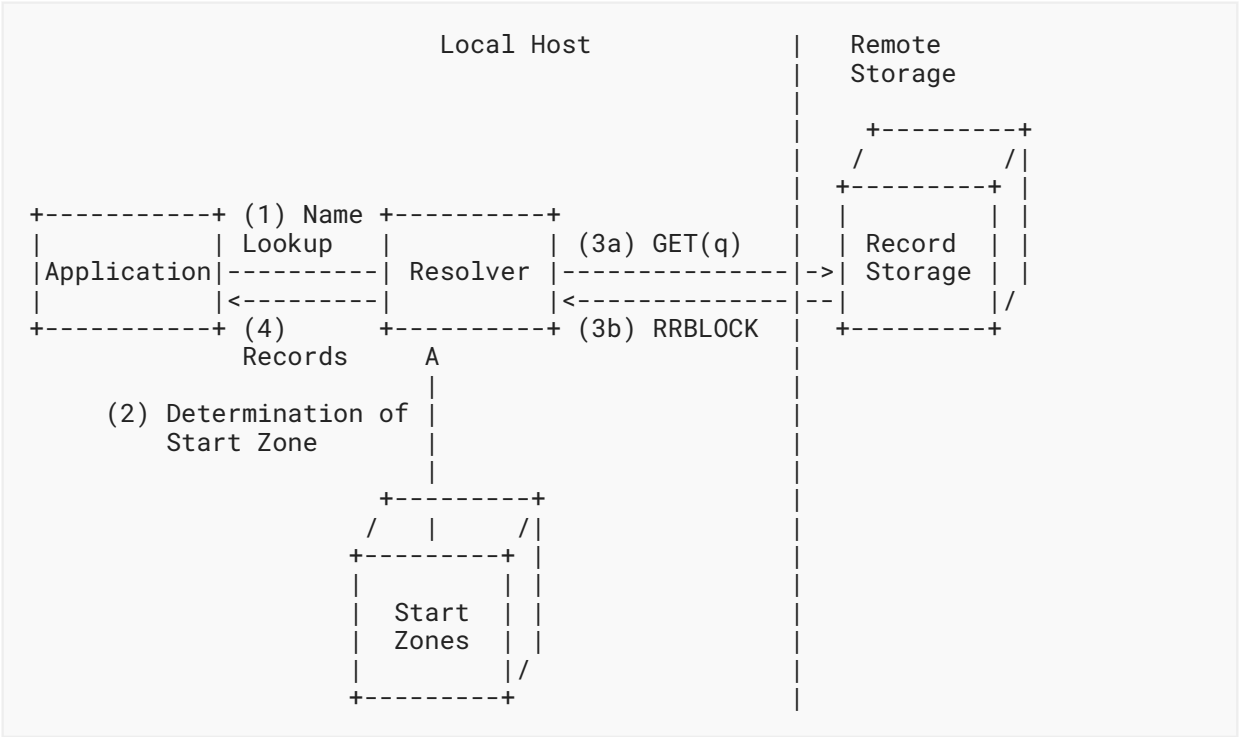

Figure 23: The Recursive GNS Resolution Process

7.1. Start Zones

The resolution of a GNS name starts by identifying the start zone suffix. Once the start zone suffix is identified, recursive resolution of the remainder of the name is initiated (see Section 7.2). There are two types of start zone suffixes: zTLDs and local suffix-to-zone mappings. The choice of available suffix-to-zone mappings is at the sole discretion of the local system administrator or user. This property addresses the issue of a single hierarchy with a centrally controlled root and the related issue of distribution and management of root servers in DNS (see Sections 3.12 and 3.10 of [RFC8324], respectively).

For names ending with a zTLD, the start zone is explicitly given in the suffix of the name to resolve. In order to ensure uniqueness of names with zTLDs, any implementation **MUST** use the given zone as the start zone. An implementation **MUST** first try to interpret the rightmost label of the given name as the beginning of a zTLD (see Section 4.1). If the rightmost label cannot be (partially) decoded or if it does not indicate a supported ztype, the name is treated as a normal name and start zone discovery **MUST** continue with finding a local suffix-to-zone mapping. If a valid ztype can be found in the rightmost label, the implementation **MUST** try to synthesize and decode the zTLD to retrieve the start zone key according to Section 4.1. If the zTLD cannot be synthesized or decoded, the resolution of the name fails and an error is returned to the application. Otherwise, the zone key **MUST** be used as the start zone:

```
Example name: www.example.<zTLD>
=> Start zone: zk of type ztype
=> Name to resolve from start zone: www.example
```

For names not ending with a zTLD, the resolver **MUST** determine the start zone through a local suffix-to-zone mapping. Suffix-to-zone mappings **MUST** be configurable through a local configuration file or database by the user or system administrator. A suffix **MAY** consist of multiple GNS labels concatenated with a label separator. If multiple suffixes match the name to resolve, the longest matching suffix **MUST** be used. The suffix length of two results **MUST NOT** be equal. This indicates a misconfiguration, and the implementation **MUST** return an error. The following is a non-normative example mapping of start zones:

```
Example name: www.example.xyz.gns.alt
Local suffix mappings:
xyz.gns.alt = zTLD0 := Base32GNS(ztype0||zk0)
example.xyz.gns.alt = zTLD1 := Base32GNS(ztype1||zk1)
example.com.gns.alt = zTLD2 := Base32GNS(ztype2||zk2)
...
=> Start zone: zk1
=> Name to resolve from start zone: www
```

The process given above **MAY** be supplemented with other mechanisms if the particular application requires a different process. If no start zone can be discovered, resolution **MUST** fail and an error **MUST** be returned to the application.

7.2. Recursion

In each step of the recursive name resolution, there is an authoritative zone zk and a name to resolve. The name **MAY** be empty. If the name is empty, it is interpreted as the apex label "@". Initially, the authoritative zone is the start zone.

From here, the following steps are recursively executed, in order:

1. Extract the rightmost label from the name to look up.
2. Calculate q using the label and zk as defined in Section 6.1.
3. Perform a storage query GET(q) to retrieve the RRBLOCK.
4. Check that (a) the block is not expired, (b) the SHA-512 hash of the derived authoritative zone key zk' from the RRBLOCK matches the query q, and (c) the signature is valid. If any of these tests fail, the RRBLOCK **MUST** be ignored and, if applicable, the storage lookup GET(q) **MUST** continue to look for other RRBLOCKs.
5. Obtain the RDATA by decrypting the BDATA contained in the RRBLOCK using S-Decrypt() as defined by the zone type, effectively inverting the process described in Section 6.3.

Once a well-formed block has been decrypted, the records from RDATA are subjected to record processing.

7.3. Record Processing

In record processing, only the valid records obtained are considered. To filter records by validity, the resolver **MUST** at least check the expiration time and the FLAGS field of the respective record. Specifically, the resolver **MUST** disregard expired records. Furthermore, SHADOW and SUPPLEMENTAL flags can also exclude records from being considered. If the resolver encounters a record with the CRITICAL flag set and does not support the record type, the resolution **MUST** be aborted and an error **MUST** be returned. Information indicating that the critical record could not be processed **SHOULD** be returned in the error description. The implementation **MAY** choose not to return the reason for the failure, merely complicating troubleshooting for the user.

The next steps depend on the context of the name that is being resolved:

Case 1: If the filtered record set consists of a single REDIRECT record, the remainder of the name is prepended to the REDIRECT data and the recursion is started again from the resulting name. Details are provided in Section 7.3.1.

Case 2: If the filtered record set consists exclusively of one or more GNS2DNS records, resolution continues with DNS. Details are provided in Section 7.3.2.

Case 3: If the remainder of the name to be resolved is of the format "_SERVICE._PROTO" and the record set contains one or more matching BOX records, the records in the BOX records are the final result and the recursion is concluded as described in Section 7.3.3.

Case 4: If the current record set consists of a single delegation record, resolution of the remainder of the name is delegated to the target zone as described in Section 7.3.4.

Case 5: If the remainder of the name to resolve is empty, the record set is the final result. If any NICK records are in the final result set, they **MUST** first be processed according to Section 7.3.5. Otherwise, the record result set is directly returned as the final result.

Finally, if none of the above cases are applicable, resolution fails and the resolver **MUST** return an empty record set.

7.3.1. REDIRECT

If the remaining name is empty and the desired record type is REDIRECT, the resolution concludes with the REDIRECT record. If the rightmost label of the redirect name is the extension label (U+002B ("+")), resolution continues in GNS with the new name in the current zone. Otherwise, the resulting name is resolved via the default operating system name resolution process. This can in turn trigger a GNS name resolution process, depending on the system configuration. If resolution continues in DNS, the name **MUST** first be converted to an IDNA-compliant representation [RFC5890].

In order to prevent infinite loops, the resolver **MUST** implement loop detection or limit the number of recursive resolution steps. The loop detection **MUST** be effective even if a REDIRECT found in GNS triggers subsequent GNS lookups via the default operating system name resolution process.

7.3.2. GNS2DNS

When a resolver encounters one or more GNS2DNS records, the remaining name is empty, and the desired record type is GNS2DNS, the GNS2DNS records are returned.

Otherwise, it is expected that the resolver first resolves the IP addresses of the specified DNS name servers. The DNS name **MUST** be converted to an IDNA-compliant representation [RFC5890] for resolution in DNS. GNS2DNS records **MAY** contain numeric IPv4 or IPv6 addresses, allowing the resolver to skip this step. The DNS server names might themselves be names in GNS or DNS. If the rightmost label of the DNS server name is the extension label (U+002B ("+")), the rest of the name is to be interpreted relative to the zone of the GNS2DNS record. If the DNS server name ends in a label representation of a zone key, the DNS server name is to be resolved against the GNS zone zk.

Multiple GNS2DNS records can be stored under the same label, in which case the resolver **MUST** try all of them. The resolver **MAY** try them in any order or even in parallel. If multiple GNS2DNS records are present, the DNS name **MUST** be identical for all of them. Otherwise, it is not clear which name the resolver is supposed to follow. If different DNS names are present, the resolution fails and an appropriate error **SHOULD** be returned to the application.

If there are DNSSEC DS records or any other records used to secure the connection with the DNS servers stored under the label, the DNS resolver **SHOULD** use them to secure the connection with the DNS server.

Once the IP addresses of the DNS servers have been determined, the DNS name from the GNS2DNS record is appended to the remainder of the name to be resolved and is resolved by querying the DNS name server(s). The synthesized name has to be converted to an IDNA-compliant representation [RFC5890] for resolution in DNS. If such a conversion is not possible, the resolution **MUST** be aborted and an error **MUST** be returned. Information indicating that the critical record could not be processed **SHOULD** be returned in the error description. The implementation **MAY** choose not to return the reason for the failure, merely complicating troubleshooting for the user.

As the DNS servers specified are possibly authoritative DNS servers, the GNS resolver **MUST** support recursive DNS resolution and **MUST NOT** delegate this to the authoritative DNS servers. The first successful recursive name resolution result is returned to the application. In addition, the resolver **SHOULD** return the queried DNS name as a supplemental LEHO record (see Section 5.3.1) with a relative expiration time of one hour.

Once the transition from GNS to DNS is made through a GNS2DNS record, there is no "going back". The (possibly recursive) resolution of the DNS name **MUST NOT** delegate back into GNS and should only follow the DNS specifications. For example, names contained in DNS CNAME records **MUST NOT** be interpreted by resolvers that support both DNS and GNS as GNS names.

GNS resolvers **SHOULD** offer a configuration option to disable DNS processing to avoid information leakage and provide a consistent security profile for all name resolutions. Such resolvers would return an empty record set upon encountering a GNS2DNS record during the recursion. However, if GNS2DNS records are encountered in the record set for the apex label and a GNS2DNS record is explicitly requested by the application, such records **MUST** still be returned, even if DNS support is disabled by the GNS resolver configuration.

7.3.3. BOX

When a BOX record is received, a GNS resolver must unbox it if the name to be resolved continues with "_SERVICE._PROTO". Otherwise, the BOX record is to be left untouched. This way, TLSA (and SRV) records do not require a separate network request, and TLSA records become inseparable from the corresponding address records.

7.3.4. Zone Delegation Records

When the resolver encounters a record of a supported zone delegation record type (such as PKEY or EDKEY) and the remainder of the name is not empty, resolution continues recursively with the remainder of the name in the GNS zone specified in the delegation record.

Whenever a resolver encounters a new GNS zone, it **MUST** check against the local revocation list (see Section 4.2) to see whether the respective zone key has been revoked. If the zone key was revoked, the resolution **MUST** fail with an empty result set.

Implementations **MUST NOT** allow multiple different zone delegations under a single label (except if some are shadow records). Implementations **MAY** support any subset of ztypes. Implementations **MUST NOT** process zone delegation records stored under the apex label ("@"). If a zone delegation record is encountered under the apex label, resolution fails and an error **MUST** be returned. The implementation **MAY** choose not to return the reason for the failure, merely impacting troubleshooting information for the user.

If the remainder of the name to resolve is empty and a record set was received containing only a single delegation record, the recursion is continued with the record value as authoritative zone and the apex label "@" as remaining name. Except in the case where the desired record type as specified by the application is equal to the ztype, in which case the delegation record is returned.

7.3.5. NICK

NICK records are only relevant to the recursive resolver if the record set in question is the final result, which is to be returned to the application. The encountered NICK records can be either supplemental (see Section 5) or non-supplemental. If the NICK record is supplemental, the resolver only returns the record set if one of the non-supplemental records matches the queried record type. It is possible that one record set contains both supplemental and non-supplemental NICK records.

The differentiation between a supplemental and non-supplemental NICK record allows the application to match the record to the authoritative zone. Consider the following example:

```
Query: alice.example.gns.alt (type=A)
Result:
A: 192.0.2.1
NICK: eve (non-supplemental)
```

In this example, the returned NICK record is non-supplemental. For the application, this means that the NICK belongs to the zone "alice.example.gns.alt" and is published under the apex label along with an A record. The NICK record is interpreted as follows: the zone defined by "alice.example.gns.alt" wants to be referred to as "eve". In contrast, consider the following:

```
Query: alice.example.gns.alt (type=AAAA)
Result:
AAAA: 2001:db8::1
NICK: john (supplemental)
```

In this case, the NICK record is marked as supplemental. This means that the NICK record belongs to the zone "example.gns.alt" and is published under the label "alice" along with a AAAA record. Here, the NICK record should be interpreted as follows: the zone defined by "example.gns.alt" wants to be referred to as "john". This distinction is likely useful for other records published as supplemental.

8. Internationalization and Character Encoding

All names in GNS are encoded in UTF-8 [RFC3629]. Labels **MUST** be canonicalized using Normalization Form C (NFC) [Unicode-UAX15]. This does not include any DNS names found in DNS records, such as CNAME record data, which is internationalized through the IDNA specifications; see [RFC5890].

9. Security and Privacy Considerations

9.1. Availability

In order to ensure availability of records beyond their absolute expiration times, implementations **MAY** allow relative expiration time values of records to be locally defined. Records can then be published recurringly with updated absolute expiration times by the implementation.

Implementations **MAY** allow users to manage private records in their zones that are not published in storage. Private records are considered just like regular records when resolving labels in local zones, but their data is completely unavailable to non-local users.

9.2. Agility

The security of cryptographic systems depends on both the strength of the cryptographic algorithms chosen and the strength of the keys used with those algorithms. This security also depends on the engineering of the protocol used by the system to ensure that there are no non-cryptographic ways to bypass the security of the overall system. This is why developers of applications managing GNS zones **SHOULD** select a default ztype considered secure at the time of releasing the software. For applications targeting end users that are not expected to understand cryptography, the application developer **MUST NOT** leave the ztype selection of new zones to end users.

This document concerns itself with the selection of cryptographic algorithms used in GNS. The algorithms identified in this document are not known to be broken (in the cryptographic sense) at the current time, and cryptographic research so far leads us to believe that they are likely to remain secure into the foreseeable future. However, this is not necessarily forever, and it is expected that new revisions of this document will be issued from time to time to reflect the current best practices in this area.

In terms of crypto-agility, whenever the need for an updated cryptographic scheme arises to, for example, replace ECDSA over Ed25519 for PKEY records, it can simply be introduced through a new record type. Zone administrators can then replace the delegation record type for future records. The old record type remains, and zones can iteratively migrate to the updated zone keys. To ensure that implementations correctly generate an error message when encountering a ztype that they do not support, current and future delegation records must always have the CRITICAL flag set.

9.3. Cryptography

The following considerations provide background on the design choices of the ztypes specified in this document. When specifying new ztypes as per Section 4, the same considerations apply.

GNS PKEY zone keys use ECDSA over Ed25519. This is an unconventional choice, as ECDSA is usually used with other curves. However, standardized ECDSA curves are problematic for a range of reasons described in the Curve25519 and EdDSA papers [ed25519]. Using EdDSA directly is also not possible, as a hash function is used on the private key which destroys the linearity that the key blinding in GNS depends upon. We are not aware of anyone suggesting that using Ed25519 instead of another common curve of similar size would lower the security of ECDSA. GNS uses 256-bit curves; that way, the encoded (public) keys fit into a single DNS label, which is good for usability.

In order to ensure ciphertext indistinguishability, care must be taken with respect to the IV in the counter block. In our design, the IV always includes the expiration time of the record block. When applications store records with relative expiration times, monotonicity is implicitly ensured because each time a block is published in storage, its IV is unique, as the expiration time is calculated dynamically and increases monotonically with the system time. Still, an implementation **MUST** ensure that when relative expiration times are decreased, the expiration

time of the next record block **MUST** be after the last published block. For records where an absolute expiration time is used, the implementation **MUST** ensure that the expiration time is always increased when the record data changes. For example, the expiration time on the wire could be increased by a single microsecond even if the user did not request a change. In the case of deletion of all resource records under a label, the implementation **MUST** keep track of the last absolute expiration time of the last published resource block. Implementations **MAY** define and use a special record type as a tombstone that preserves the last absolute expiration time but then **MUST** take care to not publish a block with such a tombstone record. When new records are added under this label later, the implementation **MUST** ensure that the expiration times are after the last published block. Finally, in order to ensure monotonically increasing expiration times, the implementation **MUST** keep a local record of the last time obtained from the system clock, so as to construct a monotonic clock if the system clock jumps backwards.

9.4. Abuse Mitigation

GNS names are UTF-8 strings. Consequently, GNS faces issues with respect to name spoofing similar to those for DNS with respect to internationalized domain names. In DNS, attackers can register similar-sounding or similar-looking names (see above) in order to execute phishing attacks. GNS zone administrators must take into account this attack vector and incorporate rules in order to mitigate it.

Further, DNS can be used to combat illegal content on the Internet by having the respective domains seized by authorities. However, the same mechanisms can also be abused in order to impose state censorship. Avoiding that possibility is one of the motivations behind GNS. In GNS, TLDs are not enumerable. By design, the start zone of the resolver is defined locally, and hence such a seizure is difficult and ineffective in GNS.

9.5. Zone Management

In GNS, zone administrators need to manage and protect their zone keys. Once a private zone key is lost, it cannot be recovered, and the zone revocation message cannot be computed anymore. Revocation messages can be precalculated if revocation is required in cases where a private zone key is lost. Zone administrators, and for GNS this includes end users, are required to responsibly and diligently protect their cryptographic keys. GNS supports signing records in advance ("offline") in order to support processes (such as air gaps) that aim to protect private keys.

Similarly, users are required to manage their local start zone configuration. In order to ensure the integrity and availability of names, users must ensure that their local start zone information is not compromised or outdated. It can be expected that the processing of zone revocations and an initial start zone is provided with a GNS implementation ("drop shipping"). Shipping an initial start zone configuration effectively establishes a root zone. Extension and customization of the zone are at the full discretion of the user.

While implementations following this specification will be interoperable, if two implementations connect to different remote storage entities, they are mutually unreachable. This can lead to a state where a record exists in the global namespace for a particular name, but the implementation is not communicating with the remote storage entity that contains the respective

block and is hence unable to resolve it. This situation is similar to a split-horizon DNS configuration. Which remote storage entities are implemented usually depends on the application it is built for. The remote storage entity used will most likely depend on the specific application context using GNS resolution. For example, one application is the resolution of hidden services within the Tor network, which would suggest using Tor routers for remote storage. Implementations of "aggregated" remote storage entities are conceivable but are expected to be the exception.

9.6. DHTs as Remote Storage

This document does not specify the properties of the underlying remote storage, which is required by any GNS implementation. It is important to note that the properties of the underlying remote storage are directly inherited by the GNS implementation. This includes both security and other non-functional properties such as scalability and performance. Implementers should take great care when selecting or implementing a DHT for use as remote storage in a GNS implementation. DHTs with reasonable security and performance properties exist [R5N]. It should also be taken into consideration that GNS implementations that build upon different DHT overlays are unlikely to be interoperable with each other.

9.7. Revocations

Zone administrators are advised to pregenerate zone revocations and to securely store the revocation information if the zone key is lost, compromised, or replaced in the future. Precalculated revocations can cease to be valid due to expirations or protocol changes such as epoch adjustments. Consequently, implementers and users must take precautions in order to manage revocations accordingly.

Revocation payloads do not include a "new" key for key replacement. Inclusion of such a key would have two major disadvantages:

1. If a revocation is published after a private key was compromised, allowing key replacement would be dangerous: if an adversary took over the private key, the adversary could then broadcast a revocation with a key replacement. For the replacement, the compromised owner would have no chance to issue a revocation. Thus, allowing a revocation message to replace a private key makes dealing with key compromise situations worse.
2. Sometimes, key revocations are used with the objective of changing cryptosystems. Migration to another cryptosystem by replacing keys via a revocation message would only be secure as long as both cryptosystems are still secure against forgery. Such a planned, non-emergency migration to another cryptosystem should be done by running zones for both cipher systems in parallel for a while. The migration would conclude by revoking the legacy zone key only when it is deemed no longer secure and, hopefully, after most users have migrated to the replacement.

9.8. Zone Privacy

GNS does not support authenticated denial of existence of names within a zone. Record data is published in encrypted form using keys derived from the zone key and record label. Zone administrators should carefully consider whether (1) a label and zone key are public or (2) one or both of these keys should be used as a shared secret to restrict access to the corresponding record data. Unlike public zone keys, low-entropy labels can be guessed by an attacker. If an attacker knows the public zone key, the use of well-known or guessable labels effectively threatens the disclosure of the corresponding records.

It should be noted that the guessing attack on labels only applies if the zone key is somehow disclosed to the adversary. GNS itself does not disclose it during a lookup or when resource records are published (as only the blinded zone keys are used on the network). However, zone keys do become public during revocation.

It is thus **RECOMMENDED** to use a label with sufficient entropy to prevent guessing attacks if any data in a resource record set is sensitive.

9.9. Zone Governance

While DNS is distributed, in practice it relies on centralized, trusted registrars to provide globally unique names. As awareness of the central role DNS plays on the Internet increases, various institutions are using their power (including legal means) to engage in attacks on the DNS, thus threatening the global availability and integrity of information on the Internet. While a wider discussion of this issue is out of scope for this document, analyses and investigations can be found in recent academic research works, including [SecureNS].

GNS is designed to provide a secure, privacy-enhancing alternative to the DNS name resolution protocol, especially when censorship or manipulation is encountered. In particular, it directly addresses concerns in DNS with respect to query privacy. However, depending on the governance of the root zone, any deployment will likely suffer from the issue of a single hierarchy with a centrally controlled root and the related issue of distribution and management of root servers in DNS, as raised in Sections 3.12 and 3.10 of [RFC8324], respectively. In DNS, those issues directly result from the centralized root zone governance at the Internet Corporation for Assigned Names and Numbers (ICANN), which allows it to provide globally unique names.

In GNS, start zones give users local authority over their preferred root zone governance. It enables users to replace or enhance a trusted root zone configuration provided by a third party (e.g., the implementer or a multi-stakeholder governance body like ICANN) with secure delegation of authority using local petnames while operating under a very strong adversary model. In combination with zTLDs, this provides users of GNS with a global, secure, and memorable mapping without a trusted authority.

Any GNS implementation **MAY** provide a default governance model in the form of an initial start zone mapping.

9.10. Namespace Ambiguity

Technically, the GNS protocol can be used to resolve names in the namespace of the global DNS. However, this would require the respective governance bodies and stakeholders (e.g., the IETF and ICANN) to standardize the use of GNS for this particular use case.

However, this capability implies that GNS names may be indistinguishable from DNS names in their respective common display format [RFC8499] or other special-use domain names [RFC6761] if a local start zone configuration maps suffixes from the global DNS to GNS zones. For applications, which name system should be used in order to resolve a given name will then be ambiguous. This poses a risk when trying to resolve a name through DNS when it is actually a GNS name, as discussed in [RFC8244]. In such a case, the GNS name is likely to be leaked as part of the DNS resolution.

In order to prevent disclosure of queried GNS names, it is **RECOMMENDED** that GNS-aware applications try to resolve a given name in GNS before any other method, taking into account potential suffix-to-zone mappings and zTLDs. Suffix-to-zone mappings are expected to be configured by the user or local administrator, and as such the resolution in GNS is in line with user expectations even if the name could also be resolved through DNS. If no suffix-to-zone mapping for the name exists and no zTLD is found, resolution **MAY** continue with other methods such as DNS. If a suffix-to-zone mapping for the name exists or the name ends with a zTLD, it **MUST** be resolved using GNS, and resolution **MUST NOT** continue by any other means independent of the GNS resolution result.

Mechanisms such as the Name Service Switch (NSS) of UNIX-like operating systems are an example of how such a resolution process can be implemented and used. The NSS allows system administrators to configure hostname resolution precedence and is integrated with the system resolver implementation.

For use cases where GNS names may be confused with names of other name resolution mechanisms (in particular, DNS), the ".gns.alt" domain **SHOULD** be used. For use cases like implementing sinkholes to block malware sites or serving DNS domains via GNS to bypass censorship, GNS **MAY** be deliberately used in ways that interfere with resolution of another name system.

10. GANA Considerations

GANA [GANA] has assigned signature purposes in its "GNUnet Signature Purposes" registry as listed in Table 1.

Purpose	Name	References	Comment
3	GNS_REVOCATION	RFC 0000	GNS zone key revocation

Purpose	Name	References	Comment
15	GNS_RECORD_SIGN	RFC 0000	GNS record set signature

Table 1: Requested Changes in the GANA GNUnet Signature Purposes Registry

10.1. GNS Record Types Registry

GANA [GANA] manages the "GNS Record Types" registry. Each entry has the following format:

Name: The name of the record type (case-insensitive ASCII string, restricted to alphanumeric characters). For zone delegation records, the assigned number represents the ztype value of the zone.

Number: A 32-bit number above 65535.

Comment: Optionally, brief English text describing the purpose of the record type (in UTF-8).

Contact: Optionally, the contact information for a person to contact for further information.

References: Optionally, references (such as an RFC) describing the record type.

The registration policy for this registry is "First Come First Served". This policy is modeled on that described in [RFC8126] and describes the actions taken by GANA:

Adding new entries is possible after review by any authorized GANA contributor, using a first-come-first-served policy for unique name allocation. Reviewers are responsible for ensuring that the chosen "Name" is appropriate for the record type. The registry will define a unique number for the entry.

Authorized GANA contributors for review of new entries are reachable at mailto:gns-registry@gnunet.org.

Any request **MUST** contain a unique name and a point of contact. The contact information **MAY** be added to the registry, with the consent of the requester. The request **MAY** optionally also contain relevant references as well as a descriptive comment, as defined above.

GANA has assigned numbers for the record types defined in this specification in the "GNS Record Types" registry as listed in Table 2.

Number	Name	Contact	References	Comment
65536	PKEY	(*)	RFC 0000	GNS zone delegation (PKEY)
65537	NICK	(*)	RFC 0000	GNS zone nickname
65538	LEHO	(*)	RFC 0000	GNS legacy hostname
(*): mailto:gns-registry@gnunet.org				

Number	Name	Contact	References	Comment
65540	GNS2DNS	(*)	RFC 0000	Delegation to DNS
65541	BOX	(*)	RFC 0000	Boxed records
65551	REDIRECT	(*)	RFC 0000	Redirection record
65556	EDKEY	(*)	RFC 0000	GNS zone delegation (EDKEY)
(*): mailto:gns-registry@gnunet.org				

Table 2: The GANA GNS Record Types Registry

10.2. .alt Subdomains Registry

GANA [GANA] manages the ".alt Subdomains" registry. Each entry has the following format:

Label: The label of the subdomain (in DNS "letters, digits, hyphen" (LDH) format as defined in Section 2.3.1 of [RFC5890]).

Comment: Optionally, brief English text describing the purpose of the subdomain (in UTF-8).

Contact: Optionally, the contact information for a person to contact for further information.

References: Optionally, references (such as an RFC) describing the record type.

The registration policy for this registry is "First Come First Served". This policy is modeled on that described in [RFC8126] and describes the actions taken by GANA:

Adding new entries is possible after review by any authorized GANA contributor, using a first-come-first-served policy for unique subdomain allocation. Reviewers are responsible for ensuring that the chosen "Subdomain" is appropriate for the purpose.

Authorized GANA contributors for review of new entries are reachable at mailto:alt-registry@gnunet.org.

Any request **MUST** contain a unique subdomain and a point of contact. The contact information **MAY** be added to the registry, with the consent of the requester. The request **MAY** optionally also contain relevant references as well as a descriptive comment, as defined above.

GANA has assigned the subdomain defined in this specification in the ".alt Subdomains" registry as listed in Table 3.

Subdomain	Contact	References	Comment
gns	(*)	RFC 0000	The .alt subdomain for GNS
(*): mailto:alt-registry@gnunet.org			

Table 3: The GANA .alt Subdomains Registry

11. IANA Considerations

This document has no IANA actions.

12. Implementation and Deployment Status

There are two implementations conforming to this specification, written in C and Go, respectively. The C implementation as part of GNUnet [GNUnetGNS] represents the original and reference implementation. The Go implementation [GoGNS] demonstrates how two implementations of GNS are interoperable if they are built on top of the same underlying DHT storage.

Currently, the GNUnet peer-to-peer network [GNUnet] is an active deployment of GNS on top of its DHT [R5N]. The Go implementation [GoGNS] uses this deployment by building on top of the GNUnet DHT services available on any GNUnet peer. It shows how GNS implementations can attach to this existing deployment and participate in name resolution as well as zone publication.

The self-sovereign identity system re:claimID [reclaim] is using GNS in order to selectively share identity attributes and attestations with third parties.

The Ascension tool [Ascension] facilitates the migration of DNS zones to GNS zones by translating information retrieved from a DNS zone transfer into a GNS zone.

13. References

13.1. Normative References

[RFC1034] Mockapetris, P., "Domain names - concepts and facilities", STD 13, RFC 1034, DOI 10.17487/RFC1034, November 1987, <https://www.rfc-editor.org/info/rfc1034>.

[RFC1035] Mockapetris, P., "Domain names - implementation and specification", STD 13, RFC 1035, DOI 10.17487/RFC1035, November 1987, <https://www.rfc-editor.org/info/rfc1035>.

[RFC2782] Gulbrandsen, A., Vixie, P., and L. Esibov, "A DNS RR for specifying the location of services (DNS SRV)", RFC 2782, DOI 10.17487/RFC2782, February 2000, <https://www.rfc-editor.org/info/rfc2782>.

[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>.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, DOI 10.17487/RFC3629, November 2003, <https://www.rfc-editor.org/info/rfc3629>.

[RFC3686] Housley, R., "Using Advanced Encryption Standard (AES) Counter Mode With IPsec Encapsulating Security Payload (ESP)", RFC 3686, DOI 10.17487/RFC3686, January 2004, <https://www.rfc-editor.org/info/rfc3686>.

[RFC3826] Blumenthal, U., Maino, F., and K. McCloghrie, "The Advanced Encryption Standard (AES) Cipher Algorithm in the SNMP User-based Security Model", RFC 3826, DOI 10.17487/RFC3826, June 2004, <https://www.rfc-editor.org/info/rfc3826>.

[RFC5237] Arkko, J. and S. Bradner, "IANA Allocation Guidelines for the Protocol Field", BCP 37, RFC 5237, DOI 10.17487/RFC5237, February 2008, <https://www.rfc-editor.org/info/rfc5237>.

[RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <https://www.rfc-editor.org/info/rfc5869>.

[RFC5890] Klensin, J., "Internationalized Domain Names for Applications (IDNA): Definitions and Document Framework", RFC 5890, DOI 10.17487/RFC5890, August 2010, <https://www.rfc-editor.org/info/rfc5890>.

[RFC5895] Resnick, P. and P. Hoffman, "Mapping Characters for Internationalized Domain Names in Applications (IDNA) 2008", RFC 5895, DOI 10.17487/RFC5895, September 2010, <https://www.rfc-editor.org/info/rfc5895>.

[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>.

[RFC6895] Eastlake 3rd, D., "Domain Name System (DNS) IANA Considerations", BCP 42, RFC 6895, DOI 10.17487/RFC6895, April 2013, <https://www.rfc-editor.org/info/rfc6895>.

[RFC6979] Pornin, T., "Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)", RFC 6979, DOI 10.17487/RFC6979, August 2013, <https://www.rfc-editor.org/info/rfc6979>.

[RFC7748] Langley, A., Hamburg, M., and S. Turner, "Elliptic Curves for Security", RFC 7748, DOI 10.17487/RFC7748, January 2016, <https://www.rfc-editor.org/info/rfc7748>.

[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>.

[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>.

[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>.

[RFC8499] Hoffman, P., Sullivan, A., and K. Fujiwara, "DNS Terminology", BCP 219, RFC 8499, DOI 10.17487/RFC8499, January 2019, <https://www.rfc-editor.org/info/rfc8499>.

[RFC9106] Biryukov, A., Dinu, D., Khovratovich, D., and S. Josefsson, "Argon2 Memory-Hard Function for Password Hashing and Proof-of-Work Applications", RFC 9106, DOI 10.17487/RFC9106, September 2021, <https://www.rfc-editor.org/info/rfc9106>.

[GANA] GNUnet e.V., "GNUnet Assigned Numbers Authority (GANA)", 2023, <https://gana.gnunet.org/>.

[MODES] Dworkin, M., "Recommendation for Block Cipher Modes of Operation: Methods and Techniques", NIST Special Publication 800-38A, DOI 10.6028/NIST.SP.800-38A, December 2001, <https://doi.org/10.6028/NIST.SP.800-38A>.

[CrockfordB32] Crockford, D., "Base 32", March 2019, <https://www.crockford.com/base32.html>.

[XSalsa20] Bernstein, D. J., "Extending the Salsa20 nonce", 2011, <https://cr.yp.to/snuffle/xsalsa-20110204.pdf>.

[Unicode-UAX15] Davis, M., Whistler, K., and M. Dürst, "Unicode Standard Annex #15: Unicode Normalization Forms", Revision 31, The Unicode Consortium, Mountain View, September 2009, <https://www.unicode.org/reports/tr15/tr15-31.html>.

[Unicode-UTS46] Davis, M. and M. Suignard, "Unicode Technical Standard #46: Unicode IDNA Compatibility Processing", Revision 31, The Unicode Consortium, Mountain View, September 2023, <https://www.unicode.org/reports/tr46>.

13.2. Informative References

[RFC1928] Leech, M., Ganis, M., Lee, Y., Kuris, R., Koblas, D., and L. Jones, "SOCKS Protocol Version 5", RFC 1928, DOI 10.17487/RFC1928, March 1996, <https://www.rfc-editor.org/info/rfc1928>.

[RFC4033] Arends, R., Austein, R., Larson, M., Massey, D., and S. Rose, "DNS Security Introduction and Requirements", RFC 4033, DOI 10.17487/RFC4033, March 2005, <https://www.rfc-editor.org/info/rfc4033>.

[RFC6066] Eastlake 3rd, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, DOI 10.17487/RFC6066, January 2011, <https://www.rfc-editor.org/info/rfc6066>.

[RFC7363] Maenpaa, J. and G. Camarillo, "Self-Tuning Distributed Hash Table (DHT) for REsource LOcation And Discovery (RELOAD)", RFC 7363, DOI 10.17487/RFC7363, September 2014, <https://www.rfc-editor.org/info/rfc7363>.

[RFC8324] Klensin, J., "DNS Privacy, Authorization, Special Uses, Encoding, Characters, Matching, and Root Structure: Time for Another Look?", RFC 8324, DOI 10.17487/RFC8324, February 2018, <https://www.rfc-editor.org/info/rfc8324>.

[RFC8806] Kumari, W. and P. Hoffman, "Running a Root Server Local to a Resolver", RFC 8806, DOI 10.17487/RFC8806, June 2020, <https://www.rfc-editor.org/info/rfc8806>.

[RFC6761] Cheshire, S. and M. Krochmal, "Special-Use Domain Names", RFC 6761, DOI 10.17487/RFC6761, February 2013, <https://www.rfc-editor.org/info/rfc6761>.

[RFC8244] Lemon, T., Droms, R., and W. Kumari, "Special-Use Domain Names Problem Statement", RFC 8244, DOI 10.17487/RFC8244, October 2017, <https://www.rfc-editor.org/info/rfc8244>.

[RFC9476] Kumari, W. and P. Hoffman, "The .alt Special-Use Top-Level Domain", RFC 9476, DOI 10.17487/RFC9476, September 2023, <https://www.rfc-editor.org/info/rfc9476>.

[Tor224] Goulet, D., Kadianakis, G., and N. Mathewson, "Next-Generation Hidden Services in Tor", Appendix A.2 ("Tor's key derivation scheme"), November 2013, <https://gitweb.torproject.org/torspec.git/tree/proposals/224-rend-spec-ng.txt#n2135>.

[SDSI] Rivest, R. and B. Lampson, "SDSI - A Simple Distributed Security Infrastructure", April 1996, <http://people.csail.mit.edu/rivest/Sdsi10.ps>.

[Kademlia] Maymounkov, P. and D. Mazières, "Kademlia: A Peer-to-peer Information System Based on the XOR Metric", DOI 10.1007/3-540-45748-8_5, 2002, <https://css.csail.mit.edu/6.824/2014/papers/kademlia.pdf>.

[ed25519] Bernstein, D. J., Duif, N., Lange, T., Schwabe, P., and B-Y. Yang, "High-speed high-security signatures", DOI 10.1007/s13389-012-0027-1, 2011, <https://ed25519.cr.yp.to/ed25519-20110926.pdf>.

[GNS] Wachs, M., Schanzenbach, M., and C. Grothoff, "A Censorship-Resistant, Privacy-Enhancing and Fully Decentralized Name System", 13th International Conference on Cryptology and Network Security (CANS), DOI 10.13140/2.1.4642.3044, October 2014, <https://sci-hub.st/10.1007/978-3-319-12280-9_9>.

[R5N] Evans, N. S. and C. Grothoff, "R5N: Randomized Recursive Routing for Restricted-Route Networks", 5th International Conference on Network and System Security (NSS), DOI 10.1109/ICNSS.2011.6060022, September 2011, <https://sci-hub.st/10.1109/ICNSS.2011.6060022>.

[SecureNS] Grothoff, C., Wachs, M., Ermert, M., and J. Appelbaum, "Toward secure name resolution on the Internet", Computers and Security, Volume 77, Issue C, pp. 694-708, DOI 10.1016/j.cose.2018.01.018, August 2018, <https://sci-hub.st/https://doi.org/10.1016/j.cose.2018.01.018>.

[GNUnetGNS] GNUnet e.V., "The GNUnet GNS Implementation", 2023, <https://git.gnunet.org/gnunet.git/tree/src/gns>.

[Ascension] GNUnet e.V., "The Ascension Implementation", 2023, <https://git.gnunet.org/ascension.git>.

[GNUnet] GNUnet e.V., "The GNUnet Project", 2023, <https://gnunet.org>.

[reclaim] GNUnet e.V., "re:claimID - Self-sovereign, Decentralised Identity Management and Personal Data Sharing", 2023, <https://reclaim.gnunet.org>.

[GoGNS] Fix, B., "gnunet-go (Go GNS)", commit 5c815ba, July 2023, <https://github.com/bfix/gnunet-go/tree/master/src/gnunet/service/gns>.

[nsswitch] GNU Project, "System Databases and Name Service Switch (Section 29)", <https://www.gnu.org/software/libc/manual/html_node/Name-Service-Switch.html>.

Appendix A. Usage and Migration

This section outlines a number of specific use cases that may help readers of this technical specification better understand the protocol. The considerations below are not meant to be normative for the GNS protocol in any way. Instead, they are provided in order to give context and to provide some background on what the intended use of the protocol is by its designers. Further, this section provides pointers to migration paths.

A.1. Zone Dissemination

In order to become a zone owner, it is sufficient to generate a zone key and a corresponding secret key using a GNS implementation. At this point, the zone owner can manage GNS resource records in a local zone database. The resource records can then be published by a GNS implementation as defined in Section 6. For other users to resolve the resource records, the respective zone information must be disseminated first. The zone owner may decide to make the zone key and labels known to a selected set of users only or to make this information available to the general public.

Sharing zone information directly with specific users not only allows an implementation to potentially preserve zone and record privacy but also allows the zone owner and the user to establish strong trust relationships. For example, a bank may send a customer letter with a QR

code that contains the GNS zone of the bank. This allows the user to scan the QR code and establish a strong link to the zone of the bank and with it, for example, the IP address of the online banking web site.

Most Internet services likely want to make their zones available to the general public in the most efficient way possible. First, it is reasonable to assume that zones that are commanding high levels of reputation and trust are likely included in the default suffix-to-zone mappings of implementations. Hence, dissemination of a zone through delegation under such zones can be a viable path in order to disseminate a zone publicly. For example, it is conceivable that organizations such as ICANN or country-code TLD registrars also manage GNS zones and offer registration or delegation services.

Following best practices, particularly those related to security and abuse mitigation, are methods that allow zone owners and aspiring registrars to gain a good reputation and, eventually, trust. This includes, of course, diligent protection of private zone key material. Formalizing such best practices is out of scope for this specification and should be addressed in a separate document that takes Section 9 of this document into account.

A.2. Start Zone Configuration

A user is expected to install a GNS implementation if it is not already provided through other means such as the operating system or the browser. It is likely that the implementation ships with a default start zone configuration. This means that the user is able to resolve GNS names ending on a zTLD or ending on any suffix-to-name mapping that is part of the default start zone configuration. At this point, the user may delete or otherwise modify the implementation's default configuration:

Deletion of suffix-to-zone mappings may become necessary if the zone owner referenced by the mapping has lost the trust of the user. For example, this could be due to lax registration policies resulting in phishing activities. Modification and addition of new mappings are means to heal the namespace perforation that would occur in the case of a deletion or to simply establish a strong direct trust relationship. However, this requires the user's knowledge of the respective zone keys. This information must be retrieved out of band, as illustrated in Appendix A.1: a bank may send the user a letter with a QR code that contains the GNS zone of the bank. The user scans the QR code and adds a new suffix-to-name mapping using a chosen local name for his bank. Other examples include scanning zone information off the device of a friend, from a storefront, or from an advertisement. The level of trust in the respective zone is contextual and likely varies from user to user. Trust in a zone provided through a letter from a bank that may also include a credit card is certainly different from a zone found on a random advertisement on the street. However, this trust is immediately tangible to the user and can be reflected in the local naming as well.

Users that are also clients should facilitate the modification of the start zone configuration -- for example, by providing a QR code reader or other import mechanisms. Implementations are ideally implemented according to best practices and addressing applicable points from Section 9. Formalizing such best practices is out of scope for this specification.

A.3. Globally Unique Names and the Web

HTTP virtual hosting and TLS Server Name Indication (SNI) are common use cases on the Web. HTTP clients supply a DNS name in the HTTP "Host"-header or as part of the TLS handshake, respectively. This allows the HTTP server to serve the indicated virtual host with a matching TLS certificate. The global uniqueness of DNS names is a prerequisite of those use cases.

Not all GNS names are globally unique. However, any resource record in GNS can be represented as a concatenation of a GNS label and the zTLD of the zone. While not human readable, this globally unique GNS name can be leveraged in order to facilitate the same use cases. Consider the GNS name "www.example.gns" entered in a GNS-aware HTTP client. At first, "www.example.gns" is resolved using GNS, yielding a record set. Then, the HTTP client determines the virtual host as follows:

If there is a LEHO record (Section 5.3.1) containing "www.example.com" in the record set, then the HTTP client uses this as the value of the "Host"-header field of the HTTP request:

```
GET / HTTP/1.1
Host: www.example.com
```

If there is no LEHO record in the record set, then the HTTP client tries to find the zone of the record and translates the GNS name into a globally unique zTLD representation before using it in the "Host"-header field of the HTTP request:

```
GET / HTTP/1.1
Host: www.000G0037FH3QTBCK15Y8BCCNRVWPV17ZC7TSGB1C9ZG2TPGHZVFV1GMG3W
```

In order to determine the canonical representation of the record with a zTLD, at most two queries are required: first, it must be checked to see whether "www.example.gns.alt" itself points to a zone delegation record; this would imply that the record set that was originally resolved is published under the apex label. If it does, the unique GNS name is simply the zTLD representation of the delegated zone:

```
GET / HTTP/1.1
Host: 000G0037FH3QTBCK15Y8BCCNRVWPV17ZC7TSGB1C9ZG2TPGHZVFV1GMG3W
```

If it does not, the unique GNS name is the concatenation of the label "www" and the zTLD representation of the zone as given in the example above. In any case, this representation is globally unique. As such, it can be configured by the HTTP server administrator as a virtual hostname and respective certificates may be issued.

If the HTTP client is a browser, the use of a unique GNS name for virtual hosting or TLS SNI does not necessarily have to be shown to the user. For example, the name in the URL bar may remain as "www.example.gns.alt" even if the used unique name differs.

A.4. Migration Paths

DNS resolution is built into a variety of existing software components -- most significantly, operating systems and HTTP clients. This section illustrates possible migration paths for both in order to enable legacy applications to resolve GNS names.

One way to efficiently facilitate the resolution of GNS names is via GNS-enabled DNS server implementations. Local DNS queries are thereby either rerouted or explicitly configured to be resolved by a "DNS-to-GNS" server that runs locally. This DNS server tries to interpret any incoming query for a name as a GNS resolution request. If no start zone can be found for the name and it does not end in a zTLD, the server tries to resolve the name in DNS. Otherwise, the name is resolved in GNS. In the latter case, the resulting record set is converted to a DNS answer packet and is returned accordingly. An implementation of a DNS-to-GNS server can be found in [GNUnet].

A similar approach is to use operating system extensions such as the NSS [nsswitch]. It allows the system administrator to configure plugins that are used for hostname resolution. A GNS nsswitch plugin can be used in a fashion similar to that used for the DNS-to-GNS server. An implementation of a glibc-compatible nsswitch plugin for GNS can be found in [GNUnet].

The methods above are usually also effective for HTTP client software. However, HTTP clients are commonly used in combination with TLS. TLS certificate validation, and SNI in particular, require additional logic in HTTP clients when GNS names are in play (Appendix A.3). In order to transparently enable this functionality for migration purposes, a local GNS-aware SOCKS5 proxy [RFC1928] can be configured to resolve domain names. The SOCKS5 proxy, similar to the DNS-to-GNS server, is capable of resolving both GNS and DNS names. In the event of a TLS connection request with a GNS name, the SOCKS5 proxy can act as a man-in-the-middle, terminating the TLS connection and establishing a secure connection against the requested host. In order to establish a secure connection, the proxy may use LEHO and TLSA records stored in the record set under the GNS name. The proxy must provide a locally trusted certificate for the GNS name to the HTTP client; this usually requires the generation and configuration of a local trust anchor in the browser. An implementation of this SOCKS5 proxy can be found in [GNUnet].

Appendix B. Example Flows

B.1. AAAA Example Resolution

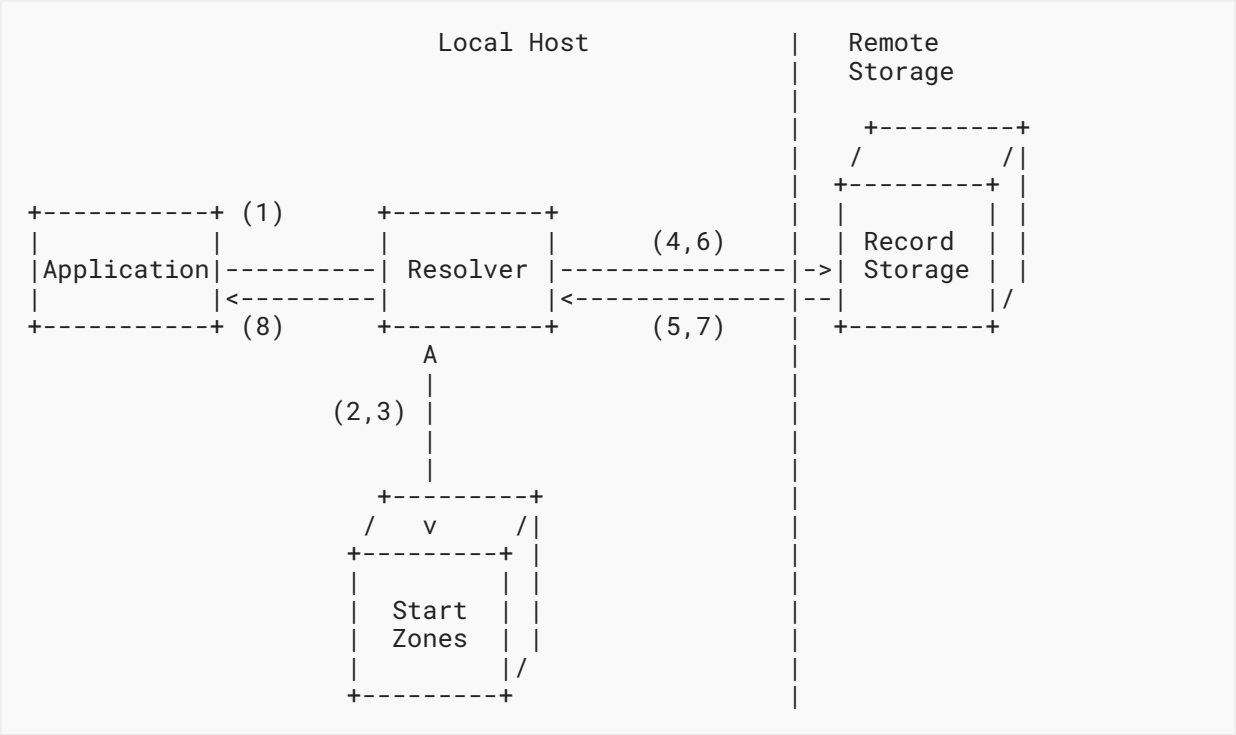

Figure 24: Example Resolution of an IPv6 Address

1. Look up AAAA record for name: www.example.gnu.gns.alt.
2. Determine start zone for www.example.gnu.gns.alt.
3. Start zone: zk0 - Remainder: www.example.
4. Calculate q0=SHA512(ZKDF(zk0, "example")) and initiate GET(q0).
5. Retrieve and decrypt RRBLOCK consisting of a single PKEY record containing zk1.
6. Calculate q1=SHA512(ZKDF(zk1, "www")) and initiate GET(q1).
7. Retrieve RRBLOCK consisting of a single AAAA record containing the IPv6 address 2001:db8::1.
8. Return record set to application.

B.2. REDIRECT Example Resolution

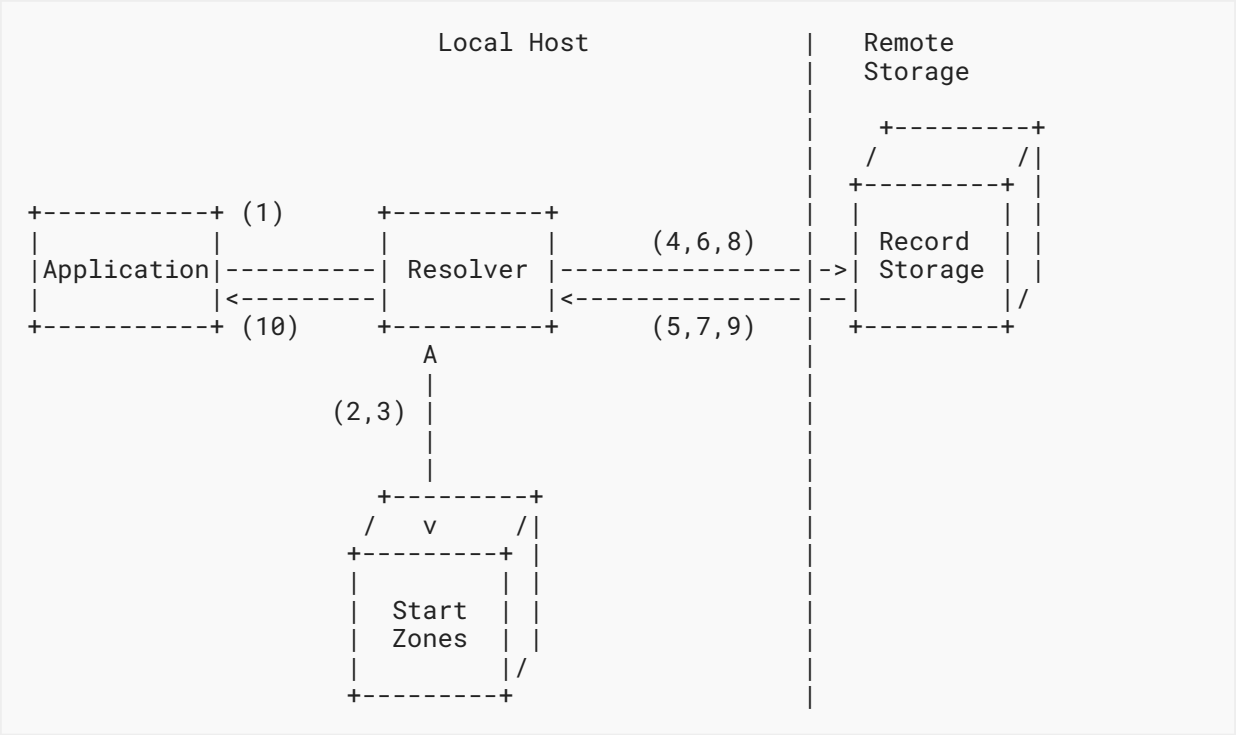

Figure 25: Example Resolution of an IPv6 Address with Redirect

1. Look up AAAA record for name: www.example.tld.gns.alt.
2. Determine start zone for www.example.tld.gns.alt.
3. Start zone: zk0 - Remainder: www.example.
4. Calculate q0=SHA512(ZKDF(zk0, "example")) and initiate GET(q0).
5. Retrieve and decrypt RRBLOCK consisting of a single PKEY record containing zk1.
6. Calculate q1=SHA512(ZKDF(zk1, "www")) and initiate GET(q1).
7. Retrieve and decrypt RRBLOCK consisting of a single REDIRECT record containing www2.+.
8. Calculate q2=SHA512(ZKDF(zk1, "www2")) and initiate GET(q2).
9. Retrieve and decrypt RRBLOCK consisting of a single AAAA record containing the IPv6 address 2001:db8::1.
10. Return record set to application.

B.3. GNS2DNS Example Resolution

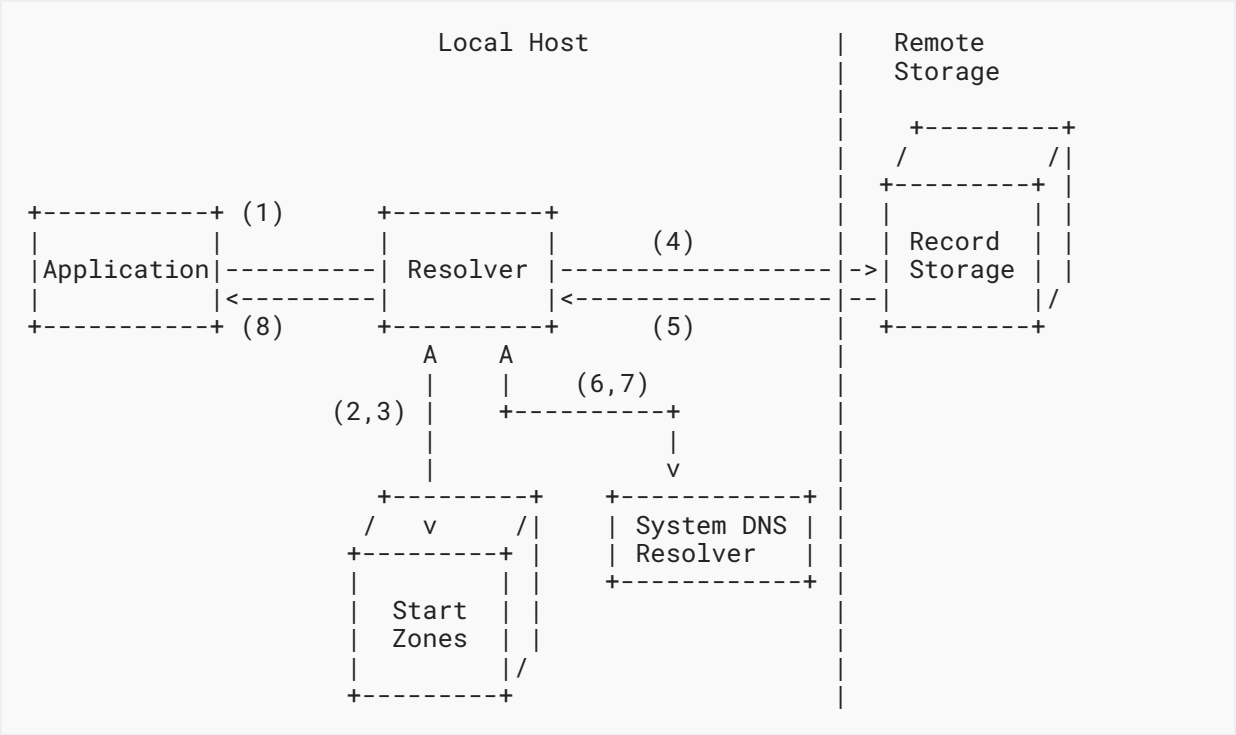

Figure 26: Example Resolution of an IPv6 Address with DNS Handover

1. Look up AAAA record for name: www.example.gnu.gns.alt.
2. Determine start zone for www.example.gnu.gns.alt.
3. Start zone: zk0 - Remainder: www.example.
4. Calculate q0=SHA512(ZKDF(zk0, "example")) and initiate GET(q0).
5. Retrieve and decrypt RRBLOCK consisting of a single GNS2DNS record containing the name "example.com" and the DNS server IPv4 address 192.0.2.1.
6. Use system resolver to look up a AAAA record for the DNS name www.example.com.
7. Retrieve a DNS reply consisting of a single AAAA record containing the IPv6 address 2001:db8::1.
8. Return record set to application.

Appendix C. Base32GNS

Encoding converts a byte array into a string of symbols. Decoding converts a string of symbols into a byte array. Decoding fails if the input string has symbols outside the defined set.

Table 4 defines the encoding and decoding symbols for a given symbol value. Each symbol value encodes 5 bits. It can be used to implement the encoding by reading it as follows: a symbol "A" or "a" is decoded to a 5-bit value 10 when decoding. A 5-bit block with a value of 18 is encoded to the character "J" when encoding. If the bit length of the byte string to encode is not a multiple of 5, it

is padded to the next multiple with zeroes. In order to further increase tolerance for failures in character recognition, the letter "U" **MUST** be decoded to the same value as the letter "V" in Base32GNS.

Symbol Value	Decoding Symbol	Encoding Symbol
0	0 O o	0
1	1 I i L l	1
2	2	2
3	3	3
4	4	4
5	5	5
6	6	6
7	7	7
8	8	8
9	9	9
10	A a	A
11	B b	B
12	C c	C
13	D d	D
14	E e	E
15	F f	F
16	G g	G
17	H h	H
18	J j	J
19	K k	K
20	M m	M
21	N n	N

Symbol Value	Decoding Symbol	Encoding Symbol
22	P p	P
23	Q q	Q
24	R r	R
25	S s	S
26	T t	T
27	V v U u	V
28	W w	W
29	X x	X
30	Y y	Y
31	Z z	Z

Table 4: The Base32GNS Alphabet, Including the Additional Encoding Symbol "U"

Appendix D. Test Vectors

The following test vectors can be used by implementations to test for conformance with this specification. Unless indicated otherwise, the test vectors are provided as hexadecimal byte arrays.

D.1. Base32GNS Encoding/Decoding

The following are test vectors for the Base32GNS encoding used for zTLDs. The input strings are encoded without the zero terminator.

```
Base32GNS-Encode:
  Input string: "Hello World"
  Output string: "91JPRV3F41BPYWKCCG"

  Input bytes: 474e55204e616d652053797374656d
  Output string: "8X75A82EC5PPA82KF5SQ8SBD"

Base32GNS-Decode:
  Input string: "91JPRV3F41BPYWKCCG"
  Output string: "Hello World"

  Input string: "91JPRU3F41BPYWKCCG"
  Output string: "Hello World"
```

D.2. Record Sets

The test vectors include record sets with a variety of record types and flags for both PKEY and EDKEY zones. This includes labels with UTF-8 characters to demonstrate internationalized labels.

(1) PKEY zone with ASCII label and one delegation record

```
Zone private key (d, big-endian):
  50 d7 b6 52 a4 ef ea df
  f3 73 96 90 97 85 e5 95
  21 71 a0 21 78 c8 e7 d4
  50 fa 90 79 25 fa fd 98

Zone identifier (ztype|zkey):
  00 01 00 00 67 7c 47 7d
  2d 93 09 7c 85 b1 95 c6
  f9 6d 84 ff 61 f5 98 2c
  2c 4f e0 2d 5a 11 fe df
  b0 c2 90 1f

zTLD:
000G0037FH3QTBCK15Y8BCCNRVWPV17ZC7TSGB1C9ZG2TPGHZVFV1GMG3W

Label:
  74 65 73 74 64 65 6c 65
  67 61 74 69 6f 6e

Number of records (integer): 1

Record #0 := (
  EXPIRATION: 8143584694000000 us
  00 1c ee 8c 10 e2 59 80

  DATA_SIZE:
  00 20

  TYPE:
  00 01 00 00

  FLAGS:   00 01

  DATA:
  21 e3 b3 0f f9 3b c6 d3
  5a c8 c6 e0 e1 3a fd ff
  79 4c b7 b4 4b bb c7 48
  d2 59 d0 a0 28 4d be 84

)

RDATA:
  00 1c ee 8c 10 e2 59 80
  00 20 00 01 00 01 00 00
  21 e3 b3 0f f9 3b c6 d3
  5a c8 c6 e0 e1 3a fd ff
  79 4c b7 b4 4b bb c7 48
  d2 59 d0 a0 28 4d be 84

Encryption NONCE|EXPIRATION|BLOCK COUNTER:
  e9 0a 00 61 00 1c ee 8c
  10 e2 59 80 00 00 00 01

Encryption key (K):
  86 4e 71 38 ea e7 fd 91
  a3 01 36 89 9c 13 2b 23
```

```
  ac eb db 2c ef 43 cb 19
  f6 bf 55 b6 7d b9 b3 b3

Storage key (q):
  4a dc 67 c5 ec ee 9f 76
  98 6a bd 71 c2 22 4a 3d
  ce 2e 91 70 26 c9 a0 9d
  fd 44 ce f3 d2 0f 55 a2
  73 32 72 5a 6c 8a fb bb
  b0 f7 ec 9a f1 cc 42 64
  12 99 40 6b 04 fd 9b 5b
  57 91 f8 6c 4b 08 d5 f4

ZKDF(zkey):
  18 2b b6 36 ed a7 9f 79
  57 11 bc 27 08 ad bb 24
  2a 60 44 6a d3 c3 08 03
  12 1d 03 d3 48 b7 ce b6

Derived private key (d', big-endian):
  0a 4c 5e 0f 00 63 df ce
  db c8 c7 f2 b2 2c 03 0c
  86 28 b2 c2 cb ac 9f a7
  29 aa e6 1f 89 db 3e 9c

BDATA:
  0c 1e da 5c c0 94 a1 c7
  a8 88 64 9d 25 fa ee bd
  60 da e6 07 3d 57 d8 ae
  8d 45 5f 4f 13 92 c0 74
  e2 6a c6 69 bd ee c2 34
  62 b9 62 95 2c c6 e9 eb

RRBLOCK:
  00 00 00 a0 00 01 00 00
  18 2b b6 36 ed a7 9f 79
  57 11 bc 27 08 ad bb 24
  2a 60 44 6a d3 c3 08 03
  12 1d 03 d3 48 b7 ce b6
  0a d1 0b c1 3b 40 3b 5b
  25 61 26 b2 14 5a 6f 60
  c5 14 f9 51 ff a7 66 f7
  a3 fd 4b ac 4a 4e 19 90
  05 5c b8 7e 8d 1b fd 19
  aa 09 a4 29 f7 29 e9 f5
  c6 ee c2 47 0a ce e2 22
  07 59 e9 e3 6c 88 6f 35
  00 1c ee 8c 10 e2 59 80
  0c 1e da 5c c0 94 a1 c7
  a8 88 64 9d 25 fa ee bd
  60 da e6 07 3d 57 d8 ae
  8d 45 5f 4f 13 92 c0 74
  e2 6a c6 69 bd ee c2 34
  62 b9 62 95 2c c6 e9 eb
```

(2) PKEY zone with UTF-8 label and three records

```
Zone private key (d, big-endian):
  50 d7 b6 52 a4 ef ea df
  f3 73 96 90 97 85 e5 95
  21 71 a0 21 78 c8 e7 d4
  50 fa 90 79 25 fa fd 98

Zone identifier (ztype|zkey):
  00 01 00 00 67 7c 47 7d
  2d 93 09 7c 85 b1 95 c6
  f9 6d 84 ff 61 f5 98 2c
  2c 4f e0 2d 5a 11 fe df
  b0 c2 90 1f

zTLD:
000G0037FH3QTBCK15Y8BCCNRVWPV17ZC7TSGB1C9ZG2TPGHZVFV1GMG3W

Label:
  e5 a4 a9 e4 b8 8b e7 84
  a1 e6 95 b5

Number of records (integer): 3

Record #0 := (
  EXPIRATION: 8143584694000000 us
  00 1c ee 8c 10 e2 59 80

  DATA_SIZE:
  00 10

  TYPE:
  00 00 00 1c

  FLAGS:   00 00

  DATA:
  00 00 00 00 00 00 00 00
  00 00 00 00 de ad be ef

)

Record #1 := (
  EXPIRATION: 17999736901000000 us
  00 3f f2 aa 54 08 db 40

  DATA_SIZE:
  00 06

  TYPE:
  00 01 00 01

  FLAGS:   00 00

  DATA:
  e6 84 9b e7 a7 b0

)
```

```
Record #2 := (
  EXPIRATION: 11464693629000000 us
  00 28 bb 13 ff 37 19 40

  DATA_SIZE:
  00 0b

  TYPE:
  00 00 00 10

  FLAGS:   00 04

  DATA:
  48 65 6c 6c 6f 20 57 6f
  72 6c 64

)

RDATA:
  00 1c ee 8c 10 e2 59 80
  00 10 00 00 00 00 00 1c
  00 00 00 00 00 00 00 00
  00 00 00 00 de ad be ef
  00 3f f2 aa 54 08 db 40
  00 06 00 00 00 01 00 01
  e6 84 9b e7 a7 b0 00 28
  bb 13 ff 37 19 40 00 0b
  00 04 00 00 00 10 48 65
  6c 6c 6f 20 57 6f 72 6c
  64 00 00 00 00 00 00 00
  00 00 00 00 00 00 00 00
  00 00 00 00 00 00 00 00
  00 00 00 00 00 00 00 00
  00 00 00 00 00 00 00 00
  00 00 00 00 00 00 00 00

Encryption NONCE|EXPIRATION|BLOCK COUNTER:
  ee 96 33 c1 00 1c ee 8c
  10 e2 59 80 00 00 00 01

Encryption key (K):
  fb 3a b5 de 23 bd da e1
  99 7a af 7b 92 c2 d2 71
  51 40 8b 77 af 7a 41 ac
  79 05 7c 4d f5 38 3d 01

Storage key (q):
  af f0 ad 6a 44 09 73 68
  42 9a c4 76 df a1 f3 4b
  ee 4c 36 e7 47 6d 07 aa
  64 63 ff 20 91 5b 10 05
  c0 99 1d ef 91 fc 3e 10
  90 9f 87 02 c0 be 40 43
  67 78 c7 11 f2 ca 47 d5
  5c f0 b5 4d 23 5d a9 77

ZKDF(zkey):
  a5 12 96 df 75 7e e2 75
```

```
  ca 11 8d 4f 07 fa 7a ae
  55 08 bc f5 12 aa 41 12
  14 29 d4 a0 de 9d 05 7e

Derived private key (d', big-endian):
  0a be 56 d6 80 68 ab 40
  e1 44 79 0c de 9a cf 4d
  78 7f 2d 3c 63 b8 53 05
  74 6e 68 03 32 15 f2 ab

BDATA:
  d8 c2 8d 2f d6 96 7d 1a
  b7 22 53 f2 10 98 b8 14
  a4 10 be 1f 59 98 de 03
  f5 8f 7e 7c db 7f 08 a6
  16 51 be 4d 0b 6f 8a 61
  df 15 30 44 0b d7 47 dc
  f0 d7 10 4f 6b 8d 24 c2
  ac 9b c1 3d 9c 6f e8 29
  05 25 d2 a6 d0 f8 84 42
  67 a1 57 0e 8e 29 4d c9
  3a 31 9f cf c0 3e a2 70
  17 d6 fd a3 47 b4 a7 94
  97 d7 f6 b1 42 2d 4e dd
  82 1c 19 93 4e 96 c1 aa
  87 76 57 25 d4 94 c7 64
  b1 55 dc 6d 13 26 91 74

RRBLOCK:
  00 00 00 f0 00 01 00 00
  a5 12 96 df 75 7e e2 75
  ca 11 8d 4f 07 fa 7a ae
  55 08 bc f5 12 aa 41 12
  14 29 d4 a0 de 9d 05 7e
  08 5b d6 5f d4 85 10 51
  ba ce 2a 45 2a fc 8a 7e
  4f 6b 2c 1f 74 f0 20 35
  d9 64 1a cd ba a4 66 e0
  00 ce d6 f2 d2 3b 63 1c
  8e 8a 0b 38 e2 ba e7 9a
  22 ca d8 1d 4c 50 d2 25
  35 8e bc 17 ac 0f 89 9e
  00 1c ee 8c 10 e2 59 80
  d8 c2 8d 2f d6 96 7d 1a
  b7 22 53 f2 10 98 b8 14
  a4 10 be 1f 59 98 de 03
  f5 8f 7e 7c db 7f 08 a6
  16 51 be 4d 0b 6f 8a 61
  df 15 30 44 0b d7 47 dc
  f0 d7 10 4f 6b 8d 24 c2
  ac 9b c1 3d 9c 6f e8 29
  05 25 d2 a6 d0 f8 84 42
  67 a1 57 0e 8e 29 4d c9
  3a 31 9f cf c0 3e a2 70
  17 d6 fd a3 47 b4 a7 94
  97 d7 f6 b1 42 2d 4e dd
  82 1c 19 93 4e 96 c1 aa
```

```
   87 76 57 25 d4 94 c7 64
   b1 55 dc 6d 13 26 91 74
```

(3) EDKEY zone with ASCII label and delegation record

```
Zone private key (d):
  5a f7 02 0e e1 91 60 32
  88 32 35 2b bc 6a 68 a8
  d7 1a 7c be 1b 92 99 69
  a7 c6 6d 41 5a 0d 8f 65

Zone identifier (ztype|zkey):
  00 01 00 14 3c f4 b9 24
  03 20 22 f0 dc 50 58 14
  53 b8 5d 93 b0 47 b6 3d
  44 6c 58 45 cb 48 44 5d
  db 96 68 8f

zTLD:
000G051WYJWJ80S04BRDRM2R2H9VGQCKP13VCFA4DHC4BJT88HEXQ5K8HW

Label:
  74 65 73 74 64 65 6c 65
  67 61 74 69 6f 6e

Number of records (integer): 1

Record #0 := (
  EXPIRATION: 8143584694000000 us
  00 1c ee 8c 10 e2 59 80

  DATA_SIZE:
  00 20

  TYPE:
  00 01 00 00

  FLAGS:   00 01

  DATA:
  21 e3 b3 0f f9 3b c6 d3
  5a c8 c6 e0 e1 3a fd ff
  79 4c b7 b4 4b bb c7 48
  d2 59 d0 a0 28 4d be 84

)

RDATA:
  00 1c ee 8c 10 e2 59 80
  00 20 00 01 00 01 00 00
  21 e3 b3 0f f9 3b c6 d3
  5a c8 c6 e0 e1 3a fd ff
  79 4c b7 b4 4b bb c7 48
  d2 59 d0 a0 28 4d be 84

Encryption NONCE|EXPIRATION:
  98 13 2e a8 68 59 d3 5c
  88 bf d3 17 fa 99 1b cb
  00 1c ee 8c 10 e2 59 80

Encryption key (K):
  85 c4 29 a9 56 7a a6 33
```

```
  41 1a 96 91 e9 09 4c 45
  28 16 72 be 58 60 34 aa
  e4 a2 a2 cc 71 61 59 e2

Storage key (q):
  ab aa ba c0 e1 24 94 59
  75 98 83 95 aa c0 24 1e
  55 59 c4 1c 40 74 e2 55
  7b 9f e6 d1 54 b6 14 fb
  cd d4 7f c7 f5 1d 78 6d
  c2 e0 b1 ec e7 60 37 c0
  a1 57 8c 38 4e c6 1d 44
  56 36 a9 4e 88 03 29 e9

ZKDF(zkey):
  9b f2 33 19 8c 6d 53 bb
  db ac 49 5c ab d9 10 49
  a6 84 af 3f 40 51 ba ca
  b0 dc f2 1c 8c f2 7a 1a

nonce := SHA-256 (dh[32..63] || h):
  14 f2 c0 6b ed c3 aa 2d
  f0 71 13 9c 50 39 34 f3
  4b fa 63 11 a8 52 f2 11
  f7 3a df 2e 07 61 ec 35

Derived private key (d', big-endian):
  3b 1b 29 d4 23 0b 10 a8
  ec 4d a3 c8 6e db 88 ea
  cd 54 08 5c 1d db 63 f7
  a9 d7 3f 7c cb 2f c3 98

BDATA:
  57 7c c6 c9 5a 14 e7 04
  09 f2 0b 01 67 e6 36 d0
  10 80 7c 4f 00 37 2d 69
  8c 82 6b d9 2b c2 2b d6
  bb 45 e5 27 7c 01 88 1d
  6a 43 60 68 e4 dd f1 c6
  b7 d1 41 6f af a6 69 7c
  25 ed d9 ea e9 91 67 c3

RRBLOCK:
  00 00 00 b0 00 01 00 14
  9b f2 33 19 8c 6d 53 bb
  db ac 49 5c ab d9 10 49
  a6 84 af 3f 40 51 ba ca
  b0 dc f2 1c 8c f2 7a 1a
  9f 56 a8 86 ea 73 9d 59
  17 50 8f 9b 75 56 39 f3
  a9 ac fa ed ed ca 7f bf
  a7 94 b1 92 e0 8b f9 ed
  4c 7e c8 59 4c 9f 7b 4e
  19 77 4f f8 38 ec 38 7a
  8f 34 23 da ac 44 9f 59
  db 4e 83 94 3f 90 72 00
  00 1c ee 8c 10 e2 59 80
  57 7c c6 c9 5a 14 e7 04
```

```
09 f2 0b 01 67 e6 36 d0
10 80 7c 4f 00 37 2d 69
8c 82 6b d9 2b c2 2b d6
bb 45 e5 27 7c 01 88 1d
6a 43 60 68 e4 dd f1 c6
b7 d1 41 6f af a6 69 7c
25 ed d9 ea e9 91 67 c3
```

(4) EDKEY zone with UTF-8 label and three records

```
Zone private key (d):
  5a f7 02 0e e1 91 60 32
  88 32 35 2b bc 6a 68 a8
  d7 1a 7c be 1b 92 99 69
  a7 c6 6d 41 5a 0d 8f 65

Zone identifier (ztype|zkey):
  00 01 00 14 3c f4 b9 24
  03 20 22 f0 dc 50 58 14
  53 b8 5d 93 b0 47 b6 3d
  44 6c 58 45 cb 48 44 5d
  db 96 68 8f

zTLD:
000G051WYJWJ80S04BRDRM2R2H9VGQCKP13VCFA4DHC4BJT88HEXQ5K8HW

Label:
  e5 a4 a9 e4 b8 8b e7 84
  a1 e6 95 b5

Number of records (integer): 3

Record #0 := (
  EXPIRATION: 8143584694000000 us
  00 1c ee 8c 10 e2 59 80

  DATA_SIZE:
  00 10

  TYPE:
  00 00 00 1c

  FLAGS:   00 00

  DATA:
  00 00 00 00 00 00 00 00
  00 00 00 00 de ad be ef

)

Record #1 := (
  EXPIRATION: 17999736901000000 us
  00 3f f2 aa 54 08 db 40

  DATA_SIZE:
  00 06

  TYPE:
  00 01 00 01

  FLAGS:   00 00

  DATA:
  e6 84 9b e7 a7 b0

)
```

```
Record #2 := (
  EXPIRATION: 11464693629000000 us
  00 28 bb 13 ff 37 19 40

  DATA_SIZE:
  00 0b

  TYPE:
  00 00 00 10

  FLAGS:   00 04

  DATA:
  48 65 6c 6c 6f 20 57 6f
  72 6c 64

)

RDATA:
  00 1c ee 8c 10 e2 59 80
  00 10 00 00 00 00 00 1c
  00 00 00 00 00 00 00 00
  00 00 00 00 de ad be ef
  00 3f f2 aa 54 08 db 40
  00 06 00 00 00 01 00 01
  e6 84 9b e7 a7 b0 00 28
  bb 13 ff 37 19 40 00 0b
  00 04 00 00 00 10 48 65
  6c 6c 6f 20 57 6f 72 6c
  64 00 00 00 00 00 00 00
  00 00 00 00 00 00 00 00
  00 00 00 00 00 00 00 00
  00 00 00 00 00 00 00 00
  00 00 00 00 00 00 00 00
  00 00 00 00 00 00 00 00

Encryption NONCE|EXPIRATION:
  bb 0d 3f 0f bd 22 42 77
  50 da 5d 69 12 16 e6 c9
  00 1c ee 8c 10 e2 59 80

Encryption key (K):
  3d f8 05 bd 66 87 aa 14
  20 96 28 c2 44 b1 11 91
  88 c3 92 56 37 a4 1e 5d
  76 49 6c 29 45 dc 37 7b

Storage key (q):
  ba f8 21 77 ee c0 81 e0
  74 a7 da 47 ff c6 48 77
  58 fb 0d f0 1a 6c 7f bb
  52 fc 8a 31 be f0 29 af
  74 aa 0d c1 5a b8 e2 fa
  7a 54 b4 f5 f6 37 f6 15
  8f a7 f0 3c 3f ce be 78
  d3 f9 d6 40 aa c0 d1 ed

ZKDF(zkey):
```

```
  74 f9 00 68 f1 67 69 53
  52 a8 a6 c2 eb 98 48 98
  c5 3a cc a0 98 04 70 c6
  c8 12 64 cb dd 78 ad 11

nonce := SHA-256 (dh[32..63] || h):
  f8 6a b5 33 8a 74 d7 a1
  d2 77 ea 11 ff 95 cb e8
  3a cf d3 97 3b b4 ab ca
  0a 1b 60 62 c3 7a b3 9c

Derived private key (d', big-endian):
  17 c0 68 a6 c3 f7 20 de
  0e 1b 69 ff 3f 53 e0 5d
  3f e5 c5 b0 51 25 7a 89
  a6 3c 1a d3 5a c4 35 58

BDATA:
  4e b3 5a 50 d4 0f e1 a4
  29 c7 f4 b2 67 a0 59 de
  4e 2c 8a 89 a5 ed 53 d3
  d4 92 58 59 d2 94 9f 7f
  30 d8 a2 0c aa 96 f8 81
  45 05 2d 1c da 04 12 49
  8f f2 5f f2 81 6e f0 ce
  61 fe 69 9b fa c7 2c 15
  dc 83 0e a9 b0 36 17 1c
  cf ca bb dd a8 de 3c 86
  ed e2 95 70 d0 17 4b 82
  82 09 48 a9 28 b7 f0 0e
  fb 40 1c 10 fe 80 bb bb
  02 76 33 1b f7 f5 1b 8d
  74 57 9c 14 14 f2 2d 50
  1a d2 5a e2 49 f5 bb f2
  a6 c3 72 59 d1 75 e4 40
  b2 94 39 c6 05 19 cb b1

RRBLOCK:
  00 00 01 00 00 01 00 14
  74 f9 00 68 f1 67 69 53
  52 a8 a6 c2 eb 98 48 98
  c5 3a cc a0 98 04 70 c6
  c8 12 64 cb dd 78 ad 11
  75 6d 2c 15 7a d2 ea 4f
  c0 b1 b9 1c 08 03 79 44
  61 d3 de f2 0d d1 63 6c
  fe dc 03 89 c5 49 d1 43
  6c c3 5b 4e 1b f8 89 5a
  64 6b d9 a6 f4 6b 83 48
  1d 9c 0e 91 d4 e1 be bb
  6a 83 52 6f b7 25 2a 06
  00 1c ee 8c 10 e2 59 80
  4e b3 5a 50 d4 0f e1 a4
  29 c7 f4 b2 67 a0 59 de
  4e 2c 8a 89 a5 ed 53 d3
  d4 92 58 59 d2 94 9f 7f
  30 d8 a2 0c aa 96 f8 81
  45 05 2d 1c da 04 12 49
```

```
8f f2 5f f2 81 6e f0 ce
61 fe 69 9b fa c7 2c 15
dc 83 0e a9 b0 36 17 1c
cf ca bb dd a8 de 3c 86
ed e2 95 70 d0 17 4b 82
82 09 48 a9 28 b7 f0 0e
fb 40 1c 10 fe 80 bb bb
02 76 33 1b f7 f5 1b 8d
74 57 9c 14 14 f2 2d 50
1a d2 5a e2 49 f5 bb f2
a6 c3 72 59 d1 75 e4 40
b2 94 39 c6 05 19 cb b1
```

D.3. Zone Revocation

The following is an example revocation for a PKEY zone:

```
Zone private key (d, big-endian):
  6f ea 32 c0 5a f5 8b fa
  97 95 53 d1 88 60 5f d5
  7d 8b f9 cc 26 3b 78 d5
  f7 47 8c 07 b9 98 ed 70

Zone identifier (ztype|zkey):
  00 01 00 00 2c a2 23 e8
  79 ec c4 bb de b5 da 17
  31 92 81 d6 3b 2e 3b 69
  55 f1 c3 77 5c 80 4a 98
  d5 f8 dd aa

Encoded zone identifier (zkl = zTLD):
000G001CM8HYGYFCRJXXXDET2WRS50EP7CQ3PTANY71QEQ409ACDBY6XN8

Difficulty (5 base difficulty + 2 epochs): 7

Signed message:
  00 00 00 34 00 00 00 03
  00 05 ff 1c 56 e4 b2 68
  00 01 00 00 2c a2 23 e8
  79 ec c4 bb de b5 da 17
  31 92 81 d6 3b 2e 3b 69
  55 f1 c3 77 5c 80 4a 98
  d5 f8 dd aa

Proof:
  00 05 ff 1c 56 e4 b2 68
  00 00 39 5d 18 27 c0 00
  38 0b 54 aa 70 16 ac a2
  38 0b 54 aa 70 16 ad 62
  38 0b 54 aa 70 16 af 3e
  38 0b 54 aa 70 16 af 93
  38 0b 54 aa 70 16 b0 bf
  38 0b 54 aa 70 16 b0 ee
  38 0b 54 aa 70 16 b1 c9
  38 0b 54 aa 70 16 b1 e5
  38 0b 54 aa 70 16 b2 78
  38 0b 54 aa 70 16 b2 b2
  38 0b 54 aa 70 16 b2 d6
  38 0b 54 aa 70 16 b2 e4
  38 0b 54 aa 70 16 b3 2c
  38 0b 54 aa 70 16 b3 5a
  38 0b 54 aa 70 16 b3 9d
  38 0b 54 aa 70 16 b3 c0
  38 0b 54 aa 70 16 b3 dd
  38 0b 54 aa 70 16 b3 f4
  38 0b 54 aa 70 16 b4 42
  38 0b 54 aa 70 16 b4 76
  38 0b 54 aa 70 16 b4 8c
  38 0b 54 aa 70 16 b4 a4
  38 0b 54 aa 70 16 b4 c9
  38 0b 54 aa 70 16 b4 f0
  38 0b 54 aa 70 16 b4 f7
  38 0b 54 aa 70 16 b5 79
  38 0b 54 aa 70 16 b6 34
```

```
38 0b 54 aa 70 16 b6 8e
38 0b 54 aa 70 16 b7 b4
38 0b 54 aa 70 16 b8 7e
38 0b 54 aa 70 16 b8 f8
38 0b 54 aa 70 16 b9 2a
00 01 00 00 2c a2 23 e8
79 ec c4 bb de b5 da 17
31 92 81 d6 3b 2e 3b 69
55 f1 c3 77 5c 80 4a 98
d5 f8 dd aa 08 ca ff de
3c 6d f1 45 f7 e0 79 81
15 37 b2 b0 42 2d 5e 1f
b2 01 97 81 ec a2 61 d1
f9 d8 ea 81 0a bc 2f 33
47 7f 04 e3 64 81 11 be
71 c2 48 82 1a d6 04 f4
94 e7 4d 0b f5 11 d2 c1
62 77 2e 81
```

The following is an example revocation for an EDKEY zone:

```
Zone private key (d):
  5a f7 02 0e e1 91 60 32
  88 32 35 2b bc 6a 68 a8
  d7 1a 7c be 1b 92 99 69
  a7 c6 6d 41 5a 0d 8f 65

Zone identifier (ztype|zkey):
  00 01 00 14 3c f4 b9 24
  03 20 22 f0 dc 50 58 14
  53 b8 5d 93 b0 47 b6 3d
  44 6c 58 45 cb 48 44 5d
  db 96 68 8f

Encoded zone identifier (zkl = zTLD):
000G051WYJWJ80S04BRDRM2R2H9VGQCKP13VCFA4DHC4BJT88HEXQ5K8HW

Difficulty (5 base difficulty + 2 epochs): 7

Signed message:
  00 00 00 34 00 00 00 03
  00 05 ff 1c 57 35 42 bd
  00 01 00 14 3c f4 b9 24
  03 20 22 f0 dc 50 58 14
  53 b8 5d 93 b0 47 b6 3d
  44 6c 58 45 cb 48 44 5d
  db 96 68 8f

Proof:
  00 05 ff 1c 57 35 42 bd
  00 00 39 5d 18 27 c0 00
  58 4c 93 3c b0 99 2a 08
  58 4c 93 3c b0 99 2d f7
  58 4c 93 3c b0 99 2e 21
  58 4c 93 3c b0 99 2e 2a
  58 4c 93 3c b0 99 2e 53
  58 4c 93 3c b0 99 2e 8e
  58 4c 93 3c b0 99 2f 13
  58 4c 93 3c b0 99 2f 2d
  58 4c 93 3c b0 99 2f 3c
  58 4c 93 3c b0 99 2f 41
  58 4c 93 3c b0 99 2f fd
  58 4c 93 3c b0 99 30 33
  58 4c 93 3c b0 99 30 82
  58 4c 93 3c b0 99 30 a2
  58 4c 93 3c b0 99 30 e1
  58 4c 93 3c b0 99 31 ce
  58 4c 93 3c b0 99 31 de
  58 4c 93 3c b0 99 32 12
  58 4c 93 3c b0 99 32 4e
  58 4c 93 3c b0 99 32 9f
  58 4c 93 3c b0 99 33 31
  58 4c 93 3c b0 99 33 87
  58 4c 93 3c b0 99 33 8c
  58 4c 93 3c b0 99 33 e5
  58 4c 93 3c b0 99 33 f3
  58 4c 93 3c b0 99 34 26
  58 4c 93 3c b0 99 34 30
```

```
58 4c 93 3c b0 99 34 68
58 4c 93 3c b0 99 34 88
58 4c 93 3c b0 99 34 8a
58 4c 93 3c b0 99 35 4c
58 4c 93 3c b0 99 35 bd
00 01 00 14 3c f4 b9 24
03 20 22 f0 dc 50 58 14
53 b8 5d 93 b0 47 b6 3d
44 6c 58 45 cb 48 44 5d
db 96 68 8f 04 ae 26 f7
63 56 5a b7 aa ab 01 71
72 4f 3c a8 bc c5 1a 98
b7 d4 c9 2e a3 3c d9 34
4c a8 b6 3e 04 53 3a bf
1a 3c 05 49 16 b3 68 2c
5c a8 cb 4d d0 f8 4c 3b
77 48 7a ac 6e ce 38 48
0b a9 d5 00
```

Acknowledgements

The authors thank all reviewers for their comments. In particular, we thank D. J. Bernstein, S. Bortzmeyer, A. Farrel, E. Lear, and R. Salz for their insightful and detailed technical reviews. We thank J. Yao and J. Klensin for the internationalization reviews. We thank NLnet and NGI DISCOVERY for funding work on the GNU Name System.

Authors' Addresses

Martin Schanzenbach
Fraunhofer AISEC
Lichtenbergstrasse 11
85748 Garching
Germany
Email: martin.schanzenbach@aisec.fraunhofer.de

Christian Grothoff
Berner Fachhochschule
Hoeheweg 80
CH-2501 Biel/Bienne
Switzerland
Email: christian.grothoff@bfh.ch

Bernd Fix
GNUnet e.V.
Boltzmannstrasse 3
85748 Garching
Germany
Email: fix@gnunet.org