Public Key Infrastructure

Yawning Angel
Claudia Diaz
Ania Piotrowska
David Stainton
Masala

Table of Contents

Abstract

1. Introduction

Mixnets are designed with the assumption that a Public Key Infrastructure (PKI) exists and it gives each client the same view of the network. This specification is inspired by the Tor and Mixminion Directory Authority systems MIXMINIONDIRAUTH TORDIRAUTH whose main features are precisely what we need for our PKI. These are decentralized systems meant to be collectively operated by multiple entities.

The mix network directory authority system (PKI) is essentially a cooperative decentralized database and voting system that is used to produce network consensus documents which mix clients periodically retrieve and use for their path selection algorithm when creating Sphinx packets. These network consensus documents are derived from a voting process between the Directory Authority servers.

This design prevents mix clients from using only a partial view of the network for their path selection so as to avoid fingerprinting and bridging attacks FINGERPRINTING, BRIDGING, and LOCALVIEW.

The PKI is also used by Authority operators to specify network-wide parameters, for example in the Katzenpost Decryption Mix Network KATZMIXNET the Poisson mix strategy is used and, therefore, all clients must use the same lambda parameter for their exponential distribution function when choosing hop delays in the path selection. The Mix Network Directory Authority system, aka PKI, SHALL be used to distribute such network-wide parameters in the network consensus document that have an impact on security and performance.

1.1 Conventions Used in This Document

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in RFC2119.

The “C” style Presentation Language as described in RFC5246 Section 4 is used to represent data structures for additional cryptographic wire protocol commands. KATZMIXWIRE

1.2 Terminology

`PKI` - Public Key Infrastructure

`Directory Authority system` - refers to specific PKI schemes used by

Mixminion and Tor

`MSL` - maximum segment lifetime

`mix descriptor` - A database record which describes a component mix

`family` - Identifier of security domains or entities operating one or more mixes in the network. This is used to inform the path selection algorithm.

`nickname` - simply a nickname string that is unique in the consensus document, see “Katzenpost Mix Network Specification” section “2.2. Network Topology”.

`layer` - The layer indicates which network topology layer a particular mix resides in.

`Provider` - A service operated by a third party that Clients communicate directly with to communicate with the Mixnet. It is responsible for Client authentication, forwarding outgoing messages to the Mixnet, and storing incoming messages for the Client. The Provider MUST have the ability to perform cryptographic operations on the relayed messages.

1.3 Security Properties Overview

This Directory Authority system has the following feature goals and security properties:

- All Directory Authority servers must agree with each other on the set of Directory Authorities.
- All Directory Authority servers must agree with each other on the set of mixes.
- This system is intentionally designed to provide identical network consensus documents to each mix client. This mitigates epistemic attacks against the client path selection algorithm such as fingerprinting and bridge attacks FINGERPRINTING BRIDGING.
- This system is NOT byzantine-fault-tolerant, it instead allows for manual intervention upon consensus fault by the Directory Authority operators. Further, these operators are responsible for expelling bad acting operators from the system.
- This system enforces the network policies such as mix join policy wherein intentionally closed mixnets will prevent arbitrary hosts from joining the network by authenticating all descriptor signatures with a list of allowed public keys.
- The Directory Authority system for a given mix network is essentially the root of all authority.

1.4 Differences from Tor and Mixminion Directory Authority systems

In this document we specify a Directory Authority system which is different from that of Tor's and Mixminion's in a number of ways:

- The list of valid mixes is expressed in an allowlist. For the time being there is no specified "bandwidth authority" system which verifies the health of mixes (Further research required in this area).
- There's no non-directory channel to inform clients that a node is down, so it will end up being a lot of packet loss, since clients will continue to include the missing node in their path selection until keys published by the node expire and it falls out of the consensus.
- The schema of the mix descriptors is different from that used in Mixminion and Tor, including a change which allows our mix descriptor to express *n* Sphinx mix routing public keys in a single mix descriptor whereas in the Tor and Mixminion Directory Authority systems, *n* descriptors are used.
- The serialization format of mix descriptors is different from that used in Mixminion and Tor.
- The shared random number computation is performed every voting round, and is required for a vote to be accepted by each authority. The shared random number is used to deterministically generate the network topology.

2. Overview of Mix PKI Interaction

Each Mix MUST rotate the key pair used for Sphinx packet processing periodically for forward secrecy reasons and to keep the list of seen packet tags short. SPHINX09 SPHINXSPEC The Katzenpost Mix Network uses a fixed interval (`epoch`), so that key rotations happen simultaneously throughout the network, at predictable times.

Each Directory Authority server MUST use some time synchronization protocol in order to correctly use this protocol. This Directory Authority system requires time synchronization to within a few minutes.

Let each epoch be exactly `1200 seconds (20 minutes)` in duration, and the 0th Epoch begin at `2017-06-01 00:00 UTC`.

To facilitate smooth operation of the network and to allow for delays that span across epoch boundaries, Mixes MUST publish keys to the PKI for at least 3 epochs in advance, unless the mix will be otherwise unavailable in the near future due to planned downtime.

At an epoch boundary, messages encrypted to keys from the previous epoch are accepted for a grace period of 2 minutes.

Thus, at any time, keys for all Mixes for the Nth through N + 2nd epoch will be available, allowing for a maximum round trip (forward message + SURB) delay + transit time of 40 minutes. SURB lifetime is limited to a single epoch because of the key rotation epoch, however this shouldn't present any useability problems since SURBs are only used for sending ACK messages from the destination Provider to the sender as described in KATZMIXE2E.

2.1 PKI Protocol Schedule

There are two main constraints to Authority schedule:

1. There MUST be enough key material extending into the future so that clients are able to construct Sphinx packets with a forward and reply paths.

2. All participants should have enough time to participate in the protocol; upload descriptors, vote, generate documents, download documents, establish connections for user traffic.

The epoch duration of 20 minutes is more than adequate for these two constraints.

NOTE: perhaps we should make it shorter? but first let's do some scaling and bandwidth calculations to see how bad it gets…

2.1.1 Directory Authority Server Schedule

Directory Authority server interactions are conducted according to the following schedule, where `T` is the beginning of the current epoch, and `P` is the length of the epoch period.

- `T` - Epoch begins
- `T + P/2` - Vote exchange
- `T + (5/8)*P` - Reveal exchange
- `T + (6/8)*P` - Tabulation and signature exchange
- `T + (7/8)*P` - Publish consensus

2.1.2 Mix Schedule

Mix PKI interactions are conducted according to the following schedule, where T is the beginning of the current epoch.

`T + P/8` - Deadline for publication of all mixes documents for the next epoch.

`T + (7/8)*P` - This marks the beginning of the period where mixes perform staggered fetches of the PKI consensus document.

`T + (8/9)*P` - Start establishing connections to the new set of relevant mixes in advance of the next epoch.

`T + P - 1MSL` - Start accepting new Sphinx packets encrypted to the next epoch's keys.

`T + P + 1MSL` - Stop accepting new Sphinx packets encrypted to the previous epoch's keys, close connections to peers no longer listed in the PKI documents and erase the list of seen packet tags.

Mix layer changes are controlled by the Directory Authorities and therefore a mix can be reassigned to a different layer in our stratified topology at any new epoch. Mixes will maintain incoming and outgoing

connections to the various nodes until all mix keys have expired, iff the node is still listed anywhere in the current document.

3. Voting for Consensus Protocol

In our Directory Authority protocol, all the actors conduct their behavior according to a common schedule as outlined in section "2.1 PKI Protocol Schedule". The Directory Authority servers exchange messages to reach consensus about the network. Other tasks they perform include collecting mix descriptor uploads from each mix for each key rotation epoch, voting, shared random number generation, signature exchange and publishing of the network consensus documents.

3.1 Protocol Messages

There are only two document types in this protocol:

- `mix_descriptor`: A mix descriptor describes a mix.
- `directory`: A directory contains a list of descriptors and other information that describe the mix network.

Mix descriptor and directory documents MUST be properly signed.

3.1.1 Mix Descriptor and Directory Signing

Mixes MUST compose mix descriptors which are signed using their private identity key, an ed25519 key. Directories are signed by one or more Directory Authority servers using their authority key, also an ed25519 key. In all cases, signing is done using JWS RFC7515.

3.2 Vote Exchange

As described in section “2.1 PKI Protocol Schedule”, the Directory Authority servers begin the voting process 1/8 of an epoch period after the start of a new epoch. Each Authority exchanges vote directory messages with each other.

Authorities archive votes from other authorities and make them available for retreival. Upon receiving a new vote, the authority examines it for new descriptors and includes any valid descriptors in its view of the network.

Each Authority includes in its vote a hashed value committing to a choice of a random number for the vote. See section 4.3 for more details.

3.2.1 Voting Wire Protocol Commands

The Katzenpost Wire Protocol as described in `KATZMIXWIRE` is used by Authorities to exchange votes. We define additional wire protocol commands for sending votes:

```
enum {

:   vote(22), vote_status(23),

} Command;
```

The structures of these commands are defined as follows:

```
struct {
:   uint64_t epoch_number; opaque public_key[ED25519_KEY_LENGTH];
    opaque payload[];

} VoteCommand;

struct {
:   uint8 error_code;

} VoteStatusCommand;
```

3.2.2 The vote Command

The vote command is used to send a PKI document to a peer Authority during the voting period of the PKI schedule.

The payload field contains the signed and serialized PKI document representing the sending Authority's vote. The public_key field contains the public identity key of the sending Authority which the receiving Authority can use to verify the signature of the payload. The epoch_number field is used by the receiving party to quickly check the epoch for the vote before deserializing the payload.

Each authority MUST include its commit value for the shared random computation in this phase along with its signed vote. This computation is derived from the Tor Shared Random Subsystem, TORSRV.

3.2.3 The vote_status Command

The vote_status command is used to reply to a vote command. The error_code field indicates if there was a failure in the receiving of the PKI document.

```
enum {

:   vote_ok(0), /\* None error condition. */ vote_too_early(1), /*
    The Authority should try again later. */ vote_too_late(2), /*
    This round of voting was missed. \*/
}
```

The epoch_number field of the vote struct is compared with the epoch that is currently being voted on. vote_too_early and vote_too_late are replied back to the voter to report that their vote was not accepted.

3.3 Reveal Exchange

As described in section "2.1 PKI Protocol Schedule", the Directory Authority servers exchange the reveal values after they have exchanged votes which contain a commit value. Each Authority exchanges reveal messages with each other.

3.3.1 Reveal Wire Protocol Commands

The Katzenpost Wire Protocol as described in KATZMIXWIRE is used by Authorities to exchange reveal values previously commited to in their votes. We define additional wire protocol commands for exchanging reveals:

```
enum {
:   reveal(25), reveal_status(26),
```

```
} Command;
```

The structures of these commands are defined as follows:

```
struct {
:   uint64_t epoch_number; opaque public_key[ED25519_KEY_LENGTH];
    opaque payload[];

} RevealCommand;

struct {
:   uint8 error_code;

} RevealStatusCommand;
```

3.3.2 The reveal Command

The reveal command is used to send a reveal value to a peer authority during the reveal period of the PKI schedule.

The payload field contains the signed and serialized reveal value. The public_key field contains the public identity key of the sending Authority which the receiving Authority can use to verify the signature of the payload. The epoch_number field is used by the receiving party to quickly check the epoch for the reveal before deserializing the payload.

3.3.3 The reveal_status Command

The reveal_status command is used to reply to a reveal command. The error_code field indicates if there was a failure in the receiving of the shared random reveal value.

```
enum {

:   reveal_ok(8), /* None error condition. */ reveal_too_early(9),
    /* The Authority should try again later. */
    reveal_not_authorized(10), /* The Authority was rejected. */
    reveal_already_received(11), /* The Authority has already revealed
    this round. */ reveal_too_late(12) /* This round of revealing was
    missed. */

} Errorcodes;
```

The epoch_number field of the reveal struct is compared with the epoch that is currently being voted on. reveal_too_early and reveal_too_late are replied back to the authority to report their reveal was not accepted. The status code reveal_not_authorized is used if the Authority is rejected. The reveal_already_received is used to report that a valid reveal command was already received for this round.

3.4 Cert Exchange

The Cert command is the same as a Vote but contains the set of Reveal values as seen by the voting peer. In order to ensure that a misconfigured or malicious Authority operator cannot amplify their ability to influence the threshold voting process, after Reveal messages have been exchanged, Authorities vote again, including the Reveals seen by them. Authorities may not introduce new MixDescriptors at this phase in the protocol.

Otherwise, a consensus partition can be obtained by witholding Reveal values from a threshold number of Peers. In the case of an even-number of Authorities, a denial of service by a single Authority was observed.

3.5 Vote Tabulation for Consensus Computation

The main design constraint of the vote tabulation algorithm is that it MUST be a deterministic process that produces the same result for each directory authority server. This result is known as a network consensus file.

A network consensus file is a well formed directory struct where the `status` field is set to `consensus` and contains 0 or more descriptors, the mix directory is signed by 0 or more directory authority servers. If signed by the full voting group then this is called a fully signed consensus.

1. Validate each vote directory:

- that the liveness fields correspond to the following epoch
- status is `vote`
- version number matches ours

2. Compute a consensus directory:

Here we include a modified section from the Mixminion PKI spec MIXMINIONDIRAUTH:

For each distinct mix identity in any vote directory:

- If there are multiple nicknames for a given identity, do not include any descriptors for that identity.
- If half or fewer of the votes include the identity, do not include any descriptors for the identity. *This also guarantees that there will be only one identity per nickname.*
- If we are including the identity, then for each distinct descriptor that appears in any vote directory:
  - Do not include the descriptor if it will have expired on the date the directory will be published.
  - Do not include the descriptor if it is superseded by other descriptors for this identity.
  - Do not include the descriptor if it not valid in the next epoch.
  - Otherwise, include the descriptor.
- Sort the list of descriptors by the signature field so that creation of the consensus is reproducible.
- Set directory `status` field to `consensus`.

3. Compute a shared random number from the values revealed in the “Reveal” step. Authorities whose reveal value does not verify their commit value MUST be excluded from the consensus round. Authorities ensure that their peers MUST participate in Commit-and-Reveal, and MUST use correct Reveal values obtained from other Peers as part of the “Cert” exchange.

4. Generate or update the network topology using the shared random number as a seed to a deterministic random number generator that determines the order that new mixes are placed into the topology.

3.6 Signature Collection

Each Authority signs their view of consensus, and exchanges detached Signatures with each other. Upon receiving each Signature it is added to the signatures on the Consensus if it validates the Consensus. The Authority SHOULD warn the administrator if network partition is detected.

If there is disagreement about the consensus directory, each authority collects signatures from only the servers which it agrees with about the final consensus.

// TODO: consider exchanging peers votes amongst authorities (or hashes thereof) to // ensure that an authority has distributed one and only unique vote amongst its peers.

3.7 Publication

If the consensus is signed by a majority of members of the voting group then it's a valid consensus and it is published.

4. PKI Protocol Data Structures

4.1 Mix Descriptor Format

Note that there is no signature field. This is because mix descriptors are serialized and signed using JWS. The `IdentityKey` field is a public ed25519 key. The `MixKeys` field is a map from epoch to public X25519 keys which is what the Sphinx packet format uses.

Note

XXX David: replace the following example with a JWS example:

```
{
    "Version": 0,
    "Name": "",
    "Family": "",
    "Email": "",
    "AltContactInfo":"",
    "IdentityKey": "",
    "LinkKey":"",
    "MixKeys": {
       "Epoch": "EpochPubKey",
    },
    "Addresses": ["IP:Port"],
    "Layer": 0,
    "LoadWeight":0,
    "AuthenticationType":""
}
```

4.1.1 Scheduling Mix Downtime

Mix operators can publish a half empty mix descriptor for future epochs to schedule downtime. The mix descriptor fields that MUST be populated are:

- Version
- Name
- Family
- Email
- Layer

- IdentityKey
- MixKeys

The map in the field called "MixKeys" should reflect the scheduled downtime for one or more epochs by not have those epochs as keys in the map.

4.2 Directory Format

Note: replace the following example with a JWS example

```
{
    "Signatures": [],
    "Version": 0,
    "Status": "vote",
    "Lambda" : 0.274,
    "MaxDelay" : 30,
    "Topology" : [],
    "Providers" : [],
}
```

4.3 Shared Random Value structure

Katzenpost's Shared Random Value computation is inspired by Tor's Shared Random Subsystem TORSRV.

Each voting round a commit value is included in the votes sent to other authorities. These are produced as follows:

```
H = blake2b-256

COMMIT = Uint64(epoch) | H(REVEAL) REVEAL = Uint64(epoch) | H(RN)
```

After the votes are collected from the voting round, and before signature exchange, the Shared Random Value field of the consensus document is the output of H over the input string calculated as follows:

1. Validated Reveal commands received including the authorities own reveal are sorted by reveal value in ascending order and appended to the input in format IdentityPublicKeyBytes_n | RevealValue_n

However instead of the Identity Public Key bytes we instead encode the Reveal with the blake2b 256 bit hash of the public key bytes.

2. If a SharedRandomValue for the previous epoch exists, it is appended to the input string, otherwise 32 NUL (x00) bytes are used.

```
REVEALS = ID_a \| R_a \| ID_b \| R_b \| \... SharedRandomValue =
H("shared-random" | Uint64(epoch) | REVEALS | PREVIOUS_SRV)
```

5. PKI Wire Protocol

The Katzenpost Wire Protocol as described in KATZMIXWIRE is used by both clients and by Directory Authority peers. In the following section we describe additional wire protocol commands for publishing mix descriptors, voting and consensus retrieval.

5.1 Mix Descriptor publication

The following commands are used for publishing mix descriptors and setting mix descriptor status:

```
enum {
      /* Extending the wire protocol Commands. */
      post_descriptor(20),
      post_descriptor_status(21),
}
```

The structures of these command are defined as follows:

```
struct {
   uint64_t epoch_number;
   opaque payload[];
} PostDescriptor;

struct {
   uint8 error_code;
} PostDescriptorStatus;
```

5.1.1 The post_descriptor Command

The post_descriptor command allows mixes to publish their descriptors.

5.1.2 The post_descriptor_status Command

The post_descriptor_status command is sent in response to a post_descriptor command, and uses the following error codes:

```
enum {
   descriptor_ok(0),
   descriptor_invalid(1),
   descriptor_conflict(2),
   descriptor_forbidden(3),
} ErrorCodes;
```

5.2 Voting

The following commands are used by Authorities to exchange votes:

```
enum {
   vote(22),
   vote_status(23),
   get_vote(24),
} Command;
```

The structures of these commands are defined as follows:

```
struct {
```

```
    uint64_t epoch_number;
    opaque public_key[ED25519_KEY_LENGTH];
    opaque payload[];
} VoteCommand;

struct {
   uint8 error_code;
} VoteStatusCommand;
```

5.2.1 The vote Command

The `vote` command is used to send a PKI document to a peer Authority during the voting period of the PKI schedule.

The payload field contains the signed and serialized PKI document representing the sending Authority's vote. The public_key field contains the public identity key of the sending Authority which the receiving Authority can use to verify the signature of the payload. The epoch_number field is used by the receiving party to quickly check the epoch for the vote before deserializing the payload.

5.2.2 The vote_status Command

The `vote_status` command is used to reply to a vote command. The error_code field indicates if there was a failure in the receiving of the PKI document.

```
enum {
   vote_ok(0),               /* None error condition. */
   vote_too_early(1),        /* The Authority should try again later. */
   vote_too_late(2),         /* This round of voting was missed. */
   vote_not_authorized(3),   /* The voter's key is not authorized. */
   vote_not_signed(4),       /* The vote signature verification failed */
   vote_malformed(5),        /* The vote payload was invalid */
   vote_already_received(6), /* The vote was already received */
   vote_not_found(7),        /* The vote was not found */
}
```

The epoch_number field of the vote struct is compared with the epoch that is currently being voted on. vote_too_early and vote_too_late are replied back to the voter to report that their vote was not accepted.

5.2.3 The get_vote Command

The `get_vote` command is used to request a PKI document (vote) from a peer Authority. The epoch field contains the epoch from which to request the vote, and the public_key field contains the public identity key of the Authority of the requested vote. A successful query is responded to with a vote command, and queries that fail are responded to with a vote_status command with error_code vote_not_found(7).

5.3 Retrieval of Consensus

Providers in the Katzenpost mix network system KATZMIXNET may cache validated network consensus files and serve them to clients over the mix network's link layer wire protocol KATZMIXWIRE. We define additional wire protocol commands for requesting and sending PKI consensus documents:

```
enum {
   /* Extending the wire protocol Commands. */
```

```
   get_consensus(18),
   consensus(19),
} Command;

The structures of these commands are defined as follows:


struct {
    uint64_t epoch_number;
} GetConsensusCommand;

struct {
   uint8 error_code;
   opaque payload[];
} ConsensusCommand;
```

5.3.1 The get_consensus Command

The get_consensus command is a command that is used to retrieve a recent consensus document. If a given get_consensus command contains an Epoch value that is either too big or too small then a reply consensus command is sent with an empty payload. Otherwise if the consensus request is valid then a consensus command containing a recent consensus document is sent in reply.

Initiators MUST terminate the session immediately upon reception of a get_consensus command.

5.3.2 The consensus Command

The consensus command is a command that is used to send a recent consensus document. The error_code field indicates if there was a failure in retrieval of the PKI consensus document.

```
enum {
   consensus_ok(0),        /* None error condition and SHOULD be accompanied with
                              a valid consensus payload. */
   consensus_not_found(1), /* The client should try again later. */
   consensus_gone(2),      /* The consensus will not be available in the future. *
} ErrorCodes;
```

5.4.1 The Cert Command

The `cert` command is used to send a PKI document to a peer Authority during the voting period of the PKI schedule. It is the same as the `vote` command, but must contain the set of SharedRandomCommit and SharedRandomReveal values as seen by the Authority during the voting process.

5.4.2 The CertStatus Command

The `cert_status` command is the response to a `cert` command, and is the same as a `vote_status` response, other than the command identifier. Responses are CertOK, CertTooEarly, CertNotAuthorized, CertNotSigned, CertAlreadyReceived, CertTooLate

5.5 Signature Exchange

Signatures exchange is the final round of the consensus protocol and consists of the Sig and SigStatus commands.

5.5.1 The Sig Command

The `sig` command contains a detached Signature from PublicKey of Consensus for Epoch.

5.5.2 The SigStatus Command

The `sig_status` command is the response to a `sig` command. Responses are SigOK, SigNotAuthorized, SigNotSigned, SigTooEarly, SigTooLate, SigAlreadyReceived, and SigInvalid.

6. Scalability Considerations

TODO: notes on scaling, bandwidth usage etc.

7. Future Work

- byzantine fault tolerance
- PQ crypto signatures for all PKI documents: mix descriptors and directories. SPHINCS256 could be used, we already have a golang implementation: https://github.com/Yawning/sphincs256/
- Make a Bandwidth Authority system to measure health of the network. Also perform load balancing as described in PEERFLOW?
- Implement byzantine attack defenses as described in MIRANDA and MIXRELIABLE where mix link performance proofs are recorded and used in a reputation system.
- Choose a different serialization/schema language?
- Use a append only merkle tree instead of this voting protocol.

8. Anonymity Considerations

- This system is intentionally designed to provide identical network consensus documents to each mix client. This mitigates epistemic attacks against the client path selection algorithm such as fingerprinting and bridge attacks FINGERPRINTING, BRIDGING.
- If consensus has failed and thus there is more than one consensus file, clients MUST NOT use this compromised consensus and refuse to run.
- We try to avoid randomizing the topology because doing so splits the anonymity sets on each mix into two. That is, packets belonging to the previous topology versus the current topology are trivially distinguishable. On the other hand if enough mixes fall out of consensus eventually the mixnet will need to be rebalanced to avoid an attacker compromised path selection. One example of this would be the case where the adversary controls the only mix is one layer of the network topology.

9. Security Considerations

- The Directory Authority / PKI system for a given mix network is essentially the root of all authority in the system. The PKI controls the contents of the network consensus documents that mix clients download and use to inform their path selection. Therefore if the PKI as a whole becomes compromised then so will the rest of the system in terms of providing the main security properties described as traffic analysis resistance. Therefore a decentralized voting protocol is used so that the system is more resiliant when attacked, in accordance with the principle of least authority. SECNOTSEP
- Short epoch durations make it is more practical to make corrections to network state using the PKI voting rounds.
- Fewer epoch keys published in advance is a more conservative security policy because it implies reduced exposure to key compromise attacks.
- A bad acting Directory Authority who lies on each vote and votes inconsistently can trivially cause a denial of service for each voting round.

10. Acknowledgements

We would like to thank Nick Mathewson for answering design questions and thorough design review.

Appendix A. References

Appendix A.1 Normative References

Appendix A.2 Informative References

Appendix B. Citing This Document

Appendix B.1 Bibtex Entry

Note that the following bibtex entry is in the IEEEtran bibtex style as described in a document called “How to Use the IEEEtran BIBTEX Style”.

```
@online{KatzMixPKI,
title = {Katzenpost Mix Network Public Key Infrastructure Specification},
author = {Yawning Angel and Ania Piotrowska and David Stainton},
url= {https://github.com/katzenpost/katzenpost/blob/main/docs/specs/pki.rst},
year = {2017}
}
```

BRIDGING

Danezis, G., Syverson, P., “Bridging and Fingerprinting: Epistemic Attacks on Route Selection”, In the Proceedings of PETS 2008, Leuven, Belgium, July 2008, https://www.freehaven.net/anonbib/cache/danezis-pet2008.pdf

FINGERPRINTING

Danezis, G., Clayton, R., “Route Finger printing in Anonymous Communications”, https://www.cl.cam.ac.uk/~rnc1/anonroute.pdf

KATZMIXE2E

Angel, Y., Danezis, G., Diaz, C., Piotrowska, A., Stainton, D., “Katzenpost Mix Network End-to-end Protocol Specification”, July 2017, https://github.com/katzenpost/katzenpost/blob/main/docs/specs/old/end_to_end.md

KATZMIXNET

Angel, Y., Danezis, G., Diaz, C., Piotrowska, A., Stainton, D., “Katzenpost Mix Network Specification”, June 2017, https://github.com/katzenpost/katzenpost/blob/main/docs/specs/mixnet.md

KATZMIXWIRE

Angel, Y. “Katzenpost Mix Network Wire Protocol Specification”, June 2017, https://github.com/katzenpost/katzenpost/blob/main/docs/specs/wire-protocol.md

LOCALVIEW

Gogolewski, M., Klonowski, M., Kutylowsky, M., “Local View Attack on Anonymous Communication”, https://www.freehaven.net/anonbib/cache/esorics05-Klonowski.pdf

MIRANDA

Leibowitz, H., Piotrowska, A., Danezis, G., Herzberg, A., 2017, “No right to ramain silent: Isolating Malicious Mixes” https://eprint.iacr.org/2017/1000.pdf

MIXMINIONDIRAUTH

Danezis, G., Dingledine, R., Mathewson, N., “Type III (Mixminion) Mix Directory Specification”, December 2005, https://www.mixminion.net/dir-spec.txt

MIXRELIABLE

Dingledine, R., Freedman, M., Hopwood, D., Molnar, D., 2001 “A Reputation System to Increase MIX-Net Reliability”, In Information Hiding, 4th International Workshop https://www.freehaven.net/anonbib/cache/mix-acc.pdf

PEERFLOW

Johnson, A., Jansen, R., Segal, A., Syverson, P., “PeerFlow: Secure Load Balancing in Tor”, Preceedings on Privacy Enhancing Technologies, July 2017, https://petsymposium.org/2017/papers/issue2/paper12-2017-2-source.pdf

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

RFC5246

Dierks, T. and E. Rescorla, “The Transport Layer Security (TLS) Protocol Version 1.2”, RFC 5246, DOI 10.17487/RFC5246, August 2008, http://www.rfc-editor.org/info/rfc5246

RFC7515

Jones, M., Bradley, J., Sakimura, N., “JSON Web Signature (JWS)”, May 2015, https://tools.ietf.org/html/rfc7515

SECNOTSEP

Miller, M., Tulloh, B., Shapiro, J., “The Structure of Authority: Why Security Is not a Separable Concern”, http://www.erights.org/talks/no-sep/secnotsep.pdf

SPHINCS256

Bernstein, D., Hopwood, D., Hulsing, A., Lange, T., Niederhagen, R., Papachristodoulou, L., Schwabe, P., Wilcox O’ Hearn, Z., “SPHINCS: practical stateless hash-based signatures”, http://sphincs.cr.yp.to/sphincs-20141001.pdf

SPHINX09

Danezis, G., Goldberg, I., “Sphinx: A Compact and Provably Secure Mix Format”, DOI 10.1109/SP.2009.15, May 2009, http://research.microsoft.com/en-us/um/people/gdane/papers/sphinx-eprint.pdf

SPHINXSPEC

Angel, Y., Danezis, G., Diaz, C., Piotrowska, A., Stainton, D., “Sphinx Mix Network Cryptographic Packet Format Specification” July 2017, https://github.com/katzenpost/katzenpost/blob/main/docs/specs/sphinx.md

TORDIRAUTH

“Tor directory protocol, version 3”, https://gitweb.torproject.org/torspec.git/tree/dir-spec.txt

TORSRV

“Tor Shared Random Subsystem Specification”, https://gitweb.torproject.org/torspec.git/tree/srv-spec.txt