0.1 Intro

This document is a description of the protocol described by Nicholas Hopper here: `https://lists.torproject.org/pipermail/tor-dev/2014-January/006053.html`. The protocol described here contains modifications to the original (these are listed in Section 0.4).

The purpose of the protocol is to periodically generate a shared random value among a set of approximately 5 to 20 Tor directory authorities. That value can then be used as a nonce by parts of the Tor protocol. No participant or group of participants smaller than the DKG (Distributed Key Generation) threshold[1] should be able to influence the value of the resulting nonce. We will denote this nonce as RAND.

To describe it in simplest terms this protocol generates an unpredictable value by using a distributed private key to generate a threshold signature of the time. Each directory authority has a single share of the distributed private key; none of them know anything else about the private key and nobody knows its full value. Each share is used to create a share of the threshold signature; these signature shares are then published. A non-interactive zero-knowledge proof is also published to prove validity of each signature share. When enough signature shares are published they can be interpolated to construct the full signature; this signature is then used as the shared random.

We make use of two sub-protocols: A DKG (Distributed Key Generation) protocol, and a consensus protocol that we call the *list creation protocol* (Section 0.3.3). Both of these sub-protocols exist to solve well known problems with pre-existing solutions.

For the sake of simplicity and modularity the internals of these sub-protocols are not defined; instead we give a generalised interface and set of requirements so that implementers can make use of whatever existing tools and frameworks they desire.

Good

- Each nonce has a single possible value that is deterministic, yet unpredictable, and can't be influenced by the participants. There may be a fall-back value, however, that is used when the primary value can't be calculated for whatever reason.
- The protocol is quite simple and efficient (ignoring the DKG sub-protocol).

[1] Defined in the email as ceiling(n / 2).

- Only a single share is published by each participant per run (not including the once-off DKG).

Bad

- It requires a setup phase, but this only needs to be performed once every x nonce generations, for some configured value x.
- (If you don't use pre-existing parts for the sub-protocols) It has a large implementation complexity. This is because it makes use of two separate sub-protocols. Of course if there's a pre-existing DKG and consensus protocol at hand then it's actually very simple.
- Only secure in the random oracle model. What this means is that the choice of hash function in the zero knowledge proof is important. If a weak hash function is used then the protocol is weak; if a strong hash function is used then the protocol is strong.
- Not resistant to changes in the network. The threshold for the shared random calculation is the same as the threshold used in the DKG protocol. Luckily directory authorities are relatively stable so this shouldn't be much of an issue[2].

0.2 Notation

$\mathbb{F}_p$ The prime order finite field containing all the integer values we use. Simply put $\mathbb{F}_p = \{0, 1, \ldots, p-1\}$. Whenever we refer to "integers" we actually mean elements of this field. Operations done between these field elements are slightly different from regular algebraic operations, most notably everything is mod p. For instance, $a + b$ actually means $a + b \bmod p$. It is important to note that this same field must be used by the DKG protocol; if it isn't then strange things may happen.

$a + b$ The addition of integer a and integer b over $\mathbb{F}_p$.

$a * b$ The multiplication of integer a and integer b over $\mathbb{F}_p$.

$\frac{a}{b}$ The multiplication of integer a by the modular multiplicative inverse of integer b modulo p. Since p is a prime we know by Euler's theorem that the multiplicative inverse of b modulo p is $= b^{p-2} \bmod p$.

$A + B$ The addition of group member A and group member B, where "addition" refers to the operation of the abelian group A and B are both from.

[2] In fact if over half of the directory authorities lost their sanity or simply vanished then the Tor network would probably experience far bigger problems than a bad nonce.

$a \cdot B$ The multiplication of group member B by an integer a. B added to itself a times.

$a||b$ The concatenation of a and b.

$|O|$ The cardinality of the set O. The number of elements in O.

When we talk about groups we'll be implicitly referring to groups over an elliptic curve, specifically Curve25519. Note that the protocol is not limited to this group and should be usable with any abelian group where the Computational Diffie-Hellman assumption and any other mentioned properties hold. Also note that the same group needs to be used by both this and the DKG protocol.[3]

A participant is an entity[4] who took part in the DKG protocol and was assigned a share of the distributed private key. Participants do not necessarily take part in the signature protocol; they may be offline or simply choose not to. Honest participants will take part in the signature protocol whenever possible, but adversarial participants may choose not to.

Clients are any entity that wants to know the shared random. Participants can be, but are not necessarily clients.

0.3 Protocol description

Before performing our nonce creation protocol a setup phase must be performed. This setup phase does *not* need to be run every time, it only needs to be performed once per x nonce generations, where x is some arbitrary number agreed upon by all the participants.

During this setup phase the n directory authorities run a DKG (Distributed Key Generation) protocol to produce a shared private key / public key pair. The private key is *not* constructed, instead each participant keeps their share of the private key secret indefinitely.

- Denote the i'th participant as S_i. S_i can be used as an integer, typically $S_i = i$ or $i+1$. All integers S_i *must* be unique and non-zero. These values are publicly known and are also used by the DKG protocol.
- Denote participant S_i's share of the shared private key as s_i. S_i learns this from the DKG protocol and keeps it secret; it is never revealed.

[3] We can get around this by recreating the DKG public key shares $P_i = s_i.B$ within our own group. The groups must still have matching prime order p, however.

[4] More specifically a directory authority.

- Denote the shared private key as x. x is *never* constructed and is always unknown. It however *would* be possible to construct x by interpolating a set of t or more private key shares s_i, where t is the DKG threshold.
- Denote B as an element of the group G that generates a subgroup of prime order p. Note that this subgroup has the same order as $\mathbb{F}_p$ [this is important]. B is publicly known from the DKG protocol.
- Denote each participant's share of the public key as $P_i = s_i \cdot B$. All P_i are publicly known from the DKG protocol.
- Denote the shared public key as $x \cdot B$. This value is publicly known and can be constructed by interpolating the public key shares P_i.

When it's time to create a new shared random the participants all calculate R. R is a function of time and can be calculated by anybody as:

$$R = \mathrm{H}(\text{“tor-hs-rand-base-point”}||T)$$

Where H is some hash function that outputs elements of the subgroup generated by B. Where T is the starting time of the current time period. And where “tor-hs-rand-base-point” is a publicly known value.

Each participant uses their share of the private key, s_i, to create a share of the threshold signature $x \cdot R$. They then publish that value along with a non-interactive zero-knowledge proof of the signature share's validity.

- Denote the actual share of the signature as $Q_i = s_i \cdot R$.
- Denote the proof of Q_i's validity as PROOF_i.

The creation and use of PROOF_i is discussed in Section 0.3.2.

For convenience we'll bundle those together along with S_i into a single published share message, SHARE_i:

$$\mathrm{SHARE}_i = S_i||Q_i||\mathrm{PROOF}_i$$

These SHARE messages are published using the list creation protocol's *Publish* operation (Section 0.3.3).

0.3.1 Calculation of RAND

This half of the protocol is performed by clients; anybody who wants to know the value of the nonce, RAND.

Once the time period begins[5] the clients are able to perform the *Fetch* operation of the list creation protocol (Section 0.3.3). Clients use this operation to download a list of all valid published signature shares and then use that to calculate RAND.

When a client downloads the list of signature shares they check the validity[6] of each using its attached proof and public information known from the DKG protocol.[7] If the list contains enough valid shares the shares are then interpolated to get the threshold signature $x \cdot R$. We then use that threshold signature as the value of RAND.

When the list doesn't contain enough valid shares to construct the threshold signature, or when no list was successfully created, we need a fallback value for RAND. A simple fallback value[8] is to set RAND $= R$.

This leaves us with the following function for calculating RAND:

$$\text{RAND} = f(O) = \begin{cases} \text{Interpolate}(O) \equiv x \cdot R & \text{if } |O| \geq t \\ R & \text{if } |O| < t \end{cases}$$

For $O \subseteq \{(S_1, Q_1), ..., (S_n, Q_n)\}$, the set of signature shares from the fetched list who pass validation, and where t is the DKG threshold.

The interpolation function[9] is defined as follows:

$$\text{Interpolate}((x_1, y_1), ..., (x_k, y_k)) = \sum_{i=1}^{k} \lambda(x, i) \cdot y_i$$

where

$$\lambda(x, i) = \prod_{\substack{1 \leq k \leq |x| \\ k \neq i}} \frac{x_k}{x_k - x_i} \in \mathbb{F}_p$$

[5] When the current time = T.

[6] Depending on its implementation the list creation protocol may have already performed validation on the signature shares for us. It is, however, good practice for the client to be careful and perform validation regardless.

[7] This publicly known DKG information can be included with the share list for convenience.

[8] While it's not particularly unpredictable this fallback value won't actually be used unless something is seriously wrong with the network.

[9] Note that this interpolation function will only work if the DKG protocol is based on Shamir's Secret Sharing or one of its derivatives (the most likely case); if that *isn't* the case then it's likely a different method of interpolation will be required (as defined by the DKG protocol).

0.3.2 Validation of Q_i

An important part of this protocol is that when a client receives a signature share they must be able to check its validity. A valid signature share will always be of the form $Q_i = s_i \cdot R$, where R is publicly known but s_i is a secret only known by S_i, the share's creator.

To prove their share is valid each participant publishes a Fiat-Shamir zero knowledge proof that $\text{dlog}_R(Q_i) = \text{dlog}_B(P_i)$ along with it. Since P_i is a share of the public key from the DKG protocol we already know that that $P_i = s_i \cdot B$, and hence if we can prove that $\text{dlog}_R(Q_i) = \text{dlog}_B(P_i)$ then we've also proved that $\text{dlog}_R(Q_i) = s_i$; all without ever revealing the value of s_i.

We will refer to the proof created by S_i as PROOF_i. The proof takes the following form:

$$\text{PROOF}_i = (U, V, z)$$

where

$$\begin{aligned} U &= a \cdot B \\ V &= a \cdot R \\ z &= a + s_i * c \\ a &= \text{random} \in \mathbb{F}_p \\ c &= \text{Hash}(U||V||m) \in \mathbb{F}_p \\ m &= (S_i||T) \end{aligned}$$

Where $\text{Hash}(x)$ is some hash function[10] whose output is in $\mathbb{F}_p$.

PROOF_i can then be verified (using only publicly available information)[11] by checking that:

$$\begin{aligned} z \cdot B &= U + c \cdot P_i \\ &= a \cdot B + c \cdot (s_i \cdot B) \\ &= a \cdot B + (c * s_i) \cdot B \\ &= (a + c * s_i) \cdot B \end{aligned}$$

[10] Note that Hash() is separate from the hash function H() used in calculation of R.

[11] Note that it's important none of the public information used in validation is supplied by the creator of the signature share, unless such information is signed by a majority of the participants. Allowing that could enable them to provide false values for R, B, or m and subvert validation of the proof.

and

$$\begin{aligned} z \cdot R &= V + c \cdot Q_i \\ &= a \cdot R + c \cdot (s_i \cdot R) \\ &= a \cdot R + (c * s_i) \cdot R \\ &= (a + c * s_i) \cdot R \end{aligned}$$

If both of those tests hold then we consider Q_i to be valid.

0.3.3 List creation

Here we give an overview of what we require from the list creation protocol. We've attempted to outline the protocol in the most general terms possible. The purpose of a separate list creation protocol is to avoid having to explicitly deal with the Byzantine General's Problem [**?**]. This simplifies the protocol greatly and allows implementers to make use of existing tools and frameworks rather than implementing a single specific Byzantine fault tolerant system.

We outline the list creation protocol in terms of two basic operations:

Publish(**SHARE**$_i$) A participant publishes a SHARE message which they have created such that when Time $\geq T$, the start of the time period, the SHARE message is publicly available for download by all clients via the *Fetch* operation.[12]

Fetch() A client downloads the list of all published SHARE messages for the current time period. In our model the client is able to download this list from any participant, however, there's no reason the list couldn't instead be served by a third party. It is important that the underlying protocol ensures all clients download the same exact list; there should be a maximum of one valid list per time period, clients must be able to determine whether a fetched list is the valid one, and the valid list should be available to all clients. Additionally, clients must be able to determine if *no* valid list is available (determine if list creation failed).

Any list creation protocol that provides this interface should be usable in the protocol.

We can provide security and efficiency benefits if we apply some restrictions to which SHARE messages will can be successfully published. The first restriction we add is:

[12] The SHARE message may be made publicly available before T, however, the clients shouldn't attempt to download it before then.

> Publish(SHARE_i) will only succeed if PROOF_i passes validation on the majority of participants.

The logic behind this is that SHARE messages with invalid PROOFs are of no use to clients, so there is no point adding them to the fetched list; doing so would only serve to inflate the list with garbage.

Recall that PROOF_i can be validated using *only* publicly available information. We assume that all clients have the same set of public information, and therefore all honest clients will validate PROOF_i the same. The overall protocol relies on the assumption that the majority of participants are honest. Using that we know that the majority of participants will have the same, correct validation result for PROOF_i.

The second restriction we add is:

> Each participant should be restricted to publishing a maximum of one SHARE message successfully per time period.

The purpose of this restriction is to prevent adversarial participants from publishing thousands of SHARE messages at a time, which would cause the fetched list to become inflated and potentially even cause denial of service or crashes if enough were published.

This restriction is reasonable as there is always exactly one *correct* value for Q_i each time period. If S_i publishes multiple SHARE_i messages during a single time period then it follows that one of these are true:

- The additional SHAREs are invalid, and should be ignored because they're useless.
- The additional SHAREs are valid and contain the same value for Q_i; in this case the additional SHARE messages are redundant and can be ignored.
- The additional SHAREs are valid but contain different values for Q_i. This means S_i has managed to subvert the zero knowledge proof and none of their SHAREs should be trusted because we are unable to determine which of them contains the correct value for Q_i.

Additionally because of restriction one we don't need to worry about adversarial participants impersonating honest participants and publishing SHARE messages on their behalf. Creation of PROOF_i requires knowledge of s_i, which is a value

that is *only* known by S_i. Creation of a valid PROOF_i without knowledge of s_i is considered computationally infeasible in the random oracle model [**?**]; therefore forgery of valid a SHARE message is also considered computationally infeasible.

List creation using Tor

One way to implement the list protocol is to use pre-existing features of Tor.[13]

The Tor network has a centralised directory protocol which is used to distribute information about the network, such as lists of onion-routers which clients can relay their traffic through.

The basic idea behind their directory protocol is that there is a small set of directory authorities on the network.[14] Routers on the Tor network upload information about themselves to each of the directory authorities.

Periodically the directory authorities generate a summary of all the router information they've received (this is called a vote). They then each share their list with the other directory authorities and work together to agree upon a "consensus status" document, which is a summary of which routers are part of the network and basic information about them.

Clients are then able to download this summary and use its contained information to interact with the Tor network.

It's not too hard to see similarities between this and our list protocol. When routers upload information about themselves this corresponds to the *publish()* operation. The authorities agree upon a consensus-status document, which is very similar to our SHARE list, and then make it available to clients for download (available for *fetch()*'ing).

The Tor directory protocol must defend against the same Byzantine attacks and faults that our list protocol does. Additionally, since the directory protocol's purpose is to distribute information about the Tor network it's logical that we use it for distributing our SHARE list rather than creating something separate.

[13] This method of course is designed for use in the Tor network. I wouldn't suggest it in other cases.

[14] New clients know about these because a listing of the default directory authorities comes bundled with the Tor software.

0.4 Changes to the original

Here is a summary of things that have been changed from the originally suggested protocol and the reasons for doing so:

- Changed SHARE_i to not include R since R is publicly calculable. Could be replaced by T if a time-period identifier is required.
- Added fallback value RAND = R. The notion of a fallback was in the original, but one wasn't explicitly defined.
- ~~Changed the value m used in the ZKP based on comments made on tor-dev.~~
- Defined the list protocol stuff more explicitly.
- Removed the signature from SHARE messages since I couldn't come up with a satisfying justification. Regardless it's likely that the list creation protocol would require (and provide) such functionality.
- Changed the signature to use "tor-hs-rand-base-point"$||T$ instead of R. The idea is that this will avoid possible hash collisions in R. The signature was removed, but if it's ever added back I'd suggest doing this.