# Swarm Formal Specification

Swarm Research Division

draft of August 12, 2025

# Contents

# List of definitions

# List of Figures

# List of Tables

# Chapter 1

# Formal specification

## 1.1 DISC basics

### 1.1.1 Notation

— roman – predicates standing for blockchain verifiable properties
— SMALLCAPS– field of a composite, tuple member function
— $\Pi^p$ – proof constructor function
— $\mathbb{V}^p$ – validator function
— `TT_FONT` – constant parameter (see appendix **??**)
— $uint[d]$ – left closed, right open integer range $\overline{[0, 2^d)}$

### 1.1.2 Sequences

**Definition 1 – sequences .**
Define $\tau\{n\}$, the non-polymorphic sequences of length $n$ (non-negative) over any type $\tau$ as an indexing function:

$$\tau\{n\} \stackrel{\text{def}}{=} \begin{cases} \varnothing & \text{if } n = 0 \\ \overline{0, n-1} \to \tau & \text{if } n > 0 \end{cases} \tag{1.1}$$

$$\tau+ \stackrel{\text{def}}{=} \bigcup_{n \in \mathbb{Z}^+} \tau\{n\} \tag{1.2}$$

$$\tau* \stackrel{\text{def}}{=} \{\varnothing\} \cup \tau+ \tag{1.3}$$

$$\tag{1.4}$$

Length function *len*:

$$len(s) \quad : \quad \tau \to uint64 \tag{1.5}$$

$$len(s) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } s = \varnothing \\ n & \text{if } s \in \tau\{n\} \end{cases} \tag{1.6}$$

The positional index 'at' function:

$$\texttt{[]} \quad : \quad \tau* \to \times uint64 \to \tau \tag{1.7}$$

$$s\,\texttt{[]} \quad : \quad \overline{0, len(s) - 1} \to \tau \tag{1.8}$$

$$s\,\texttt{[}i\texttt{]} \stackrel{\text{def}}{=} s'(i) \tag{1.9}$$

Define the *concatenation* operator '$\oplus$':

$$(x \oplus y) \quad : \quad \overline{0, len(x) + len(y) - 1} \to \tau \tag{1.10}$$

$$(x \oplus y)\,\texttt{[}i\texttt{]} \stackrel{\text{def}}{=} \begin{cases} x\,\texttt{[}i\texttt{]} & \text{if } 0 \leq i < len(x) \\ y\,\texttt{[}i - len(x)\texttt{]} & \text{otherwise} \end{cases} \tag{1.11}$$

Define slices (subsequences) with the range operator ':':

$$(s\,\texttt{[}o\!:\!o + l\texttt{]}) \quad : \quad \overline{0, l - 1} \to \tau \tag{1.12}$$

$$(s\,\texttt{[}o\!:\!o + l\texttt{]})\,\texttt{[}i\texttt{]} \stackrel{\text{def}}{=} s\,\texttt{[}o + i\texttt{]} \tag{1.13}$$

where

$$o, l \geq 0 \wedge \tag{1.14}$$

$$len(x) \geq o + l \tag{1.15}$$

Let us also define empty, prefix and suffix slices:

$$s\,\texttt{[}x\!:\!x\texttt{]} \stackrel{\text{def}}{=} \varnothing \tag{1.16}$$

$$s\,\texttt{[}:\!x\texttt{]} \stackrel{\text{def}}{=} s\,\texttt{[}0\!:\!x\texttt{]} \tag{1.17}$$

$$s\,\texttt{[}x\!:\texttt{]} \stackrel{\text{def}}{=} s\,\texttt{[}x\!:\!len(s)\texttt{]} \tag{1.18}$$

As a special case byte slices are defined as sequences of 8-bit integers.

**Definition 2 – segmentation.**
Define segment as an at most 32 long byte slice, and define segmentation of a slice of bytes as the partitioning of the slice into consecutive segments. Define segment count as the number of segments that cover a byte slice:

$$SegCnt \quad : \quad byte* \to uint64 \tag{1.19}$$

$$SegCnt(s) \stackrel{\text{def}}{=} int\left(\frac{len(s) - 1}{32}\right) + 1 \tag{1.20}$$

Now we can define the segment indexing function `[[]]` which maps the byte slice $s$ and an index $i$ to the $i$-the segment in the segmentation of $s$:

$$
\begin{aligned}
\texttt{[[]]} &: \quad \mathit{byte}{*} \rightarrow \mathit{uint64} \rightarrow \mathit{Segment} && (1.21)\\
s\texttt{[[]]} &: \quad \overline{0, \mathit{SegCnt}(s)-1} \rightarrow \mathit{Segment} && (1.22)\\
s\texttt{[[}i\texttt{]]} &\stackrel{\text{def}}{=} \begin{cases} s\texttt{[}32\cdot i : \texttt{]} & \text{if } i = \mathit{SegCnt}-1 \\ s\texttt{[}32\cdot i : 32\cdot(i+1)\texttt{]} & \text{otherwise} \end{cases} && (1.23)
\end{aligned}
$$

### 1.1.3 Custom types

**Definition 3 – Swarm overlay address of node $n$.**
A Swarm node is associated with an Ethereum account that the node operator must possess the private key for, called *bzz account* ($K_n^{bzz}$). The node's overlay address is derived as the hash of the binary serialisation of the Ethereum address of this account with the Swarm network ID and a minable nonce appended.

$$\mathit{overlay}(n) \stackrel{\text{def}}{=} H(\mathit{acc} \oplus \mathit{id} \oplus \nu) \tag{1.24}$$

where

$$
\begin{aligned}
\text{ETH ADDRESS } \mathit{acc} &= \mathit{Account}(K_n) && (1.25)\\
\text{NETWORK ID } \mathit{id} &= \texttt{BZZ\_NETWORK\_ID} && (1.26)\\
\text{OVERLAY NONCE } \nu &\in \mathit{Nonce} && (1.27)
\end{aligned}
$$

**Definition 4 – DISC custom types .**
Let us define the DISC specific custom types used in the formalisation.

$\mathit{Segment} \equiv \mathit{byte}\{32\} \equiv \mathit{uint256}$
  32-long slice of raw bytes
  in numerical context cast as BigEndian encoded 256-bit unsigned integer
$\mathit{Address} \equiv \mathit{Segment}$
  swarm chunk address, swarm peers' overlay address
$\mathit{Nonce} \equiv \mathit{Segment}$
  deterministically random *Segment*
$\mathit{Account} \equiv \mathit{byte}\{20\}$
  Ethereum address deriveed from EC keypair $K$
  $\mathit{Account}(K) \stackrel{\text{def}}{=} H(\mathit{PubKey}(K))\texttt{[}12:32\texttt{]}$
$\mathit{Nodes} \equiv \mathit{Segment}$
  swarm client node (peer)

$Sig \equiv byte\{65\}$
$\langle r, s, v\rangle$ representation of an EC signature (32+32+1 bytes)
$Timestamp \equiv uint64$
64-bit unsigned integer for unix time, nanosecond resolution
big endian binary serialisation.
$H : byte* \rightarrow Segment$
the 256-bit Keccak SHA3 hash function, the base hash used in swarm.

### 1.1.4 XOR distance and proximity order

**Definition 5 – XOR distance ($\chi$).**
Consider the set of bit sequences with fixed length $d$ as points in a space. Define a distance metric $\chi$ such that the distance between two such sequences is the numerical value of their bitwise XOR (^) using big endian (= most significant bit first) encoding.

$$\chi \quad : \quad uint[d] \times uint[d] \rightarrow uint[d] \tag{1.28}$$

$$\chi(x, y) \stackrel{\text{def}}{=} Uint^d(BE^d(x) \hat{} BE^d(x))) \tag{1.29}$$

Given the fixed length $d > 0$, there is a maximum distance ($2^d - 1 = \chi(0\{d\}, 1\{d\})$) in this space, and thus we can define the notion of *normalised distance*:

$$\overline{\chi} \quad : \quad uint[d] \times uint[d] \rightarrow \mathbb{Q}[0, 1] \tag{1.30}$$

$$\overline{\chi}(x, y) \stackrel{\text{def}}{=} \frac{\chi(x, y)}{2^d - 1} \tag{1.31}$$

**Definition 6 – Proximity order ($PO$ ).**
Proximity order (PO) is a discrete logarithmic scaling of proximity.

$$PO \quad : \quad uint[d] \times uint[d] \rightarrow \overline{0, d} \tag{1.32}$$

$$PO(x, y) \stackrel{\text{def}}{=} \begin{cases} d & \text{if } x = y \\ int(\log_2(Proximity(x, y))) & \text{otherwise} \end{cases} \tag{1.33}$$

where proximity is the inverse of normalised distance:

$$Proximity \quad : \quad uint[d] \times uint[d] \rightarrow uint[d] \tag{1.34}$$

$$Proximity(x, y) \stackrel{\text{def}}{=} \frac{1}{\overline{\chi}(x, y)} \tag{1.35}$$

Given two points $x$ and $y$, the order of their proximity $PO(x, y)$ equals the number of initial bits shared by their respective most significant bit first binary representations. In practice, with $d = 256$, $uint[d] \equiv Segment$, so $PO$ also applies to a pair of slices of 32 bytes.

### 1.1.5 Binary Merkle tree hash

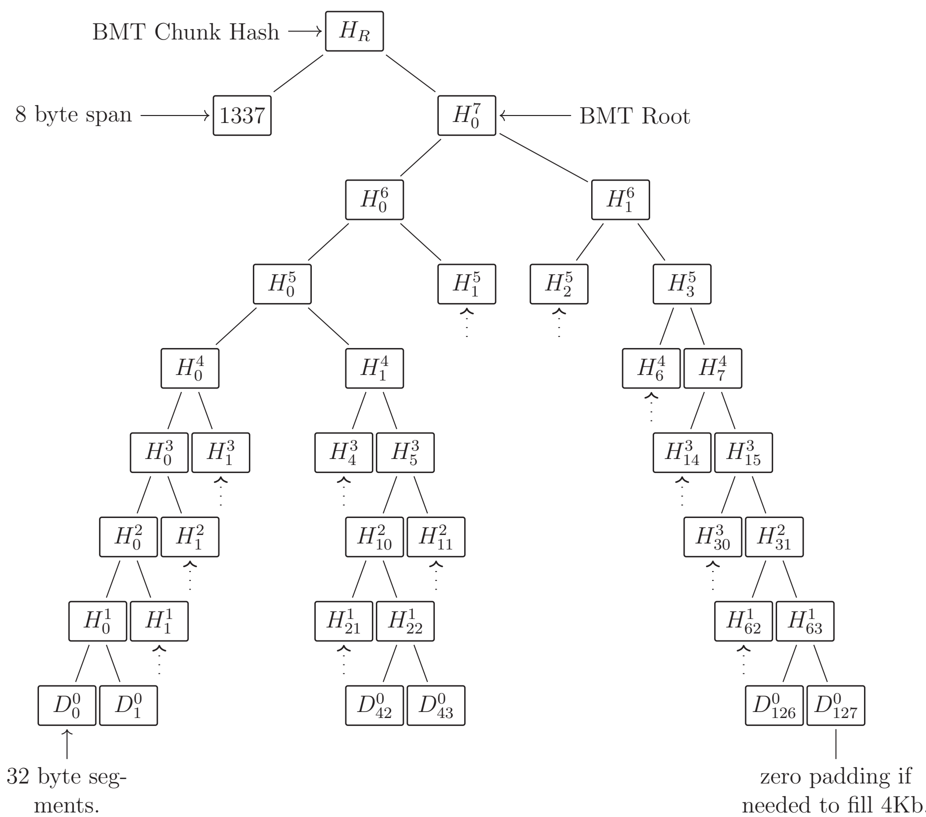


**Figure 1.1:** BMT (Binary Merkle Tree) used as the native chunk hash in Swarm. In this example, 1337 bytes of chunk data is segmented into 32 byte segments. Zero padding is used to fill up the rest up to 4 kilobytes. Pairs of segments are hashed together using the ethereum-native Keccak256 hash to build up the binary tree. On level 8, the binary Merkle root is prepended with the 8 byte span and hashed to yield the BMT chunk hash.

The BMT chunk address is the hash of the 8 byte metadata (span) and the root hash of a *binary Merkle tree* ($BMT$) built on the 32-byte segments of the underlying data (see figure 1.1). If the chunk content is less than 4k, the hash is calculated as if the chunk was padded with all zeros up to 4096 bytes.

**Definition 7 – Binary Merkle Tree Root .**
Define $\triangle[H, n](i, d)$ as the Binary Merkle Tree Root of data $d$ fitting in at most $2^n$ 32-byte

segments using $H$ as its base hash:

$$\begin{aligned}
\triangle \quad &: \quad (byte* \to Segment) \times uint8 \to uint8 \times byte* \to Segment && (1.36)\\
\triangle[H, n] = \quad &: \quad \overline{0, n} \times byte\{, 2^n\} \to Segment && (1.37)\\
\triangle[H, n](i, d) \quad &\stackrel{\text{def}}{=} \quad \begin{cases} d & \text{if } i = 0 \\ \triangle[H, n](i, d \oplus 0\{2^{n+5} - len(d)\}) & \text{if } len(d) < 2^{i+5} \\ H(\triangle[H, n](d[:2^{i-1}], i-1) \oplus & \\ \quad \oplus \triangle[H, n](d[2^{i-1}:], i-1)) & \text{otherwise} \end{cases} && (1.38)
\end{aligned}$$

**Definition 8 – Binary Merkle Tree Hash .**
Define $BMT[H](d, m)$ as the hash of Binary Merkle Tree Root of chunk length data $d$ prepended with metadata $m$. $H$ is the base hash function, by default 256-bit Keccak. Note that a chunk size blob of bytes can accommodate 128 32-byte segments, hence depth 7:

$$\begin{aligned}
BMT \quad &: \quad (byte* \to Segment) \to Chunk \times byte\{8\} \to Segment && (1.39)\\
BMT[H] \quad &: \quad Chunk \times byte\{8\} \to Segment && (1.40)\\
BMT[H](d, m) \quad &\stackrel{\text{def}}{=} \quad H(m \oplus \triangle[H, 7](7, d)) && (1.41)
\end{aligned}$$

### 1.1.6 Chunks

**Definition 9 – Content addressed chunks .**
Define content address chunk $c$ as a function from bytes data with size limit of 4096 bytes and an associated address calculated with BMT:

$$\begin{aligned}
CAC \quad &: \quad Chunk \times byte\{8\} \to Chunks && (1.42)\\
CAC(d, m) \quad &\stackrel{\text{def}}{=} \quad \langle addr, cont \rangle && (1.43)
\end{aligned}$$

such that

$$\begin{aligned}
addr \quad &\stackrel{\text{def}}{=} \quad BMT(d, m) && (1.44)\\
cont \quad &\stackrel{\text{def}}{=} \quad m \oplus d && (1.45)
\end{aligned}$$

By convention the metadata prefix $m$ encodes the *span* using 64-bit little endian. If the chunk is an intermediate chunk (see definition 11), the span is the length of the data that the subtree spans over. If the chunk is a data chunk, then span encodes the data length:

$$m \stackrel{\text{def}}{=} LE^{64}(len(d)) \qquad (1.46)$$

We also say that for $cac = CAC(d, m) = \langle addr, cont \rangle$:

$$\textsc{address}(cac) \stackrel{\text{def}}{=} addr \tag{1.47}$$
$$\textsc{payload}(cac) \stackrel{\text{def}}{=} cont \tag{1.48}$$
$$\textsc{data}(cac) \stackrel{\text{def}}{=} d \tag{1.49}$$
$$\textsc{metadata}(cac) \stackrel{\text{def}}{=} m \tag{1.50}$$

For convenience *SegCnt* and the segment indexing function '`[[]]`' can be trivially extended to apply to chunks:

$$SegCnt \quad : \quad Chunks \rightarrow uint64 \tag{1.51}$$
$$SegCnt(c) \stackrel{\text{def}}{=} SegCnt(\textsc{data}(c)) \tag{1.52}$$

$$\texttt{[[]]} \quad : \quad Chunks \times uint64 \rightarrow Segment \tag{1.53}$$
$$c\texttt{[[}i\texttt{]]} \stackrel{\text{def}}{=} \textsc{data}(c)\texttt{[[}i\texttt{]]} \tag{1.54}$$

### 1.1.7 Single Owner Chunks

**Definition 10 – Single owner chunks .**
A single owner chunk is defined as a content addressed chunk associated with an ID and an ethereum address:

$$SOC \stackrel{\text{def}}{=} Account \times Segment \times CAC \rightarrow Chunks \tag{1.55}$$
$$SOC(owner, id, cac) \stackrel{\text{def}}{=} \langle addr, cont \rangle \tag{1.56}$$

where

$$addr \stackrel{\text{def}}{=} H(o \oplus id) \tag{1.57}$$
$$cont \stackrel{\text{def}}{=} id \oplus Sig(o, id \oplus \textsc{address}(cac)) \oplus \textsc{payload}(cac) \tag{1.58}$$

A single owner chunk's address is the Keccak256 hash of identifier prepended to owner account, while its data is serialised as follows:

- *identifier* – 32 bytes arbitrary identifier,
- *signature* – 65 bytes $\langle r, s, v \rangle$ representation of an EC signature (32+32+1 bytes),

– *span* – 8 byte little endian binary of uint64 chunk span,
– *data* – max 4096 bytes of regular chunk data.

Integrity of a *single owner chunk* is verified with the following process:

1. Deserialise the chunk content into fields for identifier, signature and payload.
2. Construct the expected plaintext composed of the identifier and the *BMT hash* of the payload.
3. Recover the owner's address from the signature using the plaintext.
4. Check the hash of the identifier and the owner (expected address) against the chunk address.

**Definition 11 – Packed address chunk.**
Define the packed address chunk for a sequence of chunks $C$ as the concatenation of all the addresses of the chunks in the sequence:

$$\begin{aligned} PAC &: \mathit{Chunks}{*} \times \mathit{byte}\{8\} \rightarrow \mathit{Chunks} & (1.59)\\ PAC(C,m) &\stackrel{\text{def}}{=} CAC\left(\bigoplus_{i=0}^{len(C)-1} \mathit{Address}(C\texttt{[}i\texttt{]}), m\right) & (1.60)\end{aligned}$$

### 1.1.8 Segment inclusion proofs

Using BMT hashes allows for compact *segment inclusion proofs* (substring relationship with a 32-byte resolution).

**Definition 12 – BMT segment inclusion proof .**
Define $\Pi^{\text{SIP}}(c,i)$ as the *BMT* inclusion proof on chunk $c$ for segment index $i$:

$$\begin{aligned} \Pi^{\text{SIP}} &: \mathit{Chunks} \times \overline{0,127} \rightarrow \mathit{SIP} & (1.61)\\ \mathit{SIP} &\stackrel{\text{def}}{=} \mathit{Segment} \times \mathit{Segment}^7 \times \mathit{byte}\{8\} & (1.62)\\ \Pi^{\text{SIP}}(c,i) &\stackrel{\text{def}}{=} \langle c\texttt{[[}i\texttt{]]}, \langle h_0, h_1, \ldots, h_6\rangle, \textsc{metadata}(c)\rangle & (1.63)\end{aligned}$$

where

$$\begin{aligned} h_j &\stackrel{\text{def}}{=} BMT(s_j, j) & (1.64)\\ s_j &\stackrel{\text{def}}{=} c\texttt{[}\mathit{start}(i,j) : \mathit{start}(i,j) + 32 \cdot 2^j\texttt{]} & (1.65)\end{aligned}$$

where

$$\mathit{start}(i,j) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } j = 7 \\ \mathit{start}(i,j+1) & \text{if } \mathit{int}\,(i/2^j) = 0 \mod 2 \\ \mathit{start}(i,j+1) + 32 \cdot 2^j & \text{otherwise} \end{cases} \qquad (1.66)$$

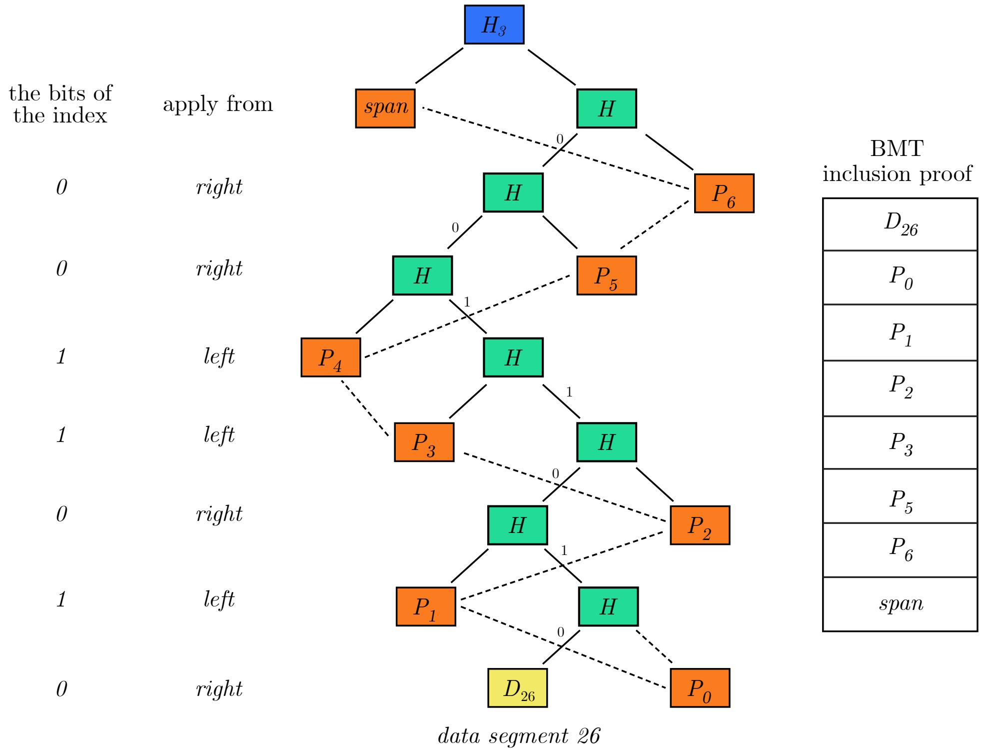


**Figure 1.2:** Compact segment inclusion proofs for chunks. Assume we need proof for segment 26 of a chunk (yellow). The orange hashes of the BMT are the sister nodes on the path from the data segment up to the root and constitute what needs to be part of a proof. When these are provided together with the root hash and the segment index, the proof can be verified. The side on which proof item $i$ needs to be applied depends on the $i$-th bit (starting from least significant) of the binary representation of the index. Finally the span is prepended and the resulting hash should match the chunk root hash.

In order to validate segment inclusion proofs we first introduce the prover hash function $H_\Pi$.

**Definition 13 – BMT prover function.**

$$H_\Pi \quad : \quad \overline{0,127} \times SIP \rightarrow Address \tag{1.67}$$

$$H_\Pi(i, \langle d, sisters, m \rangle) \stackrel{\text{def}}{=} H(m, H_\Pi^{\triangle}(7, d, sisters)) \tag{1.68}$$

where

$$
\begin{aligned}
H^{\triangle}_{\Pi} &: \overline{0,7} \times \mathit{Segment} \times \mathit{Segment}^7 \rightarrow \mathit{Address} && (1.69) \\
H^{\triangle}_{\Pi}(j,d,s) &\stackrel{\text{def}}{=} \begin{cases} d & \text{if } j = 0 \\ H(H^{\triangle}_{\Pi}(j-1,d,s) \oplus s\texttt{[}j-1\texttt{]}) & \text{if } \mathit{int}(i/2^{j-1}) = 0 \mod 2 \\ H(s\texttt{[}j-1\texttt{]} \oplus H^{\triangle}_{\Pi}(j-1,d,s)) & \text{otherwise} \end{cases} && (1.70)
\end{aligned}
$$

**Definition 14 – BMT SIP validation .**
Define $\mathbb{V}^{\text{SIP}}(a,i,p)$ as the validator of a BMT segment inclusion proof $p$ for chunk at address $a$ on segment index $i$:

$$
\begin{aligned}
\mathbb{V}^{\text{SIP}} &: \mathit{Segment} \times \overline{0,127} \times \mathit{SIP} \rightarrow \{\texttt{T},\texttt{F}\} && (1.71) \\
\mathbb{V}^{\text{SIP}}(a,i,p) &\Leftrightarrow H_{\Pi}(i,p) = a && (1.72)
\end{aligned}
$$

**Definition 15 – Single owner chunks data integrity proof .**
Define a single owner chunk storage proof $\Pi^{\text{SOC}}(c,i)$ as a segment inclusion proof of the data payload of SOC $c$ on index $i$ together with the ID and signature of SOC $c$:

$$
\begin{aligned}
\mathit{SIP}_{\mathit{SOC}} &\stackrel{\text{def}}{=} \mathit{SIP} \times \mathit{Sig} \times \mathit{Segment} && (1.73) \\
\Pi^{\text{SIP[SOC]}} &: \mathit{SOC} \times \overline{0,127} \rightarrow \mathit{SIP}_{\mathit{SOC}} && (1.74) \\
\Pi^{\text{SIP[SOC]}}(\langle o, \mathit{id}, \mathit{cac}\rangle, i) &\stackrel{\text{def}}{=} \langle p, \mathit{sig}, \mathit{id}\rangle && (1.75)
\end{aligned}
$$

where

$$
\begin{aligned}
p &= \Pi^{\text{SIP}}(\mathit{cac}, i) && (1.76) \\
\mathit{sig} &= \mathit{Sig}(o, \mathit{id} \oplus \mathit{address}(\mathit{cac})) && (1.77)
\end{aligned}
$$

**Definition 16 – Single owner chunks data integrity validation .**
Define $\mathbb{V}^{\text{SIP[SOC]}}(a,i,p)$ as the validator of a single owner chunk storage proof $p$ for chunk at address $a$ on segment index $i$:

$$
\begin{aligned}
\mathbb{V}^{\text{SIP[SOC]}} &: \mathit{Address} \times \overline{0,127} \times \mathit{SIP}_{\mathit{SOC}} \rightarrow \{\texttt{T},\texttt{F}\} && (1.78) \\
\mathbb{V}^{\text{SIP[SOC]}}(a,i,\langle p, \mathit{sig}, \mathit{id}\rangle) &\Leftrightarrow a = H(\mathit{id} \oplus o) && (1.79)
\end{aligned}
$$

such that

$$
\begin{aligned}
\text{OWNER } o &= \mathit{ECRecover}(\mathit{sig}, \mathit{id} \oplus a') && (1.80) \\
\text{PAYLOAD } a' &= H_{\Pi}(i,p) && (1.81)
\end{aligned}
$$

### 1.1.9 Postage stamps

**Definition 17 – Postage stamps .**

$$\begin{aligned}
\mathit{Stamps} &\stackrel{\text{def}}{=} \mathit{Segment} \times \mathit{uint64} \times \mathit{Timestamp} \times \mathit{Address} && (1.82)\\
ps &= \langle b, i, ts, a\rangle \in \mathit{Stamps} && (1.83)\\
\textsc{batch}\mathrm{ID}(ps) &\stackrel{\text{def}}{=} b && (1.84)\\
\textsc{index}(ps) &\stackrel{\text{def}}{=} i && (1.85)\\
\textsc{timestamp}(ps) &\stackrel{\text{def}}{=} ts && (1.86)\\
\textsc{address}(ps) &\stackrel{\text{def}}{=} a && (1.87)
\end{aligned}$$

**Definition 18 – Storage slot reference .**
Define the *storage slot reference* $slot(ps)$ of a postage stamp $ps$ as the tuple of the batch identifier and the within-batch stamp counter:

$$\begin{aligned}
\mathit{Slots} &\stackrel{\text{def}}{=} \mathit{Segment} \times \mathit{uint64} && (1.88)\\
\mathit{slot} &: \mathit{Stamps} \rightarrow \mathit{Slots} && (1.89)\\
\mathit{slot}(ps) &\stackrel{\text{def}}{=} \langle \textsc{batch}\mathrm{ID}(ps), \textsc{index}(ps)\rangle && (1.90)
\end{aligned}$$

**Definition 19 – Postage stamp validity .**
Define $\mathbb{V}^{\text{STAMP}}(ps)$ as the validator of the proof of relevance expressed as the postage stamp $ps$ relying on blockchain information:

$$\begin{aligned}
\mathbb{V}^{\text{STAMP}} &: \mathit{Stamps} \times \Gamma \times \mathit{Nodes} \rightarrow \{\mathtt{T}, \mathtt{F}\} && (1.91)\\
\mathbb{V}^{\text{STAMP}}(ps, \gamma, n) &\Leftrightarrow && (1.92)\\
\textsc{authentic} &\quad \textsc{batch}\mathrm{ID}(ps) \in \mathrm{Batches}(\gamma) \wedge && (1.93)\\
\textsc{alive} &\quad \mathrm{Balance}(ps) > 0 \wedge && (1.94)\\
\textsc{authorised} &\quad \mathit{ECRecover}(\mathit{Sig}(ps), \mathit{encode}(ps)) = \mathrm{Owner}(ps) \wedge && (1.95)\\
\textsc{available} &\quad 0 <= \textsc{index}(ps) < \mathrm{Size}(ps) \wedge && (1.96)\\
\textsc{aligned} &\quad \mathit{PO}(\textsc{index}(ps), n) \geq \textsc{depth}(\mathit{reveal}(\gamma, n)) && (1.97)
\end{aligned}$$

### 1.1.10 Ordering and sampling

**Lemma 20 – Ordering and indexing functions.**
Given an arbitrary finite set $C$, and another set $I$ with a total order $<$. Any invertible

function total over $C$, $f : C \to I$ defines a total order $<_f$ over $C$ as follows:

$$<_f \quad \subseteq \quad C \times C \tag{1.98}$$
$$\forall c, c' \in C, c <_f c' \quad \Leftrightarrow \quad f(c) < f(c') \tag{1.99}$$

*Proof.* $f$ is injective wrt $I$, so $Image(f) = J \subseteq I$. Since $<$ restricted to a subset ($<_J$) is also a total order over $J$. Since $f$ is invertible, $C$ and $J$ are isomorphic and therefore the total order $<$ on $J$ carries over to $C$.

**Corollary 21 – hash orders.**
Any hash function defines a total order on a finite set of byte slices.

*Proof.* The collision free nature of the hash function makes it practically invertible. The actual hashes when read as binary encodings of integers, offer a natural integer ordering over the values.

Example: the prefixed BMT hash transform (see 28) defines a total order over a set of chunks.

**Definition 22 – Sampler function.**
Using $f$ and its derivative ordering on $C \subseteq Dom(f)$ we represent $C$ as an ordered sequence:

$$\overrightarrow{Seq} \quad : \quad (T \to T) \times \mathcal{P}(T) \to T* \tag{1.100}$$
$$\overrightarrow{Seq}(f, C) \quad \stackrel{\text{def}}{=} \quad C' \tag{1.101}$$
$$\text{such that } f(C'[i]) < f(C'[j]) \qquad \text{for every } 0 \leq i < j < |C| \tag{1.102}$$

Finally, we define a sampler function for any $f$ invertible with an image having a total order and $C \subseteq Dom(f)$ such that it selects a prefix slice of length $l$ from the ordered $C$:

$$Sampler \quad : \quad (T \to T) \times \mathcal{P}(T) \times uint64 \to T+ \tag{1.103}$$
$$Sampler(f, C, l) \quad \stackrel{\text{def}}{=} \quad \overrightarrow{Seq}(f, C)[:l] \tag{1.104}$$

## 1.2 Redistribution game

**Definition 23 – Redistribution game round.**
Define $\gamma$ as a redistribution game round. $\gamma$ is conceived of as a multidimensional index:

$$\begin{aligned}
\Gamma &\stackrel{\text{def}}{=} \mathit{uint64} \times \mathit{uint64} \times \mathit{uint64} && (1.105)\\
\gamma \in \Gamma &= \langle c, \sigma, i \rangle && (1.106)\\
\textsc{chain}(\gamma) & \quad c \quad \text{ID of the blockchain context} && (1.107)\\
\textsc{series}(\gamma) & \quad \sigma \quad \text{index of the parallel series} && (1.108)\\
\textsc{round}(\gamma) & \quad i \quad \text{sequential index of the round} && (1.109)
\end{aligned}$$

Define $\textsc{block}(\gamma)$ as the starting block height of this particular game $\gamma$:

$$\textsc{block}(\gamma) \stackrel{\text{def}}{=} \textsc{round}(\gamma) \cdot \texttt{ROUND\_LENGTH} + \texttt{START\_BLOCK} \tag{1.110}$$

Ordering by sequential index defines the chain of games, which lets us define the *Prev* function:

$$\begin{aligned}
\mathit{Prev} &: \Gamma \to \Gamma && (1.111)\\
\mathit{Prev}(\langle c, \sigma, i \rangle) &\stackrel{\text{def}}{=} \langle c, \sigma, i-1 \rangle && (1.112)
\end{aligned}$$

### 1.2.1 Transactions and on-chain registers

The smart contract receives transactions from applicants in phases. The following virtual registers capture the information given in these transactions that are relevant for defining the winner:

- batches (see definition 19)
- stakes (see definition 24)
- commits (see definition 25)
- reveals (see definition 26)

**Definition 24 – Stakes .**
We define *Stakes* as the registry of stakes resulting from transactions sent to the staking contract. A record is a tuple of a node overlay, the stake balance and the committed stake

and can be updated.

$$\begin{aligned}
\mathit{Stakes} &: \quad \Gamma \to \mathit{Nodes} \times \mathit{uint64} \times \mathit{uint64} && (1.113)\\
\langle n, s, m\rangle \in \mathit{Stakes}(\gamma) &\Leftrightarrow && (1.114)\\
\textsc{right age} & \quad \exists b' < b - \texttt{MIN_STAKE_AGE}, \tau \in \text{Transactions}(b') && (1.115)\\
\textsc{node overlay} & \quad n = H(\text{origin}(\tau) \oplus \texttt{BZZ_NETWORK_ID} \oplus \text{data}(\tau)\texttt{[0]}) && (1.116)\\
\textsc{stake balance} & \quad s = \text{amount}(\tau) && (1.117)\\
\textsc{committed stake} & \quad m = \text{data}(\tau)\texttt{[1]} && (1.118)
\end{aligned}$$

There is only one stake allowed per node, so we can define the staked amount belonging to a node as the minumum of the stake balance and the committed stake times the unit price of storage:

$$\begin{aligned}
\mathit{Stake} &: \quad \Gamma \times \mathit{Nodes} \to \mathit{uint64} && (1.119)\\
\mathit{Stake}(\gamma, n) &\stackrel{\text{def}}{=} \quad \mathit{min}(s, m \cdot \mathit{Price}(\gamma)) && (1.120)
\end{aligned}$$

**Definition 25 – Commits .**
We define *Commits*$(\gamma)$ as the registry of applications for a game $\gamma$ resulting from a transaction sent to the game contract's *commit* endpoint. A tuple of the overlay of the committing node, its commitment hash and the number of the block containing the transaction is entered in the register after verifying that (i) the transaction was sent during the commit phase (right time), and (ii) that the node has enough stake and is not frozen (right amount).

$$\begin{aligned}
\mathit{Commits} &: \quad \Gamma \to \mathit{Nodes} \times \mathit{Segment} \times \mathit{Blocks} && (1.121)\\
\langle n, h, b\rangle \in \mathit{Commits}(\gamma) &\Leftrightarrow && (1.122)\\
\textsc{right time} & \quad b < \texttt{PHASE_LENGTH} \mod \texttt{ROUND_LENGTH} && (1.123)\\
\textsc{right amount} & \quad \mathit{stake}(\gamma, n) \geq \texttt{MINIMUM_STAKE} && (1.124)\\
(\textsc{redundancy}) & && (1.125)
\end{aligned}$$

**Definition 26 – Reveals.**
We define *Reveals* as the registry of reveals resulting from a transaction sent to the game contract's reveal endpoint. The reveal record is a tuple of the node overlay, the two commitment hashes, the self-reported storage depth, a serial index used for sorting, the obfuscation key, and the block number. The record is entered in the register after it is validated that (i) it was submitted during the reveal phase (right time), (ii) the commitments when obfuscated match the commit by the same node (right reveal), and (iii) that the neighbourhood selection anchor falls within the node's area of responsibility using the self-reported depth

(right location).

$$
\begin{aligned}
\mathit{RevealEntry} &: \mathit{Nodes} \times \mathit{Address}^2 \times \mathit{uint8}^2 \times \mathit{Nonce} \times \mathit{Blocks} && (1.126)\\
\mathit{Reveals} &: \Gamma \to \mathcal{P}(\mathit{RevealEntry}) && (1.127)\\
r = \langle n, chc, chs, sd, i, k, b\rangle \in \mathit{Reveals}(\gamma) &\Leftrightarrow \langle && (1.128)\\
\textsc{node}(r) &= n && (1.129)\\
\textsc{chc}(r) &= chc && (1.130)\\
\textsc{chs}(r) &= chs && (1.131)\\
\textsc{depth}(r) &= sd && (1.132)\\
\textsc{index}(r) &= i && (1.133)\\
\textsc{nonce}(r) &= k && (1.134)\\
\textsc{block}(r) &= b && (1.135)\\
& \phantom{=}\ \rangle && (1.136)\\
&\Leftrightarrow && (1.137)\\
\textsc{right time} &\quad p \leq b < 2p \mod r && (1.138)\\
&\quad p = \texttt{PHASE\_LENGTH}, r = \texttt{ROUND\_LENGTH} && (1.139)\\
\textsc{right reveal} &\quad H(n \oplus sd \oplus chc \oplus chs \oplus k) = h \text{ such that} && (1.140)\\
&\quad \langle n, h, b'\rangle \in \mathit{Commits}(\gamma) \text{ for some } b' && (1.141)\\
\textsc{right location} &\quad \mathit{PO}(\mathit{addr}(n), \mathit{NSA}(\mathit{Prev}(\gamma))) \geq sd && (1.142)\\
(\textsc{responsibility}) & && (1.143)
\end{aligned}
$$

There is only one reveal allowed per node, so we can define the reveal belonging to a node:

$$
\begin{aligned}
\mathit{Reveal} &: \Gamma \times \mathit{Nodes} \to \mathit{RevealEntry} && (1.144)\\
\mathit{Reveal}(\gamma, n) &= r && (1.145)
\end{aligned}
$$

such that

$$
\begin{aligned}
n &= \mathit{Node}(r) && (1.146)\\
r &\in \mathit{Reveals}(\gamma) && (1.147)
\end{aligned}
$$

### 1.2.2 Random nonces

**Definition 27 – Random nonces for the round.**
From the round's random seed (see definition 103 appendix **??**) we can derive all the necessary

random input nonces:

$$\begin{aligned}
\text{N.hood Selection Anchor } NSA(\gamma) &\stackrel{\text{def}}{=} H(\mathcal{R}(\gamma) \oplus BE^{64}(SK(\gamma))) && (1.148)\\
\text{Truth Selection Nonces } TSN(\gamma) &\stackrel{\text{def}}{=} H(\mathcal{R}(\gamma) \oplus BE^{8}(0)) && (1.149)\\
\text{Winner Selection Nonces } WSN(\gamma, i) &\stackrel{\text{def}}{=} H(\mathcal{R}(\gamma) \oplus BE^{8}(1)) && (1.150)\\
\text{Reserve sampling salt } RSS(\gamma) &\stackrel{\text{def}}{=} H(\mathcal{R}(\gamma)) && (1.151)\\
\text{Segment Selection Nonce } SSN(\gamma, i) &\stackrel{\text{def}}{=} H(\mathcal{R}(\gamma) \oplus BE^{8}(i)) && (1.152)
\end{aligned}$$

where

$$\begin{aligned}
SK &: \Gamma \to \mathit{uint64} && (1.153)\\
SK(\gamma) &\stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } |\mathit{Reveals}(\gamma)| > 0 \\ SK(\mathit{Prev}(\gamma)) + 1 & \text{otherwise} \end{cases} && (1.154)
\end{aligned}$$

Let us now define the witness selection function $\mathcal{W}$ that selects two random witness indexes as well as the last index of the reserve sample such that they are all distinct:

$$\begin{aligned}
\mathcal{W} &: \Gamma \times \mathit{uint64} \times \{0, 1, 2\} \to \mathit{uint8} && (1.155)\\
\mathcal{W}(\gamma, m, k) &\stackrel{\text{def}}{=} \begin{cases} SSN(\gamma, 0) \mod m-1 & \text{if } k = 0 \\ m-1 & \text{if } k = 2 \\ m-2 & \text{if } k = 1 \wedge \\ & SSN(\gamma, 0) = SSN(\gamma, 1) \mod m-1 \\ SSN(\gamma, 1) \mod m-2 & \text{otherwise} \end{cases} && (1.156)
\end{aligned}$$

### 1.2.3 Winner selection and claim validation

**Definition 28 – Prefixed hash .**
Define the hash prefixing function $\mathit{prefix}(H, p)$ as a function which when applied to a hash function $H$ and a constant byte slice $p$ outputs a hash function which for every input returns the hash of the input prefixed by $p$ using $H$.

$$\begin{aligned}
\mathit{prefix} &: (\mathit{byte}* \to \mathit{Segment}) \times \mathit{byte}* \to (\mathit{byte}* \to \mathit{Segment}) && (1.157)\\
\mathit{prefix}(H, p) &: \mathit{byte}* \to \mathit{Segment} && (1.158)\\
\mathit{prefix}(H, p)(b) &\stackrel{\text{def}}{=} H(p \oplus b) && (1.159)
\end{aligned}$$

Exceptionally, we define the prefixed version of BMT hash (denoted as $BMT[]$) as one that uses the prefixed verson of its base hash:

$$\begin{aligned}
BMT[] &: \mathit{byte}* \to \mathit{Chunk} \times \mathit{byte}\{8\} \to \mathit{Segment} && (1.160)\\
BMT[p] &\stackrel{\text{def}}{=} BMT[\mathit{prefix}(H, p)] && (1.161)
\end{aligned}$$

**Definition 29 – Transformed chunk reserve sample.**
Let us now define the chunk transformation function $\Delta^C(p)$ for a random nonce prefix $p$ as follows:

$$\begin{aligned} \Delta^C &: \quad \mathit{Nonce} \to \mathit{Chunks} \to \mathit{Segment} && (1.162)\\ \Delta^C(p) &: \quad \mathit{Chunks} \to \mathit{Segment} && (1.163)\\ \Delta^C(p)(c) &\stackrel{\text{def}}{=} \quad \mathit{BMT}[p](\textsc{data}(c), \textsc{metadata}(c)) && (1.164) \end{aligned}$$

Define the transformed reserve sample $RS^C(\gamma, n)$ for game $\gamma$ and node $n$ as the first $2^d$ chunks of the node's reserve at block $\textsc{block}(\gamma)$, using the ordering defined (see lemma 20 and definition 22) by the hash (see definition 28) of their data using BMT with 256-bit Keccak prefixed with random nonce $p$ as its base hash.

$$\begin{aligned} RS^C &: \quad \Gamma \times \mathit{Nodes} \to \mathit{Chunks}* && (1.165)\\ RS^C(\gamma, n) &\stackrel{\text{def}}{=} \quad \mathit{Sampler}(\Delta^C(p), \mathit{Reserve}(\gamma, n), 2^d) && (1.166) \end{aligned}$$

where

$$\begin{aligned} d &= \texttt{SAMPLE\_DEPTH} \text{ (see \textbf{??})} && (1.167)\\ p &= \mathit{RSS}(\gamma) \text{ (see definition 27)} && (1.168) \end{aligned}$$

**Definition 30 – Chunk reserve sample commitment hash .**
Define $CH^C(\gamma, n, )$ for game $\gamma$ and node $n$ as the BMT chunk hash of the packed address chunk (see definition 11) packing the chunks of the transformed reserve sample (see definition 29) for game $\gamma$ and node $n$.

$$\begin{aligned} CH^C &: \quad \Gamma \times \mathit{Nodes} \to \mathit{Address} && (1.169)\\ CH^C(\gamma, n) &\stackrel{\text{def}}{=} \quad \mathit{Address}(PAC^C(RS^C(\gamma, n), p)) && (1.170) \end{aligned}$$

where

$$p = \mathit{RSS}(\gamma) \text{ (see definition 27)} \qquad (1.171)$$

and where

$$\begin{aligned} PAC^C &: \quad \mathit{Chunks}* \times \mathit{Nonce} \to \mathit{Chunks} && (1.172)\\ PAC^C(C, p) &\stackrel{\text{def}}{=} \quad CAC\left(\bigoplus_{i=0}^{len(C)-1} \mathit{Seg}(C\texttt{[}i\texttt{]}, p), m\right) && (1.173) \end{aligned}$$

where

$$\begin{aligned} \mathit{Seg}(C\texttt{[}i\texttt{]}, p) &= \mathit{Address}(C\texttt{[}i\texttt{]}) \oplus \Delta^C(p)(C\texttt{[}i\texttt{]}) && (1.174)\\ m &= LE^{64}(2 \cdot 32 \cdot len(C)) && (1.175) \end{aligned}$$

**Definition 31 – Transformed slots reserve sample.**
Let us now define the slots transformation function $\Delta^S(p)$ for a random nonce $p$ as follows:

$$\Delta^S \quad : \quad \mathit{Nonce} \to \mathit{Chunks} \to \mathit{Segment} \tag{1.176}$$
$$\Delta^S(p) \quad : \quad \mathit{Chunks} \to \mathit{Segment} \tag{1.177}$$
$$\Delta^S(p)(c) \quad \stackrel{\text{def}}{=} \quad H[p](\mathit{Slot}(\mathit{Stamp}(c))) \tag{1.178}$$

Define the transformed reserve sample $RS^S(\gamma, n)$ for game $\gamma$ and node $n$ as the first $2^d$ chunks of the node's reserve at block $\textsc{block}(\gamma)$, using the ordering defined by the slot transformation function using prefix $p$:

$$RS^S \quad : \quad \Gamma \times \mathit{Nodes} \to \mathit{Chunks}* \tag{1.179}$$
$$RS^S(\gamma, n) \quad \stackrel{\text{def}}{=} \quad \mathit{Sampler}(\Delta^S(p), \mathit{Reserve}(\gamma, n), 2^d) \tag{1.180}$$

where

$$d \quad = \quad \texttt{SAMPLE\_DEPTH} \text{ (see ??)} \tag{1.181}$$
$$p \quad = \quad \mathit{RSS}(\gamma) \text{ (see definition 27)} \tag{1.182}$$

**Definition 32 – Transformed slots reserve sample commitment hash .**
Define $CH^S(\gamma, n)$ for game $\gamma$ and node $n$ as the BMT chunk hash of the packed address chunk (see definition 11) the chunks of the transformed reserve sample (see definition 29) for game $\gamma$ and node $n$.

$$CH^S \quad : \quad \Gamma \times \mathit{Nodes} \to \mathit{Address} \tag{1.183}$$
$$CH^S(\gamma, n) \quad \stackrel{\text{def}}{=} \quad \mathit{Address}(\mathit{PAC}(RS^S(\gamma, n), m)) \tag{1.184}$$

where

$$m \quad = \quad LE^{64}(32 \cdot 2^d) \tag{1.185}$$
$$d \quad = \quad \texttt{SAMPLE\_DEPTH} \text{ (see ??)} \tag{1.186}$$

**Definition 33 – Weighted selection.**
We define $\mathit{WeightedSelect}(w, k)$ as a sampler function which selects an index $0 < i < \mathit{len}(w)$ determined by the input nonce (pseudorandom number) $k$ in such a way that the indexes have a probability of being selected proportional to the weights in $w$.

$$\mathit{WeightedSelect} \quad : \quad \mathit{uint256}+ \times \mathit{uint256} \to \mathit{uint256}+ \tag{1.187}$$
$$\mathit{WeightedSelect}(w, k) \quad \stackrel{\text{def}}{=} \quad \begin{cases} i & \text{if } k < w\texttt{[}i\texttt{]} \mod W\texttt{[}i\texttt{]} \\ \mathit{WeightedSelect}(w\texttt{[}:i\texttt{]}, k) & \text{otherwise} \end{cases} \tag{1.188}$$

such that $i = len(w) - 1$, and where $W$ (cumulative weights) is defined as

$$W \quad : \quad uint256+ \rightarrow uint256 \tag{1.189}$$

$$W\,[i] \stackrel{\text{def}}{=} \begin{cases} w\,[0] & \text{if } i = 0 \\ W\,[i-1] + w\,[i] & \text{otherwise} \end{cases} \tag{1.190}$$

**Definition 34 – Truth selection.**
We determine the truth from reveals through selection weighted by stake density using the truth selection nonce as random input.

$$Truth \quad : \quad \Gamma \rightarrow RevealEntry \tag{1.191}$$

$$Truth(\gamma) \stackrel{\text{def}}{=} R(\gamma)\,[WeightedSelect(weights, TSN(\gamma))] \tag{1.192}$$

where weights are stake densities such that

$$weights\,[i] \quad = \quad Stake(\gamma, \text{NODE}(R\,[i])) \cdot 2^{\text{DEPTH}(R[i])} \tag{1.193}$$

where $R$ is the reveals of the round sorted by index:

$$R \quad = \quad \overrightarrow{Seq}(\text{INDEX}, Reveals(\gamma)) \tag{1.194}$$

**Definition 35 – Honest reveals.**
We define honest reveals as the subset of reveals for the round agreeing with the truth in reserve commitment hashes and storage depth.

$$HonestReveals \quad : \quad \Gamma \rightarrow RevealEntry* \tag{1.195}$$

$$HonestReveals(\gamma) \stackrel{\text{def}}{=} \overrightarrow{Seq}(\text{INDEX}, \{r \in Reveals(\gamma) | Honest(r)\}) \tag{1.196}$$

where

$$Honest \quad : \quad RevealEntry \rightarrow \{\mathtt{T},\mathtt{F}\} \tag{1.197}$$

$$Honest(r) \quad \leftrightarrow \tag{1.198}$$

$$\text{CHC}(r) = \text{CHC}(truth(\gamma)) \;\wedge \tag{1.199}$$

$$\text{CHS}(r) = \text{CHS}(truth(\gamma)) \;\wedge \tag{1.200}$$

$$\text{DEPTH}(r) = \text{DEPTH}(truth(\gamma)) \tag{1.201}$$

**Definition 36 – Winner selection.**
We determine the winner from honest reveals through selection weighted by stake using the winner selection nonce as random input.

$$Winner \quad : \quad \Gamma \rightarrow RevealEntry \tag{1.202}$$

$$Winner(\gamma) \stackrel{\text{def}}{=} HonestReveals(\gamma)\,[WeightedSelect(weights, WSN(\gamma))] \tag{1.203}$$

where weights are stakes such that

$$weights[i] \quad = \quad Stake(\gamma, \text{NODE}(HonestReveals[i])) \tag{1.204}$$

### 1.2.4 Proofs of reserve

**Definition 37 – Proof of reserve .**
Proof of reserve provides evidence that the reserve is replicating relevant content and shows a proof of recency of retaining chunk data in full integrity.

$$\begin{aligned}
POR &: SIP^2 \times Stamps && (1.205)\\
\Pi^{\text{R}} &: \Gamma \times Chunks \times Chunks{+} \times \{0,1,2\} \to POR && (1.206)\\
\Pi^{\text{R}}(\gamma, c, C, k) &\stackrel{\text{def}}{=} \langle && (1.207)\\
\text{WITNESS PROOF} &\quad \Pi^{\text{SIP}}(c, d \cdot i), && (1.208)\\
\text{RETENTION PROOF} &\quad \Pi^{\text{SIP}}(C[i], j), && (1.209)\\
\text{POSTAGE STAMP} &\quad Stamp(C[i]) && (1.210)\\
&\quad \rangle && (1.211)
\end{aligned}$$

where

$$\begin{aligned}
i &= \mathcal{W}(\gamma, len(C), k) && (1.212)\\
j &= SSN(\gamma, k) \mod 128 && (1.213)\\
d &= \frac{SegCnt(c)}{len(C)} \left( = \begin{cases} 1 & \text{if } C = RS^S \\ 2 & \text{if } C = RS^C \end{cases} \right) && (1.214)
\end{aligned}$$

**Definition 38 – Proof of reserve validation.**

$$\begin{aligned}
\mathbb{V}^{\text{R}} &: \Gamma \times Nodes \times Address \times \{0,1,2\} \times POR \to \{\mathtt{T},\mathtt{F}\} && (1.215)\\
\mathbb{V}^{\text{R}}(\gamma, n, ch, k, \pi) &\leftrightarrow && (1.216)\\
\text{RELEVANCE} &\quad \mathbb{V}^{\text{SIP}}(ch, d \cdot i, p_w) \wedge && (1.217)\\
&\quad \mathbb{V}^{\text{STAMP}}(ps, \gamma, n) \wedge && (1.218)\\
\text{RETENTION} &\quad \text{DATA}(p_w) = a \wedge && (1.219)\\
&\quad \mathbb{V}^{\text{SIP}}(a, j, p_r) \wedge && (1.220)\\
\text{RECENCY} &\quad i = \mathcal{W}(\gamma, SegCnt(p_w), k) \wedge && (1.221)\\
&\quad j = SSN(\gamma, k) \mod 128 \wedge && (1.222)\\
\text{RETRIEVABILITY} &\quad PO(a, n) \geq sd && (1.223)
\end{aligned}$$

where

$$
\begin{aligned}
a &= \textsc{Address}(ps) && (1.224)\\
\pi &= \langle p_w, p_r, ps\rangle && (1.225)\\
sd &= \textsc{depth}(\mathit{Reveal}(\gamma, n)) && (1.226)\\
d &= \frac{\mathit{SegCnt}(p_w)}{2^D}\left(= \begin{cases}1 & \text{if } C = RS^S\\ 2 & \text{if } C = RS^C\end{cases}\right) && (1.227)\\
D &= \texttt{SAMPLE\_DEPTH} \text{ (see ??)} && (1.228)
\end{aligned}
$$

**Definition 39 – Proof of chunk density validation.**
Define $\mathbb{V}^{\text{CD}}(\gamma, p_0, p_1, p_2)$ as the validation function for the proof of chunk density for round $\gamma$ and proof of reserve and segment inclusion proof pairs $p_0, p_1, p_2$.

$$
\begin{aligned}
\mathit{PORT} &: \mathit{POR} \times \mathit{SIP} && (1.229)\\
\mathbb{V}^{\text{CD}} &: \Gamma \times \mathit{PORT}^3 \to \{\texttt{T},\texttt{F}\} && (1.230)\\
\mathbb{V}^{\text{CD}}(\gamma, \pi_0, \pi_1, \pi_2) &\leftrightarrow && (1.231)\\
\textsc{right data segment} &\quad \textsc{data}(pr_k) = \textsc{data}(pt_k) \text{ for } k \in \{0,1,2\} \wedge && (1.232)\\
\textsc{right address} &\quad \textsc{sister}(pw_k, 0) = ta_k \text{ for } k \in \{0,1,2\} \wedge && (1.233)\\
\textsc{right order} &\quad ta_0 < ta_1 < ta_2 \wedge && (1.234)\\
\textsc{right size} &\quad ta_2 \le \texttt{MAX\_SAMPLE\_VALUE} && (1.235)
\end{aligned}
$$

where for $k \in \{0, 1, 2\}$

$$
\begin{aligned}
ta_k &= H_\Pi(p, j_k, pt_k) && (1.236)\\
j_k &= \mathit{SSN}(\gamma, k) \mod 128 && (1.237)\\
\pi_k &= \langle\langle pw_k, pr_k, \_\rangle, pt_k\rangle && (1.238)
\end{aligned}
$$

and where

$$
p = \mathit{RSS}(\mathit{Prev}(\gamma)) \text{ (see definition 27)} \quad (1.239)
$$

**Definition 40 – Proof of stamp density validation.**
Define $\mathbb{V}^{\text{SD}}(\gamma, ps_0, ps_1, ps_2)$ as the validation function for the proof of stamp density for round $\gamma$ and postage stamps $ps_0, ps_1, ps_2$.

$$
\begin{aligned}
\mathbb{V}^{\text{SD}} &: \Gamma \times \mathit{Stamps}^3 \to \{\texttt{T},\texttt{F}\} && (1.240)\\
\mathbb{V}^{\text{SD}}(\gamma, ps_0, ps_1, ps_2) &\leftrightarrow && (1.241)\\
\textsc{right order} &\quad ta_0 < ta_1 < ta_2 \wedge && (1.242)\\
\textsc{right size} &\quad ta_2 \le \texttt{MAX\_SAMPLE\_VALUE} && (1.243)
\end{aligned}
$$

where for $k \in \{0, 1, 2\}$

$$ta_k \quad = \quad H(Slot(ps_k) \oplus p) \tag{1.244}$$

and where

$$p \quad = \quad RSS(Prev(\gamma)) \text{ (see definition 27)} \tag{1.245}$$

**Definition 41 – Proof of entitlement.**
Proof of entitlement captures all the evidence a node needs to submit with their claim transaction to valildate.

$$\begin{aligned}
POE &: PORT^3 \times POR^3 && (1.246)\\
\Pi^{\text{ENT}} &: \Gamma \times Nodes \to POE && (1.247)\\
\Pi^{\text{ENT}}(\gamma, n) &\stackrel{\text{def}}{=} \langle && (1.248)\\
&\langle \Pi^{\text{POR}}(\gamma, n, PAC^C(crs, p), crs, 0), pt_0\rangle, && (1.249)\\
&\langle \Pi^{\text{POR}}(\gamma, n, PAC^C(crs, p), crs, 1), pt_1\rangle, && (1.250)\\
&\langle \Pi^{\text{POR}}(\gamma, n, PAC^C(crs, p), crs, 2), pt_2\rangle, && (1.251)\\
&\Pi^{\text{POR}}(\gamma, n, PAC(srs), srs, 0), && (1.252)\\
&\Pi^{\text{POR}}(\gamma, n, PAC(srs), srs, 1), && (1.253)\\
&\Pi^{\text{POR}}(\gamma, n, PAC(srs), srs, 2), && (1.254)\\
&\rangle && (1.255)
\end{aligned}$$

where for $k \in \{0, 1, 2\}$

$$\begin{aligned}
pt_k &= \Pi^{\text{SIP}}_{prefix}(p, r\texttt{[}i_k\texttt{]}, j_k) && (1.256)\\
j_k &= SSN(\gamma, k) \mod 128 && (1.257)\\
i_k &= \mathcal{W}(\gamma, length(crs), k) && (1.258)
\end{aligned}$$

and where

$$\begin{aligned}
crs &= RS^C(\gamma, n) && (1.259)\\
srs &= RS^S(\gamma, n) && (1.260)\\
p &= RSS(Prev(\gamma)) \text{ (see definition 27)} && (1.261)
\end{aligned}$$

**Definition 42 – Winner's claim validation.**
Define $\mathbb{V}^{\text{POE}}(\gamma, p)$ as the validation function for the proof of entitlement $p$ as part of the

winning claim for game $\gamma$:

$$
\begin{aligned}
\mathbb{V}^{\text{POE}} &: \quad \Gamma \times POE \rightarrow \{\mathtt{T},\mathtt{F}\} && (1.262)\\
\mathbb{V}^{\text{POE}}(\gamma, p) &\Leftrightarrow && (1.263)\\
\textsc{reserve:} & && (1.264)\\
\textsc{chunks} & \quad \forall k \in \{0,1,2\}, \mathbb{V}^{\text{R}}(\gamma, n, \textsc{chc}(r), k, \pi_k) \wedge && (1.265)\\
\textsc{stamps} & \quad \forall k \in \{0,1,2\}, \mathbb{V}^{\text{R}}(\gamma, n, \textsc{chs}(r), k, \phi_k) \wedge && (1.266)\\
\textsc{reserve size:} & && (1.267)\\
\textsc{chunk density} & \quad \mathbb{V}^{\text{CD}}(\gamma, \langle \pi_0, pt_0 \rangle, \langle \pi_1, pt_1 \rangle, \langle \pi_2, pt_2 \rangle) \wedge && (1.268)\\
\textsc{stamp density} & \quad \mathbb{V}^{\text{SD}}(\gamma, \textsc{ps}(\phi_0), \textsc{ps}(\phi_1), \textsc{ps}(\phi_2)) && (1.269)
\end{aligned}
$$

where

$$
\begin{aligned}
n &= \textsc{node}(r) && (1.270)\\
r &= Winner(\gamma) && (1.271)\\
p &= \langle \langle \pi_0, pt_0 \rangle, \langle \pi_1, pt_1 \rangle, \langle \pi_2, pt_2 \rangle, \phi_0, \phi_1, \phi_2 \rangle && (1.272)
\end{aligned}
$$

**Corollary 43 – Outpayment scheme is fair.**
Rewarding the pot to randomly selected neighbourhood implements a redistribution scheme that is fair across neighbourhoods.

*Proof.* With the consensus mechanism we can show that the Nash-optimal strategy of nodes is to follow the protocol and consent on the reserve. On the other hand, the optimal strategy for uploaders is to uniformly distribute chunks across the name space. As a consequence, nodes are expected to have identical storage depth and its variance is independent of being chosen. Long term then relative cumulative outpayments by the redistribution game converge to the fair share.

Secondly, we argue that the mode of selecting the winner is fair within neighbourhoods.

*Proof.*

# Chapter 2

# Data types and algorithms

## 2.1 Built-in primitives

### 2.1.1 Crypto

This section describes the crypto primitives used throughout the specification. They are exposed as buzz built-in functions. The modules are hashing, random number generation, key derivation, symmetric and asymmetric encryption (ECIES), mining (i.e., finding a nonce), elliptic curve key generation, digital signature (ECDSA), Diffie–Hellman shared secret (ECDH) and Reed-Solomon (RS) erasure coding.

Some of the built-in crypto primitives (notably, sha3 hash, and ECDSA ecrecover) are replicating crypto functionality of the Ethereum VM. These are defined here with the help of ethereum api calls to a smart contract. This smart contract just implements the primitives of "buzz" and only has read methods.

#### Hashing

The base hash function implements Keccak256 hash as used in Ethereum.

**Definition 44 – Hashing.**

```
   // /crypto

define function hash @input []byte
    ?and/with @suff
    return segment
as
```

```
    ethereum/call "sha3" with @input append= @suff
         on context contracts "buzz"
```

### Random number generation

**Definition 45 – Random number generation.**

```
   // /crypto

define function random type
    return [@type size]byte
```

### Scrypt key derivation

The crypto key derivation function implements `scrypt`

**Definition 46 – Scrypt key derivation.**

```
   // /crypto

define type salt as [segment size]byte

define type key as [segment size]byte

// params for scrypt key derivation function
// scrypt.key(password, salt, n, r, p, 32) to generate key

define type kdf
    n int // 262144
    r int // 8
    p int // 1

define function scrypt from @password
    with   salt
    using  kdf
    return key
```

### Mining helper

This module provides a very simple helper function that finds a nonce that when given as the single argument to a mining function returns true.

**Definition 47 – Mining a nonce.**

```
// /crypto

define type nonce as [segment size]byte

define function mine @f function of nonce return bool
as
    @nonce = random key
    return @nonce if call @f @nonce
    self @f
```

### Symmetric encryption

Symmetric encryption uses a modified blockcipher with 32 byte blocksize in counter mode. The segment keys are generated by hashing the chunk-specific encryption key with the counter and hash that again. This second step is required so that a segment can be selectively disclosed in a 3rd party provable way yet without compromising the security of the rest of the chunk.

The module provides input length preserving blockcipher encryption.

**Definition 48 – Blockcipher.**

```
// /crypto

// two-way (en/de)crypt function for segment
define function crypt.segment segment
    with key
    at @i uint8
as
    hash @key and @i        // counter mode
        hash                // extra hashing
        to @segment length  // chop if needed
        xor @segment        // xor key with segment

// two-way (en/de)crypt function for arbitrary length
define function crypt @input []byte
```

```
    with key
    return [@input length]byte
as
    @segments = @input each segment size    // iterate segments of input
        go crypt.segment at @i++ with @key  // concurrent crypt on segments
    return wait for @segments               // wait for results
        join                                // join (en/de) crypted segments
```

## Elliptic curve keys

Public key cryptography is the same as in Ethereum, it uses the secp256k1 elliptic curve.

**Definition 49 – Elliptic curve key generation.**

```
  // /crypto
define type pubkey  as [64]byte
define type keypair
    privkey [32]byte
    pubkey

define type address as [20]byte

define function address pubkey
    return address
as
    hash pubkey
        from 12

define function generate
    ?using entropy
as
    @entropy = random segment if no @entropy
    http/get "signer/generate?entropy=" append @entropy
        as keypair
```

### Asymmetric encryption

Asymmetric encryption implements ECIES based on the secp256k1 elliptic curve.

**Definition 50 – Asymmetric encryption.**

```
// /crypto

define function encrypt @input []byte
    for pubkey
    return [@input length]byte

define function decrypt @input []byte
    with keypair
    return [@input length]byte
```

### Signature

Crypto's built-in signature module implements secp2156k1 elliptic curve based ECDSA. The actual signing happens in the external signer running as a separate process (possibly within the secure enclave). As customary in Ethereum, the signature is represented and serialised using the r/s/v format,

**Definition 51 – Signature.**

```
// /crypto

define type signature
    r segment
    s segment
    v uint8
    signer private keypair


define type doc
    preamble []byte
    context  []byte
    asset    segment

define function sign @input []byte
    by keypair
    return signature
as
```

```
    @doc = doc{ "swarm signature", context caller, @input }
    @sig = http/get "signer/sign?text=" append @doc
        append "&account=" append @keypair pubkey address
            as signature
    @sig signer = @keypair
    @sig

define function recover signature
    with @input []byte
    from @caller []byte
    return pubkey
as
    @doc =  doc{ "swarm signature", @caller, @input } as bytes
    ethereum/call "ecrecover" with
        on context contracts "buzz"
            as pubkey
```

### Diffie-Hellmann shared secret

The shared secret module implements elliptic curve based Diffie–Helmann shared secret (ECDH) using the usual secp256k1 elliptic curve. The actual DH comes from the external signer which is then hashed together with a salt.

**Definition 52 – Shared secret.**

```
   // /crypto

define function shared.secret between keypair
    and pubkey
    using salt
    return [segment size]byte
as
    http/get "signer/dh?pubkey=" append @pubkey append "&
       account=" @keypair address
        hash with @salt
```

### Erasure coding

Erasure coding interface provides wrappers `extend/repair` for the encoder/decoder that work directly on a list of chunks.[1]

Assuming $n$ out of $m$ coding. `extend` takes a list of $n$ data chunks and an argument for the number of required parities. It returns the parity chunks only. `repair` takes a list of $m$ chunks (extended with *all* parities) and an argument for the number of parities $p = m - n$, that designate the last $p$ chunks as parity chunks. It returns the list of $n$ repaired data chunks only. The encoder does not know which parts are invalid, so missing or invalid chunks should be set to `nil` in the argument to repair. If parity chunks are needed to be repaired, you call `repair @chunks with @parities; extend with @parities`

**Definition 53 – CRS erasure code interface definition.**

```
  // /crypto/crs

define function extend @chunks []chunk
    with @parities uint
    return [@parities]chunk

define function repair @chunks []chunk
    with @parities uint
   return [@chunks length - @parities]chunk
```

**Definition 54 – CRS erasure coding parameters.**

```
  // /crypto/crs
define strategy as "race"|"fallback"|"disabled"

define type params
    parities uint
    strategy
```

## 2.1.2 State store

**Definition 55 – State store.**

```
  // /statestore

define type key []byte
```

[1] Cauchy-Reed-Solomon erasure codes based on `https://github.com/klauspost/reedsolomon`.

```
define type db  []byte
define type value []byte

define function create db

define function destroy db

define function put value
    to db
    on key

define function get key
    from db
    return value
```

### 2.1.3 Local context and configuration

**Definition 56 – Context.**

```
  // /context

define type contract as "buzz"|"chequebook"|"postage"|""

define type context
    contracts [contract]ethereum/address
```

## 2.2 Bzz address

### 2.2.1 Overlay address

Swarm's overlay network uses 32-byte addresses. In order to help uniform utilisation of the address space, these addresses must be derived using a hash function. A Swarm node must be associated a Swarm base account or bzz account, which is an Ethereum account that the node (operator) must possess the private key for. The node's overlay address is derived the public key of this account.

**Definition 57 – Swarm overlay address of node A.**

$$\mathit{overlayAddress}(A) \stackrel{\text{def}}{=} \mathit{Hash}(\mathit{ethAddress}|\mathit{bzzNetworkID}) \tag{2.1}$$

where

- *Hash* is the 256-bit Keccak SHA3 hash function
- *ethAddress* - the ethereum address (bytes, not hex) derived from the node's base account public key: $\mathit{account} \stackrel{\text{def}}{=} \mathit{PubKey}(K_A^{bzz})[12:32])$, where
  - *PubKey* is the *uncompressed* form of the public key of a keypair *including* its 04 (uncompressed) prefix.
  - $K_A^{bzz}$ refers to the node's bzz account key pair
- *bzzNetworkID* is the bzz network id of the swarm network serialised as a little-endian binary *uint64*.

In a way, deriving the node address from a public key means the overlay address space is not freely available: to occupy an address you must possess the private key of the address which other nodes need to verify. Such authentication is easy using a digital signature of a shared consensual piece of text, see 2.2.

### 2.2.2 Underlay address

To enable peers to locate the a node on the network, the overlay address is paired with an underlay address. The underlay address is a string representation of the node's network location on the underlying transport layer. It is used by nodes to dial other nodes to establish keep-alive peer to peer connections.

### 2.2.3 BZZ address

Bzz address is functionally the pairing of overlay and underlay addresses. In order to ensure that an overlay address is derived from an account the node possesses as well as verifiably attest to an underlay address a node can be called on, bzz addresses are communicated in the following transfer format:

**Definition 59 – Swarm bzz address transfer format.**

```
// ID: /swarm/handshake/1.0.0/bzzaddress

syntax = "proto3";
```

```
message BzzAddress {
    bytes Underlay = 1;
    Signature Sig  = 2;
    bytes Overlay  = 3;
}
```

Here the signature is attesting to the association of an overlay and an underlay address for a network.

**Definition 60 – Signed underlay address of node A.**

$$signedUnderlay(A) \stackrel{\text{def}}{=} Sign(underlay|overlay|bzzNetworkID) \tag{2.2}$$

of the underlay with overlay and bzz network ID appended as plaintext and hashes the resulting public key together with the bzz network ID.

**Definition 61 – Node addresses: overlay, underlay, bzz address.**

```
// /bzz

define type overlay as [segment size]byte
define type underlay []byte

define function overlay.address of pubkey
    within @network uint64
as
    hash @pubkey address and @network
        as overlay

define function valid bzz.address
    within @network uint64
as
    assert @bzz.address overlay == overlay.address of crypto/
       recover from @bzz.address signature with @bzz.address
       @underlay
        within @network

define function bzz.address of overlay
    from underlay
```

```
    by @account ethereum/address
as
    @sig = crypto/sign @underlay by @account
    bzz.address{ @overlay, @underlay, @sig }
```

In order to get the overlay address from the transfer format peer info, one recovers from signature the peer's base account public key using the plaintext that is constructed as per 60. From the public key, the overlay can be calculated as in 3. The overlay address thus obtained needs to be checked against the one provided in the handshake.

Signing of the underlay enables preflight authentication of the underlay of a trusted but not connected node.

Since underlays are meant to be volatile, we can assume and in fact expect multiple underlays signed by the same node. However, these are meant to be temporally ordered. So one with a newer timestamp invalidates the older one.

In order to make sure that the node connected through that underlay does indeed operate the overlay address, its authentication must be obtained through the peer connection that was initiated by dialing the underlay. The This protects against malicious impersonation of a trusted overlay potentially.

## 2.3 Chunks, encryption and addressing

### 2.3.1 Content addressed chunks

First let us define some basic types, such as *payload, span, segment*. These fixed length byte slices enables verbose expression of fundamental units like `segment size` or `payload size`.
**Definition 62 – segment, payload, span, branches.**

```
    // /chunk

define type segment as [32]byte     // unit for type definitions
define type payload as [:4096]byte // variable length max 4 Kilobyte
define type span as uint64          // little endian binary marshalled

define function branches
```

```
    as payload size / segment size
```

Now let's turn to the definition of *address*, *key* and *reference*:

**Definition 63 – Chunk reference.**

```
   // /chunk

define type address as [segment size]byte

define type reference
    address                    // result of bmt hash
    key if context.encryption // decryption key optional (
       context dependent)
```

Now, define chunk as a object with span and payload.

**Definition 64 – Content addressed chunk.**

```
   // /chunk

define type chunk
    span      // length of data span subsumed under node
    payload   // max 4096 bytes

define function address of chunk
as
    @chunk payload bmt/hash with @chunk span

define function create from payload
    ?over span
as
    @span = @payload length if no @span
    @chunk = chunk{ @span, @payload }
    return @chunk if no context encryption
    @key = encryption.key for @chunk
    @chunk encrypt with @key
```

Where length is the content of the length field and reference size is the sum of size of the referencing hash value and that of the decryption key, which is currently 64, as we use 256-bit hashes and 256-bit keys.

In order to remove the padding after decryption before returning the plaintext chunk.
**Definition 65 – Span to payload length.**

```
   // /chunk

define function payload.length of span
as
    while @span >= 4096
        @span = @span + 4095
           / 4096
           * reference size
    return @span
```

Finally, we can define the public API of chunks for retrieval and storage.
**Definition 66 – Chunk API retrieval.**

```
   // /chunk

define function retrieve reference
has api GET on "chunk/<reference>"
as
    retrieve @reference address as chunk
        (decrypt with @reference key if @reference key)
```

**Definition 67 – Chunk API: storage.**

```
   // /chunk


define function store payload
    ?over span
has api POST on "chunk/(?span=<span>)"
    from payload as body
as
    @chunk = create from @payload over @span
    reference{ @chunk address, @chunk key }
```

### 2.3.2 Single owner chunk

Single owner chunks are the second type of chunk in swarm. They constitute the basis of swarm feeds.

**Definition 68 – Single owner chunks.**

```
  // /soc

// data structure for single owner chunk
define type soc
    id          [segment size]byte // id 'within' owner namespace
    signature   crypto/signature    // owner attests to <content, id>
    chunk                           // content: embeds a content chunk

// constructor for single owner chunks
define function create from chunk
    by @owner crypto/keypair
    on @id [segment size]byte
 as
    @sig = crypto/sign @id and @chunk address by @owner
    soc{ @id, @sig, @chunk }

define function address soc
as
    hash @soc id and @soc signature signer address
```

**Definition 69 – Single owner chunk API: retrieval.**

```
  // /soc

define function retrieve @id [segment size]byte
    by @owner ethereum/address
    ?with key
has api GET on "soc/<owner>/<id>(?key=<key>)"
as
    retrieve hash @id and @owner
        as soc
            chunk (decrypt with @key if @key)
```

**Definition 70 – Single owner chunk API: storage.**

```
// /soc

define function store payload
    on @id [segment size]byte
    by @owner ethereum/address
    ?over span
has api POST on "soc/<owner>/<id>?span=<span>&encrypt=<encrypt>
   "
    from <payload> as body
as
    @span = @payload length if no @span
    @chunk = chunk{ @span, @payload }
    if context encryption then
        @key = encryption.key for @chunk
        @chunk encrypt= with @key
    @soc = create from @chunk on @id by private key of @owner
    reference{ @soc store address, @key }
```

### 2.3.3 Binary Merkle Tree Hash

The hashing method used to obtain the address of the default content addressed chunk is called the binary Merkle tree hash, or BMT hash for short.

#### Calculating the BMT hash

The base segments of the binary tree are subsequences of the chunk content data. The size of segments is 32 bytes, which is the digest size of the *base hash* used to construct the tree. Given the Swarm hash tree used to represent files (see 2.5.1) assumes that intermediate chunks package references to other chunks.

Obtaining the BMT hash of a sequence involves the following steps:

1. *padding* - If the content is shorter than the maximum chunk Size (4096 bytes), it is padded with zeros up to chunk size. Note that this zero padding is only for hashing and does not impact chunk data sizes.
2. *chunk data layer* - Calculate the base hash of *pairs of segments* in the padded chunk, i.e., segment size $(2 * 32)$ units of data and concatenate the results.

3. *building the tree* - Repeat previous step on the result until the result is just one section.
4. *calculate span* - Calculate the span of the data, i.e., the size of the data that is subsumed under the chunk represented by the unpadded data as a 64-bit little-endian integer value (see 2.5.1).
5. *integrity protection* - Prepend the span to the root hash of the binary tree and calculate the *base hash* of the data.

**Definition 71 – BMT hash.**

```
  // /bmt

define function hash payload
    with span
as
    @padded = @payload as [:chunk size]byte     // use zero
       padding
    // for BMT hashing only
    hash @span and root of @padded over chunk size

define function root of @section []byte
    over @len uint
as
    return hash @section          // data level
        if @len == 2 * segment size
    @len /= 2                     // recursive call
    @children = @section each @len go self over @len
    wait for @children
        join hash
```

### Inclusion proofs

Having the segments align with the hashes packaged in these chunks one can extend the notion of inclusion proofs to files. The BMT hash enables compact 3rd party verifiable segment inclusion proofs.

## 2.3.4 Encryption

Symmetric encryption in Swarm is using a slightly modified version blockcipher in counter mode.

The encryption seed for the chunk is derived from the master seed if given, otherwise just generated randomly.

The reference to a single chunk (and the whole content) is the concatenation of the hash of encrypted data and the decryption key (see 63). This means the encrypted Swarm reference (64 bytes) will be longer than the unencrypted one (32 bytes). When a node syncs encrypted chunks, it does not share the full references (or the decryption keys) with the other nodes in any way. Therefore, other nodes will be unable to access the original data, or in fact, even to detect whether a chunk is encrypted.

**Definition 72 – Chunk encryption/decryption API.**

```
// generate key for a chunk
define encryption.key for chunk
    ?with @seed [segment.size]byte
as
    return crypto/random key if no @seed // generate new
    hash @seed and @chunk address

define function encrypt chunk
    with key
as
    @segments = @chunk data pad to chunk size
        each segment size
            go crypt at @i++ with @key
    @span = chunk span crypt at branches with @seed
    @payload = wait for @segments
        join
    chunk{ @span, @payload }


define function decrypt chunk
    with key
as
    @span  = @chunk span
        crypt at branches with @key
    @segments = chunk data to @span payload.length
        each segment size
            go crypt at @i++ with @key
    @payload = wait for @segments
        join
    chunk{ @span, @payload }
```

Encrypted Swarm chunks are not different from plaintext chunks and therefore no change is needed on the P2P protocol level to accommodate them. The proposed encryption scheme is end-to-end, meaning that encryption and decryption is done on endpoints, i.e., where the http proxy layer runs. This has an important consequence that public gateways cannot be used for encrypted content. On the other hand, the apiary modular design allows for client side encryption on top of external APIs while proxying all other calls via the gateway.

## 2.4 Postage stamps

### 2.4.1 Witness type

There can be different implementations of postage stamps that differ in the structure and semantics of the *proof of payment*. To allow for new cryptographic mechanisms to be used as they are developed, the `witnessType` argument indicates the type of the witness used.

Witness type 0 stands for ECDSA witness, which is an ECDSA signature on the byte slice resulting from the concatenation of 1) preamble constant 2) chunk hash 3) batch reference 4) valid until date.[2] This is the bare minimum that postage stamp contracts and clients must implement.[3]

Witness type 1 refers to the RSA witness, which is an RSA signature on the same 128 bytes as above. The binary encoding of the RSA signature is of variable length, and is an Solidity ABI encoded array of the RSA signature $s$.[4]

The RSA witness is specified so that blind stamping services can be implemented in a simple fashion, in order to mitigate the privacy issues arising from the ability to link chunks signed with the same private key. Even though blind ECDSA signatures also exist, their protocol requires more rounds of communication, making the implementation of such a service more complex, more error-prone and less performant.

The inclusion of the entire public key in each RSA witness rather than storing the public key in contract state and just referencing it from the witness is justified by reducing the gas costs

[2] The binary encoding of the ECDSA signature is 65 bytes resulting from the concatenation of the $r$ (32 bytes), $s$ (32 bytes) and $v$ (1 byte) parameters of the signature, in this order. The signature is calculated on the secp256k1 elliptic curve, just like the signatures of Ethereum transactions.

[3] The ECDSA witness is the simplest and cheapest solution both in terms of gas consumed by the stamp verification contract and in terms of computational resources used off chain. Also, it does not rely on cryptographic assumptions in addition to those on which Ethereum critically relies, therefore as long as Ethereum is considered cryptographically secure, no advance in cryptorgraphy can render this witness type insecure. This is the justification for this witness type to be the only mandatory witness type to be implemented.

[4] as defined in *PKCS #1*, `https://tools.ietf.org/html/rfc8017` and the RSA public key parameters $n$ (RSA modulus) and $e$ (public exponent).

of interactions with the contract as well as future-proofing the design in case contract state rent is introduced in Ethereum. These considerations are more important than the brevity of postage stamps, marginally reducing the bandwith costs of uploading and forwarding stamped content.

Note that cryptographic advances can render RSA witnesses insecure without rendering Ethereum insecure, therefore RSA witnesses can be phased out in future versions of the protocol, if the security of RSA signatures gets compromised. Note, furthermore, that such blind signing services are not entirely trustless, through the damage they can incur is bounded. Trustless blind stamping services based on ZK proofs are not feasible at this stage, as the current algorithms are not sufficiently performant for the purpose, but given the rapid advances in the field, the development of suitable algorithms can be expected in the future, in which case a corresponding witness type will have to be specified in a separate SWIP.

### 2.4.2 Contract Upgrades

In order to facilitate the upgrade of the contract either in case of a discovered vulnerability or some feature extension (such as adding new witness types), it is recommended that the part holding the funds with the database of payments and the part that verifies witnesses are in separate contracts so that a backwards-compatible upgrade can be performed with minimal disruption.

In order to avoid centralized control, it is also recommended that it is the witness-verifying contract that is referenced in client configuration so that client operators can independently decide for themselves when and whether to switch to a new contract, as they become available.

Nodes participating in the same postage system are configured to reference the same contract on the same blockchain. This contract must conform to the following interface:

**Definition 73 – Postage contract.**

This accessor method returns `true` if the proof embodied by `witness` checks out for all other arguments within the claimed validity period, i.e. when `block.timestamp` (the output of `TIMESTAMP EVM` opcode) is between `beginValidity` (inclusive) and `endValidity` (exclusive). Outside of the validity period, the return value is `undefined`.

**Definition 74 – Postage stamp basic types: batchID, address, witness, stamp, validity.**

```
// /postage

define type batchid as [segment size]byte
```

```
define type address as bzz/address
define type witness as crypo/signature

// postage stamp
define type stamp
    batchid
    address
    witness

define function valid stamp
as
    // check validity on blockchain
    ethereum call "valid" using context contracts "postage"
        with @stamp
```

## 2.5 Files, manifests and data structures

### 2.5.1 Files and the Swarm hash

This table gives an overview of data sizes a chunk span represents, depending on the level of recursion.

| | span | | | | | |
|---|---|---|---|---|---|---|
| | unencrypted | | | encrypted | | |
| level | chunks | $log_2$ of bytes | standard | chunks | $log_2$ of bytes | standard |
| 0 | 1 | 12 | 4KB | 1 | 12 | 4KB |
| 1 | 128 | 19 | 512KB | 64 | 18 | 256KB |
| 2 | 16,384 | 26 | 67MB | 4,096 | 24 | 16MB |
| 3 | 2,097,152 | 33 | 8.5GB | 262,144 | 30 | 1.07GB |
| 4 | 268.44M | 40 | 1.1TB | 16.78M | 36 | 68.7GB |
| 5 | 34,359.74M | 47 | 140TB | 1,073.74M | 42 | 4.4TB |

**Table 2.1:** Size of chunk spans

#### Calculating the Swarm Hash

Client-side custom redundancy is achieved by RS erasure coding; using it neccessitates some RS parameters.

**Definition 75 – File API: upload/storage; swarm hash.**

```
   // /file

define function encode @levels []chunk stream
    for @level uint
as
    @chunks = @levels at @level     // read chunk stream
    @crs = context crs              //
    @m = branches (- @crs parities if @crs)

    @parent = read @m from @chunks  // read up to m chunks from stream blocking
        (append crs/extend with @crs parities if @crs)
            each chunk/store        // package children reference
                join as chunk

    if @levels length == @level+1 then
        @levels append= stream{}
        go self @levels for @level+1

    write @parent to @levels at @level+1
    if no @chunks then
        close @levels at @level + 1
    else
        self @chunks for @level

define function split @data byte stream
as
    @level = chunk stream{}
    go @data each chunk size as chunk
        write to @chunks
    @level

define function upload byte stream as @data
has api POST  on "file/" from @data as body
as
    @levels append= @data split     //
    go encode @levels for 0
    @top = @levels each wait for  // wait for all levels to close
    return @top at 0              // return root hash as
```

```
address
```

**Definition 76 – File API: download/retrieval.**

```
// /file

define function copy to @reader stream of byte{}
    from @chunks stream of []chunk
    using @buffers stream of [@buffer.size]stream of [branches] chunk
as
    @chunks = read @buffers if no @chunks
    @chunk = read @chunks
    if no @chunk
        write @chunks to @buffer
        self @reader using @buffers
    write @chunk data to @reader

define function download reference
    ?using @buffer.size uint64
has api GET  on "file/<reference>"
as
    @reader = stream of byte{}
    @buffers = stream of [@buffer.size]stream of [branches] chunk
    chunk/retrieve @reference            // root chunk retrieval
        go decode into @buffer down to 1 // traverse
    copy into @reader from @buffers

define function decode chunk
    into @response chunk stream
    ?down to @limit uint8
as

    @crs = context crs
    @all = @m = branches
    if @crs then
        @m -= @crs parities
        if @crs strategy is not "race" then
            @all = @m

```

```
    @chunks = @chunk segments up to @all
        each go as reference retrieve
    wait for @m in @chunks

    if @crs then
        cancel @chunks
        @chunks = @chunks crs/repair with @crs parities

    if @chunk span < chunk size exp @limit + 1 then
        @chunks each  into @reader


    @chunks each go self down to @limit
```

**Definition 77 – File info.**

```
   // /file

define type info
    mode        int64
    size        int64
    modified    time
```

### 2.5.2 Manifests

Manifests represent a mapping of strings to references. The primary purpose is to implement document collections (websites) on swarm and enable URL-based addressing of content. This section defines the data structures relevant for manifests as well as the algorithms for lookup and update which implement the manifest API (see **??**).

A manifest entry can be conceived of as metadata about a file pointed to and retrievable by its reference (see 2.5.1). The metadata is quite diverse, ranging from information needed for access control, file information similar to one given on file systems, information needed for erasure coding (see **??** and 2.1.1), information for browser, i.e., response headers such as content type (MIME info) and most importantly the reference to the file. Using manifests as simple key-value store is exemplified by access control (see 2.6 and **??** for the specification).
**Definition 78 – Manifest entry.**

```
   // /manifest

```

```
// manifest entry encodes attributes
define type entry
    file/info              // FS file/dir info
    access/params          // access control params
    crs/params             // erasure coding - CRS params
    reference              // reference
    headers                // http response headers

define type headers
    content.type  [segment size]byte
```

**Definition 79 – Manifest data structure.**

```
// /manifest

define type node
    entry   *entry         // reference to chunk serialised as
       entry
    forks   [<<256]fork    // sparse array of max 256 fork

// fork encodes a branch
define type fork
    prefix segment   // compaction
    node   *node     // reference to chunk serialised as node
```

**Definition 80 – Manifest API: path lookup.**

```
// /manifest

define function lookup @path []byte
    in *node
has api GET on "/manifest/<@node:reference>/<@path>"
as
    context access = @node entry access/params
    // manifest is  a compacted trie
    @fork = @node forks at head @path
    // if @path empty, the  paths matched return the entry
    if no @path then
        return @node entry

    if @fork prefix is prefix of @path then // including ==
      return self @path from @fork prefix length
```

```
        in @fork node
    fail with "not found"
```

**Definition 81 – Manifest API: update.**

```
   // /manifest

define function add *entry
     to *node
     on @path []byte
has api PUT on "/manifest/<@node>"

as
     // if called on nil call on zero value
     @node = node{} if no @node

     // if empty path then change entry field of node
     if no @path then
         @node entry = @entry
         return store @node

     // lookup the fork based on the first byte of path
     @fork = @node forks at head @path
     // if no fork yet, add the singleton node
     if no @fork then
         @node forks at head @path =
             fork{@path, store node{@entry}}
         return store @node

     @common = prefix of @path and @fork prefix // common cannot be empty
     @rest = @fork prefix from @common length
     @newnode = node{}
     @newnode forks at head @rest = fork{@rest, @fork node}
     @midnode = self @entry to @newnode on @path from @common length
     @node forks at head @path = fork{ @common, @midnode }
     @node store
```

**Definition 82 – Manifest API: remove.**

```
   // /manifest
```

```
define function remove @path []byte
    from *node
has api DELETE on "/manifest/<@node>/<@path>"
as
    // if called on nil call on zero value
    return node{} if no @node

    // if empty path then change entry field of node
    if no @path then
        return nil if @node forks length == 0
        @node entry = nil //entry exists
        return store @node

    // lookup the fork based on the first byte of path
    @fork = @node forks at head @path
    // if no fork yet, add the singleton node
    return @node if no @fork

    @common = prefix of @path and @fork prefix // common cannot be empty
    return @node if @common and @fork prefix have different length          // path not found

    @rest = @fork prefix from @common length
    @newnode =  self @rest from @fork node
    if no @newnode then                 // deleted item was terminal node, delete fork
        @node forks at head @res = nil
    else if @newnode forks length == 1 then // compact non-forking nodes
        @singleton = @newnode forks first
        @newprefix = @common append @singleton prefix
        @node forks at head @path =
            fork{ @newprefix, @singleton node }
    else
        @node fork at head @path node = @newnode

    @node store
```

**Definition 83 – Manifest API: merge.**

```
  // /manifest

define function merge @new *node
    to  @old *node
has api POST on "/manifest/<@old:reference>/<@new:reference>"
as
    // if called on nil call on zero value
    return @new if no @old
    return @old if no @new
    @node = node{ @new or @old }
    @new forks pos or @old forks pos
        each bit @pos go
            @fork = merge.fork @new forks at @pos
                to @old forks at @pos
            @node forks at @fork prefix head = @fork
    @node store

define function merge.fork @new fork
    to @old fork
as
    @common = prefix of @new prefix and @old prefix
    @restnew = @new prefix from @common length
    @restold = @old prefix from @common length
    if no @restnew and no @restold then
        return fork{@common, merge.node @new reference to @old
           reference}
    @node = add @new reference to nil on @restnew
        add to @old reference @restold
    fork{ @common, @node }
```

### 2.5.3 Resolver

**Definition 84 – Resolver.**

```
  // /resolver

define type resolver
    api url
    address ethereum/address
    tlds []string

```

```
define function resolve @host []byte through @resolver
as
    @tld = @host split on '.'' last     @resolver = @resolvers
       any tlds any ==  @tld

    ethereum/call "getContentHash" of @host using @resolver api
        at @resolver address
        as address
```

### 2.5.4 Pinning

**Definition 85 – Pinning.**

```
   // /pin

define type pin
    reference
    chunks uint64

define function list
    return []pin
has api GET "/pin/"

define function view reference
    return uint64
has api GET "/pin/<reference>"

define function pin reference
    return uint64
has api PUT "/pin/<reference>"

define function pin reference
    return uint64
has api DELETE "/pin/<reference>"
```

### 2.5.5 Tags

**Definition 86 – Tags.**

```
// /tag

define type tag
    id      [segment size]byte
    reference     // the current root
    complete bool // if local upload finished
    total  uint64 // number of chunks expected
    split  uint64 // number of chunks split
    stored uint64 // number of chunks stored locally
    seen   uint64 // number of chunks already in db
    sent   uint64 // number of chunks sent with push-sync
    synced uint64 // number of chunks synced


define function list
    return []tag
has api GET "/tag/"

define function view reference
    return uint64
has api GET "/tag/<reference>"

define function add reference
    return uint64
has api POST "/tag/"

define function remove reference
    return uint64
has api DELETE "/tag/<reference>"
```

### 2.5.6 Storage

**Definition 87 – Public storage API.**

```
// /bzz

define function upload @data stream of byte
has api POST on "bzz:/<host>/<path>" from @data as body
as
    @root = resolver/resolve @host
```

```
    @manifest = access/unlock @root
    @entry = file/upload @data
    @reference =  chunk/store @entry as []byte
    manifest/add @reference to @manifest on @path


define function download @path []byte     from @host []byte
has api POST on "bzz:/<host>/<path>" from @data as body
as
    @root = resolver/resolve @host
    @manifest = access/unlock @root
    @entry = manifest/lookup @path in @manifest
    file/download @entry reference
```

## 2.6 Access Control

**Definition 88 – Access control: auth, hint, parameters, root access manifest.**

```
   // /access

define type auth as "pass"|"pk"
define type hint as [segment size]byte

// access control parameters
define type params
  auth                          // serialises uint8
  publisher crypto/pubkey       // 65 byte
  salt                          // salt for scrypt/dh
  hint                          // hint to link identity
  act       *node               // reference to act manifest root
  kdf                           // params for scrypt
}

// root access manifest
define type root
    params
    reference
```

**Definition 89** – **Session key and auth with credentials.**

```
// /access

define function session.key.pass from hint
    with  salt
    using kdf
as
    crypto/scrypt from input/password using @hint
        with @salt using @kdf

define function session.key.pk from hint
    with crypto/pubkey
    using salt
as
    crypto/shared.secret between
        input/select key by @hint
            and @pubkey
        hash with @salt

define function session.key
    using params
as
    if @params auth == "pass" then
        return session.key.pass from  @params hint
            with @params salt using @params kdf

    session.key.pk from @params hint
        with @params publisher using @params salt
```

**Definition 90** – **Access key.**

```
// /access
define function access.key
    using params
as
    @key = session.key using @params
    return @key if no @params act
    act.lookup @key in @params act

define function act.lookup key
    in @act *node
as
```

```
        manifest/lookup hash @key and 0
            in @act
                xor hash @key and 1
```

**Definition 91** – **Access control API: lock/unlock.**

```
   // /access

// control
define function lock reference
    using params
has api POST on "/access/<address>"
    with @params as body
as
    @key = hash @reference address and @key
    @encrypted = @reference as bytes
        crypto/crypt with @key
    root{ @params, @encrypted }
        store


define function unlock address
has api GET on "access/<address>"
as
    @root = retrieve @address
        try as root otherwise return @address
    @key = access.key using @root params
    @root encrypted crypto/crypt with @key
        as *node
```

**Definition 92** – **ACT manipulation API: add/remove.**

```
   // /access

define type act as manifest/node

define function add @keys []crypto/pubkey
    to *root
has api PUT on "/access/<root>/"
    with @keys as body
as
    // get params from the root access structure
```

```
    @params = retrieve @root as root params
    @access.key = access.key using @params
    @keys each @key
        @session.key = session.key using @params
        manifest/add @access.key xor hash @session.key with 1
            to @act on hash @session.key with 0


define function remove @keys []crypto/pubkey
    from *root
has api DELETE on "/access/<root>"
    with @keys as body
as
    // get params from the root access structure
    @params = retrieve @root as root params
    @keys each @key
        @session.key = session.key using @params
        manifest/remove hash @session.key with 0
            from @params act
```

## 2.7 PSS

### 2.7.1 PSS message

### 2.7.2 Direct pss message with trojan chunk

Pss has two fundamental types, a message and a trojan chunk structure which wraps the encrypted serialised message and contains a nonce that is mined to make the resulting chunk's content address (BMT hash) to match the targets.

**Definition 93 – Basic types: topic, targets, recipient, message and trojan.**

```
   // /pss


define type topic        as [segment size]byte        // obfuscated topic matcher
define type targets      as [][]byte        // overlay prefixes
define type recipient    as crypto/pubkey

```

```
// pss message
define type message
    seal    segment
    payload [!:4030]byte    // varlength padded to 4030B

// trojan chunk
// the nonce
define type trojan
    nonce   segment              // the nonce to mine
    key  pubkey                  // compressed format
    message [4064]byte           // encrypted msg
```

The message is encoded in a way that allows integrity checking and at the same time obfuscates the topic. The operation to package the payload with a topic is called *sealing*

**Definition 94 – Sealing/unsealing the message.**

```
// /pss

define function seal @payload []byte
    with topic
as
    @seal = hash @payload and @topic // obfuscate topic
        xor @topic
    return message{ @seal, @payload }

define function unseal message
    with topic
as
    @seal = hash @message payload and @topic
    if @topic == @seal xor @message seal then // check
        return @payload
    return nil
```

Functions `wrap/unwrap` transform between message and trojan chunk. `wrap` takes an optional recipient public key to asymmetrically encrypt the message. The targets are a list of overlay address prefixes derived from overlay addresses of recipients, with length specified to guarantee that a chunk matching it will end up with the recipient solely as a result if push-syncing.

**Definition 95 – Wrapping/unwrapping.**

```
// /pss

define function wrap message
    for recipient
    to  targets
as
    @msg = @message
        (crypto/encrypt for @recipient if @recipient)

    @nonce = crypto/mine @n such that
        @targets any is prefix of
            trojan{@n, @msg} as chunk address
    trojan{@nonce, @msg} as chunk

define function unwrap chunk
    for recipient
as
    @chunk bytes
        (crypto/decrypt  for @recipient if @recipient)
            as message
```

When a chunk arrives at the node, `pss/deliver` is called as a hook by the storage component. First the message is unwrapped using the recipient private key and unsealed with all the topics API clients subscribed to. If the unsealing is successful, message integrity as well as topic matching is proven so the payload is written into the stream registered for the topic in question.

**Definition 96 – Incoming message handling.**

```
// /pss

// mailbox is a handler type, expects payload
// sent sealed with the topic to be delivered via the stream
define type mailbox
    topic
    deliveries stream of []byte

define context mailboxes as []mailbox

define function deliver chunk
    @msg = @chunk unwrap for context recipient
```

```
    mailboxes each @mailbox
        @payload = unseal @msg  with @mailbox topic
        if @payload then
            write @msg payload
                to @mailbox deliveries
```

**Definition 97** – pss API: send.

```
   // /pss

define function send @payload []byte
    about topic
    for recipient
    ?to    targets
has api POST on "/pss/<recipient>/<topic>(?targets=<targets>)"
    with @payload as body
as
    targets = lookup.targets for @recipient if no @targets
    context tag = tag/tag{}
    seal @payload with @topic          // seal with topic
        wrap for @recipient            // encrypt if given
                                          recipient
            to @targets                // mine nonce and returns
                                          trojan chunk
                store                  // to be sent by push-sync
    return tag                         // tag to monitor status
```

**Definition 98** – pss API: receive.

```
   // /pss

define function receive about topic
    on uint64 @channel
has api POST on "/pss/subscribe/<topic>(?on=<channel>)"
as
    @stream = open @channel
    context mailboxes append= mailbox{ @topic, @stream }

define function cancel topic
    on @channel uint64
has api DELETE on "/pss/subscribe/<topic>(?on=<channel>)"
as
```

```
    context mailboxes any ch
```

### 2.7.3 Envelopes

**Definition 99 – Envelope.**

```
   // /pss

define type envelope
    id  [segment size]byte
    sig crypto/signature
    ps  postage/stamp
```

### 2.7.4 Update notifications

### 2.7.5 Chunk recovery

**Definition 100 – Targeted delivery request.**

```
   // /targeted.delivery

define type envelope
    id        segment
    signature crypto/signature


define function wrap address
    ?by @key crypto
    to @targets []target
as
    @key = crypto/random  if no @key
    @n = crypto/mine @n such that
        @targets any is prefix of
            hash @n and @key account
    @sig = crypto/sign hash @n and @address
        by @key
    envelope{ @n, @sig  }
```

**Definition 101 – Prod missing chunk notification with recovery request.**
c

```
// /recovery

define type request
    address
    envelope

define function request address
    with @response bool
    to @targets
as
    @request = request{ @address }
    if @response then
        @request envelope =
            targeted.delivery/wrap @address by @key to @targets
    pss/send @request bytes about "RECOVERY" to @targets
```

# Part I

# Appendix

# Appendix A

# Density-based size estimation

Nodes in Swarm must utilise their full reserve capacity: nodes will potentially further replicate chunks in case they have unused reserve capacity beyond storing their share necessary for a system-wide level of redundancy required. To incentivise this, participating in the redistribution game involves a check called *proof of resources* which is supposed to verify the size of reserve from which the reserve samples are generated. The insight here is that the sample is the lowest range of a uniformly random variate over the entire 256-bit address space. Intuitively, the higher the original volume of the sampled set, the denser it is, the lower the expected maximum value in the sample. Conversely, a constraint on the maximum value of the last element in the sample practically puts a minimum cardinality requirement on the sampled set using a solution called *density based set size estimation*.

We are given $n$ independent, uniformly distributed values between 0 and 1.[1] Let the value of the $k$th smallest of these be $x_k$ (so the smallest of the $n$ values is $x_1$, the second smallest is $x_2$, and so on, up to $x_n$). What is the distribution of $x_k$, given $n$? And what is the threshold value $u$ such that for any given probability $\alpha$, the chance of obtaining an $x_k$ lower than $u$ is $\alpha$?

Note that the distance between any two adjacent values out of $n$ independent uniform variates follows an exponential distribution, as long as $n$ is sufficiently large.[2] The rate parameter of this exponential distribution is $n + 1$, where $n$ is increased by one to account for the fact

[1] Because we work with densities, the actual integer range is not relevant and results obtained for the unit interval can simply be rescaled to the Swarm use case by multiplying with $2^{256}$.

[2] This follows from the fact that $n$ independent uniform variates can be thought of as realizing a Poisson process, whereby the timing of events is random, and it is known that the nearest-neighbour distribution (i.e., waiting time between two consecutive events) is then exponentially distributed.

that the expected gap between adjacent values is $1/(n+1)$.[3]

We are after the distribution of the $k$th value, [4] $x_k$, which then can be thought of as arising from the sum of $k$ independent exponential variables, each with a rate parameter of $n+1$. This is known to result in an Erlang distribution with shape parameter $k$ and rate parameter $n+1$. Using $X(\lambda)$ to denote the exponential distribution with rate $\lambda$ and $E(k,\lambda)$ to denote the Erlang distribution with shape $k$ and rate $\lambda$:

$$\sum_{i=1}^{k} X_i(n+1) = E(k, n+1), \tag{A.1}$$

where the subscript $i$ in $X_i(n+1)$ distinguishes between independent exponentially distributed random variables. The probability density function $E(x,k,n+1)$ of the Erlang distribution itself is given by

$$E(x,k,n+1) = \frac{(n+1)^k x^{k-1} \mathrm{e}^{-(n+1)x}}{(k-1)!}. \tag{A.2}$$

This distribution contains the answer to the first question: what is the distribution of the $k$th smallest value out of $n$ independent uniform variates between 0 and 1? For example, if $k=16$ and $n$ is either 500, 750, or 1000, we get the distributions shown in Figure A.1.

We can now answer the second question: given $n$ and a confidence level $\alpha$, what is the threshold value $u$ for $x_k$ such that the probability that $x_k < u$ is equal to $\alpha$? That is, we wish to know the value $x = u$ at which the probability distribution has encompassed a given area of $\alpha$ (see figure A.2).

The area under the curve of the Erlang distribution is given by its cumulative distribution function $P(x,k,n+1)$, which is known to be

$$P(x,k,n+1) = \int_0^x E(y,k,n+1)\,\mathrm{d}y = \frac{1}{(k-1)!}\int_0^{(n+1)x} t^{k-1}\mathrm{e}^{-t}\,\mathrm{d}t. \tag{A.3}$$

The latter expression is sometimes written as $\tilde{\gamma}(k,(n+1)x)$, where

$$\tilde{\gamma}(k,x) = \frac{1}{\Gamma(k)}\int_0^x t^{k-1}\mathrm{e}^{-t}\,\mathrm{d}t \tag{A.4}$$

[3] For $n=2$, the mean outcome is $x_1 = 1/3$ and $x_2 = 2/3$; for $n=3$, it is $x_1 = 1/4$, $x_2 = 2/4$, $x_3 = 3/4$; and so on: for arbitrary $n$, $x_i = i/(n+1)$, with the gap between adjacent values in this ideal case always being $1/(n+1)$.

[4] An alternative approach using order statistic expresses $x_k$ via a beta distribution. It is very difficult to prove that the Beta distribution's quantile function is a strictly decreasing function of $n$, which is a key piece of the argument presented here. Although this method is exact even for small $n$, in our case, $n$ always a very large number, therefore we adopted the other method.

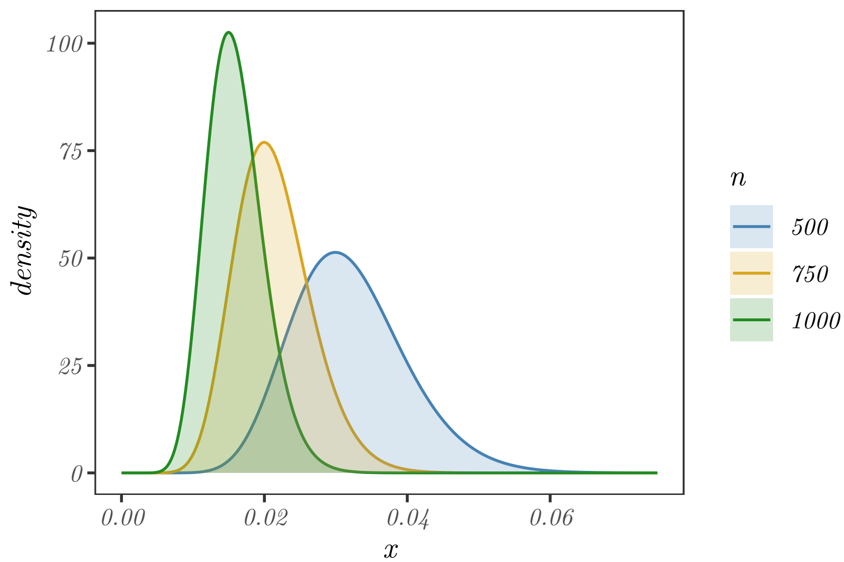


**Figure A.1:** Probability density function $E(x, k, n + 1)$ of the Erlang distribution, with $k = 16$ and $n$ either 500, 750, or 1000.

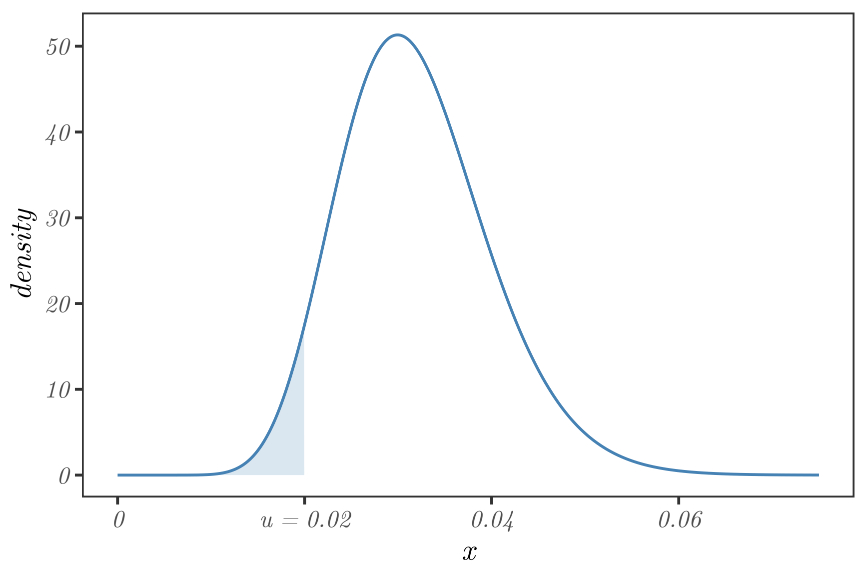


**Figure A.2:** The distribution of $x_{16}$, i.e., the 16th smallest value from among $n = 500$ independently and uniformly drawn variates between 0 and 1. The area under the curve is shaded up to 5% of its area. The point at which the shading stops is therefore the value $u$ for which there is only a 5% chance of getting an even smaller $x_{16}$.

is the regularized lower incomplete gamma function. We therefore want to solve the equation

$$\alpha = \int_0^u E(y, k, n + 1) \, \mathrm{d}y = P(u, k, n + 1) \tag{A.5}$$

for $u$.

Inverting this expression in $u$ (since the cumulative distribution function increases monotonically in $u$, the inverse exists) leads to the quantile function $Q(\alpha, k, n+1)$ of the Erlang distribution: $u = Q(\alpha, k, n+1)$. The quantile function is known to be expressible as

$$Q(\alpha, k, n+1) = \frac{\tilde{\gamma}^{-1}(k, x)}{n+1}, \tag{A.6}$$

where $\tilde{\gamma}^{-1}(k, x)$ is the inverse regularized lower incomplete gamma function. Its particular form is of no interest to us, except for two properties. First, it is positive for all $x$.[5] Second, it is independent of $n$. Instead, the entire dependence of $Q(\alpha, k, n+1)$ on $n$ is given by the $n+1$ term in the denominator of Equation A.6. From this, we conclude that $Q(\alpha, k, n+1)$ is a strictly decreasing function of $n$.

These two points lead to an important consequence. Say we compute the threshold $u$ for a given $\alpha$ and $n$ in order to have an upper bound on a lower quantile. Now, if we were to decrease $n$ but hold all other things equal, the threshold will always get higher than what it was before. The threshold obtained for higher values of $n$ may therefore serve as a conservative estimate of the threshold for lower values: if $u$ is a threshold such that the $k$th smallest out of $n$ uniform variates is only smaller than $u$ in $\alpha$ of cases, then for any amount $m < n$, the chance of the $k$th variate conforming to the same constraint (i.e., $x_k < u$) is now even smaller than $\alpha$.

Conversely, if we were to constrain $x_k$ so that the probability of not getting a value smaller than $u$ is lower than $\beta$ (minimising a higher quantile), we find that the constraint remains true as $n$ is increased.

Armed with these results, let us see how Equation A.6 can be used for the estimation procedure. There are two problems to tackle, ultimately relating to the two aspects of a test's accuracy. First, we want to catch inadequate storers slacking on volume. In other words, we want to constrain the $x_k$ values so that we can safely say that any attacker with a stored volume below an acceptable size $n$ has a probability less than $\alpha$ to obtain such a small $x_k$ by pure chance. Construing the condition for $x_k < u$ as a test to filter honest players (just based on the size of their reserve), $1 - \alpha$ expresses the *sensitivity* of the test. From the previous argument on the monotonic dependence of $\alpha$ on $n$, it is safe to use a condition that requires $x_k$ to stay below a threshold obtained for $n$.

Second, we want to avoid situations when honest participants end up not satisfying the above constraint even though they sampled from a set larger than the required minimum. Given a target volume $m > n$, the error rate of false negatives is guaranteed to be less than

[5] This stands to reason: the quantile function of a distribution on $x \in [0, \infty)$ is itself between 0 and $\infty$, and $\tilde{\gamma}^{-1}(k, x)$ is just the quantile function of the Erlang distribution times the positive constant $n+1$.

$\beta$ obtained from the quantile function with parameters $m, k,$ and $u$. The quantity $1 - \beta$ is the *specificity* or *precision* of the test.

Figure A.3 illustrates this idea, for two different distributions in both the lower- and upper-end estimation. What we want is to choose $u$ to simultaneously make sure that dishonest players do not sneak through the system *and* also that honest players do not get excluded too often. This translates to make both $\alpha$ and $\beta$ as small as possible.

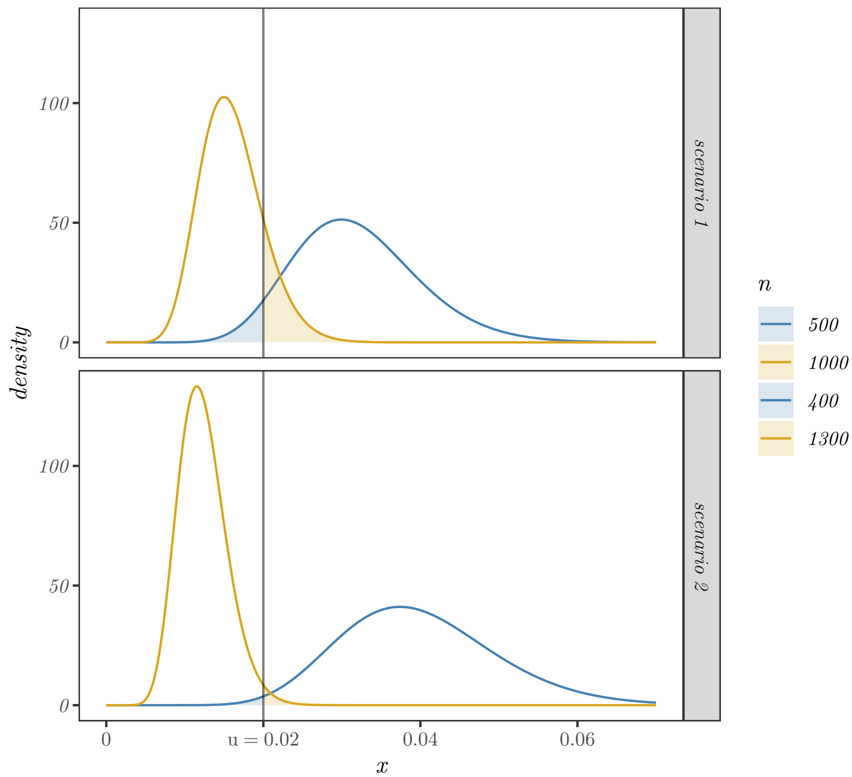


**Figure A.3:** Recall and precision of proof of reserve size validation: Any chosen $u$ will lead to different $\alpha$ and $\beta$ values, depending on $n$. Here $u$ is fixed at 0.02. The top panel shows distributions for $x \equiv x_{16}$ with $n = 500$ (blue) and $n = 1000$ (yellow). The area left under the blue curve to the left of $u$ is equal to $\alpha$ (blue shade); the area under the yellow curve right of $u$ is equal to $\beta$ (yellow shade). If the curves overlap considerably (top), it is impossible to choose an $u$ such that $\alpha$ and $\beta$ are simultaneously small.

One way to try and find the best compromise is by minimising $\alpha + \beta$ (the *accuracy* of the test) and pick the $u$ value at the optimum to be used in the proof of resources test. To this end, one can vary $\alpha$ between 0 and 1 and, for each of its values, solve the equation $\alpha = Q(1 - \beta, k, n + 1)$ (where $n$ is the larger value, used for estimating $\beta$). This way, we get a $\beta$ value for every possible $\alpha$. Then, we can find the combination which minimises $\alpha + \beta$, and determine the value of $u$ that leads to this optimum. As illustrated in figure A.4, larger values of $k$ yield a trade-off curve along which better accuracies can be achieved.

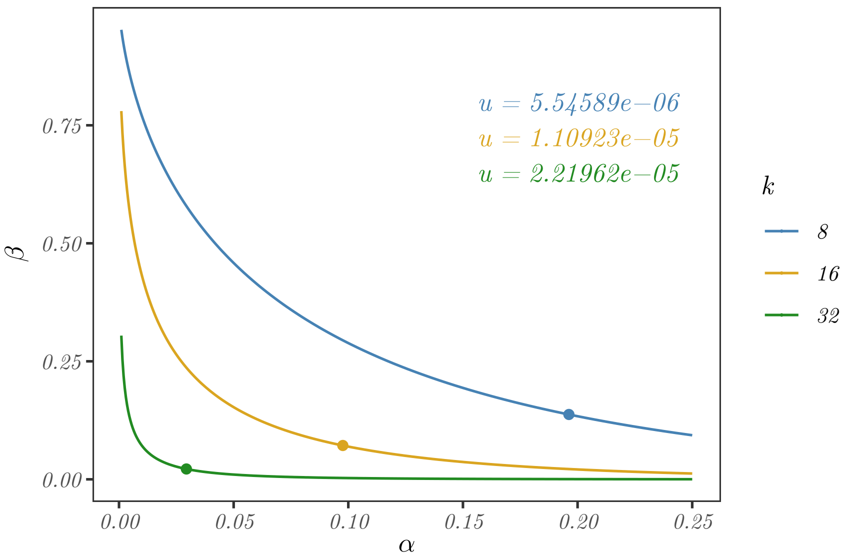


**Figure A.4:** Optimal accuracy of reserve size probe By increasing $k$, one can get better optima for minimising $\alpha + \beta$. Here $n = 10^6$ for estimating $\alpha$ and $2 \cdot 10^6$ for estimating $\beta$, and $k$ is either 8, 16, or 32 (colors). The values of $u$ associated with the optima are also shown.

Table A.1 summarizes the important numerical results in Figure A.4.[6]

| $k$ | $\alpha$ | $\beta$ | $u$ | $u \cdot 2^{256}$ |
|---|---|---|---|---|
| 8 | 0.196216 | 0.1373622 | $5.54589 \cdot 10^{-6}$ | $6.421705 \cdot 10^{71}$ |
| 16 | 0.097612 | 0.0716570 | $1.10923 \cdot 10^{-5}$ | $1.284401 \cdot 10^{72}$ |
| 32 | 0.029386 | 0.0219151 | $2.21962 \cdot 10^{-5}$ | $2.570140 \cdot 10^{72}$ |

**Table A.1:** Proof of density parameter calibration. Assuming $n = 10^6$ and $m = 2 \cdot 10^6$ to calculate recall and precision error rates $\alpha$ and $\beta$, respectively, the cutoff value for the proof is calibrated by optimizing on acccuracy using sample sizes $8, 16, 32$.

[6] Since the hash function used to generate random variates does so in the range $[0,\ 2^{256} - 1]$ instead of $[0,\ 1]$, the calculated thresholds are scaled with $2^{256}$ to show where they would fall in their actual range.

# Appendix B

# Source of randomness

As the *neighbourhood selection anchor* will directly affect which neighbourhood wins the pot, it is prudent to derive the randomness from a source of entropy that cannot be manipulated. A common solution to obtain randomness is to have independent parties committing to a random nonce with a stake. The random seed for a round is defined as the xor of all revealed nonces. Given the nonces are independent sources fixed in the commit, no individual participant has the ability to skew randomness by selecting a particular nonce. Thanks to the commutativity of xor, the order of reveals is also irrelevant. However, if the reveal transactions are sequential, committers compete at holding out since the last one to reveal can effectively choose the resulting seed to be either including its committed nonce (if they do reveal) or not (if they do not reveal). The threat to slash the stake of non-revealers serves to eliminate this degree of freedom from last revealers and thus renders this scheme a secure random oracle assuming there is at least one honest (non-colluding) party.[1]

Now note that the redistribution scheme already has a commit reveal scheme as well as stake slashed for non-revealers, so a potential random oracle is already part of the proposed scheme. Incidentally, the beginning of the claim phase is when new randomness is needed to select the truth and a winner. Importantly, these random values are only needed if there is a claim which implies that there were some commits and reveals to choose from. Or conversely, if there are no reveals in the round,[2] the random seed is undefined but is also not needed for the claim.

The random seed that transpires at the beginning of the claim phase can serve as the *reserve*

[1]If the stake is higher than the reward pot, one cannot afford being slashed with even just one commit without a loss. If this cannot be guaranteed, slashing of the stake is not an effective deterrent.

[2]If saboteurs get slashed or frozen in the claim transaction, if there is no claim, the committers get away without being punished. This can be remedied if the staking contract keeps a flag on each overlay (set when commits, unsets when reveals in the same round) and the check and punishment happens as a result of a commit call in the case the flag is found set.

*sample salt* (nonce input to modified hash used in the sampling) with which the nodes in the selected neighbourhood can start calculating their reserve sample.

The neighbourhood for the next round is selected by the *neighbourhood selection anchor*, which is, similarly to the truth and winner selection nonces, deterministically derived from the same random seed. Unlike the nonces used to select from the reveals, neighbourhood selection should be well defined for the following round even if a round is skipped, i.e., when there is no reveals. To cover this case skipped rounds keep the random seed of the previous round. However, in order to rotate selected neighbourhoods through skipped rounds, we derive the neighbourhood selection anchor from the seed by factoring in the number of game rounds passed since the last reveal.[3]

In order to provide protection against the case when each committer in the neighbourhood is colluding, and can afford losing stake we need to make sure that the entropy is still high otherwise the nodes can influence the neighbourhood selection nonce and reselect themselves or a fixed colluding neighbourhood (or increase the chances of reselection).

**Definition 103 – Random seed for the round.**
Define the random seed of the round as the xor of all obfuscation keys sent as part of the reveal transaction data during the entire reveal period:

$$\mathcal{R} \quad : \quad \Gamma \mapsto \mathit{Nonce} \tag{B.1}$$

$$\mathcal{R}(\gamma) \quad \stackrel{\text{def}}{=} \quad \begin{cases} \mathcal{R}(\mathit{Prev}(\gamma)) & \text{if } |\mathit{Reveals}(\gamma)| = 0 \\ \underline{\bigvee}_{r \in \mathit{Reveals}(\gamma)} \text{NONCE}(r) & \text{otherwise} \end{cases} \tag{B.2}$$

**Lemma 104 – Round seed is a secure random oracle.**
The nonce produced by xoring the revealed obfuscation keys is a correct source of entropy.

*Proof.* Assuming $n$ independent parties committing, choosing any particular nonce will leave the outcome fully random.

[3]Otherwise a selected neighbourhood could collude maliciously not to commit/reveal and have the pot roll over to the following round. By simply holding out for a number of redistribution rounds, they could unfairly multiply their reward when they eventually claim the pot.

# Appendix C

# Parameter constants

Table C.1 lists the constants used by Swarm with their type, default value and description.

| CONSTANT NAME | TYPE | VALUE | DESCRIPTION |
|---|---|---|---|
| `BZZ_NETWORK_ID` | *uint64* | 0 | ID of the swarm network |
| `PHASE_LENGTH` | *uint64* | 38 | length of commit phase of the redistribution game round in number of blocks |
| `ROUND_LENGTH` | *uint64* | 152 | length of one redistribution game round in number of blocks, fixed at 4 times the `PHASE_LENGTH` |
| `NODE_RESERVE_DEPTH` | *uint8* | 23 | size requirement for client reserve capacity given in base 2 log number of chunks |
| `SAMPLE_DEPTH` | *uint8* | 4 | base 2 log number of chunks in sample |
| `MAX_SAMPLE_VALUE` | *uint256* | $1.2844 \cdot 10^{72}$ | maximum value for last transformed address in reserve sample, i.e., $< 1\%$ chance the sampled set size is below a fourth of the prescribed node reserve size. |
| `MINIMUM_STAKE` | *uint256* | 10 | minimum stake amount in BZZ to be redefined as minimum stake given in storage rent units |
| `MIN_STAKE_AGE` | *uint256* | 228 | minimum number of blocks stakers need to wait after update or creation for the stake to be useable. Defaults to one and a half rounds to prevent opportunistic manipulation of stake after a neighbourhood is selected. |

| | | | |
|---|---|---|---|
| `PRICE_2X_ROUNDS` | *uint64* | 64 | number of rounds it takes for the price to double in the presence of a consistent lowest degree signal of undersupply. |
| `NHOOD_PEER_COUNT` | *uint8* | 4 | minimum number of nodes required to form a fully connected neighbourhood. |

**Table C.1:** Parameter constants