Enshroud:
Privacy as a Service for Public Blockchains

v1.9.1: 13 December, 2022
v2.0.1: 07 February, 2024 (post-implementation update)
v2.1.1: 18 April, 2025 (corrections post-testing)

Abstract:

Privacy of value transactions is generally not available on most public blockchains. This is a consequence of their nature as distributed public ledgers. Pseudonymity is provided via wallet addresses, but spends are done in the open. There is weak obfuscation of *who*, but typically zero obfuscation of *what.* This is a disappointing and indeed dangerous characteristic for an industry which aspires to rewrite and replace global systems of currency, banking, and finance.

To address the need for privacy, specific coins have been developed which possess privacy-enhancing properties, such as Monero ($XMR), Zcash ($ZEC), Dash ($DASH), and Pirate Coin ($ARRR). Bitcoin has recently improved its privacy infrastructure through the deployment of Taproot and TAS. Blockchains which implement smart contracts with native support for private data are also emerging, e.g. Secret ($SCRT), Dero ($DERO), Mina ($MINA), and Dark.Fi lib. While these projects usefully provide improved privacy, it remains disappointing that one has to utilize specialty tokens or deploy apps on specialty blockchains in order to gain access to meaningful privacy. Layer 1 or 2 solutions using ZKs also exist (Aztec, Railgun, Offshift), but face a host of difficulties related to cumbersome bridging issues and higher gas costs.

Enshroud, as described in this paper, presents a generic, blockchain-independent mechanism for privacy as a service, by which private transactions can be conducted on otherwise public ledgers. The key concept is representing value as *eNFT*s (enshrouded / encrypted non-fungible tokens), in which the value actually being spent is concealed in the encrypted NFT metadata. In this way the eNFT can be minted and burned in the normal, public manner specified by the ERC-1155 NFT standard, while simultaneously facilitating the transfer of an invisible "attached" quantity of another asset. An eNFT is thus a receipt token for a share of asset value deposited into a smart contract. While deposits to and withdrawals from the smart contract vault are public by necessity, the intermediate circulation of eNFTs between wallets is opaque.

A parallel benefit in addition to privacy is that one-to-many batch payments of tokens (such as ERC-20s) can be implemented in a single transaction, with increased privacy. This could also increase gas efficiency in certain payment models, and work similar to Multisender.app. It represents a different approach to that afforded by ERC-998 composable NFTs, which makes transferring groups of tokens in a single transaction possible, but provides no privacy on-chain. Also, Enshroud transactions are not susceptible to MEV front-running, because the submissions to the mempool are details hash-blinded and cannot be interpreted.

We shall show herein that Enshroud implements a private payment method which leverages the consensus mechanism of a public blockchain, without the need for a separate Layer 2 settlement network (such as Lightning) or a pegged sidechain (such as Liquid), and which can be deployed on any Layer 1 or Layer 2 blockchain capable of supporting the ERC-1155 standard.

Extending ERC-1155

I. MetaData Schema

Most NFTs reference an external digital object, such as a unique JPEG image or an MP4 video. Because such objects are typically large, and on-chain storage is expensive, the data is stored off-chain, usually in a Web2 centralized database, but sometimes pinned on a distributed filesystem such as IPFS, or on a specialized blockchain like Filecoin, Aleph.im or TheGraph. The path to the actual data representing the NFT, to which the NFT acts like a transferable title deed, is specified in a block of metadata in JSON format. The location of the unique NFT metadata in turn is specified by means of a URI contained within the token itself. The schema of this JSON metadata, per the ERC-1155 specification, looks like this:

```
{
    "title": "Token Metadata",
    "type": "object",
    "properties": {
        "name": {
            "type": "string",
            "description": "Identifies the asset to which this token represents"
        },
        "decimals": {
            "type": "integer",
            "description": "The number of decimal places that the token amount
should display - e.g. 18, means to divide the token amount by 1000000000000000000
to get its user representation."
        },
        "description": {
            "type": "string",
            "description": "Describes the asset to which this token represents"
        },
        "image": {
            "type": "string",
            "description": "A URI pointing to a resource with mime type image/*
representing the asset to which this token represents. Consider making any images
at a width between 320 and 1080 pixels and aspect ratio between 1.91:1 and 4:5
inclusive."
        },
        "properties": {
            "type": "object",
            "description": "Arbitrary properties. Values may be strings, numbers,
object or arrays."
        }
    }
}
```

For our purpose of an eNFT representing another asset type, the {image} property is irrelevant, and the nested generic {properties} object (**bolded**) can be utilized to describe the pertinent details of the eNFT. This nested object will be encrypted so that only the wallet owner can read it. The nested {properties} JSON object will be structured like this:

“properties”: {
“enshrouded”: {
“type”: “string”,

```
        "description": "Enshroud properties, JSON sub-object in encrypted base64 format"
    }
}
```

Where, once decrypted using the appropriate AES key, the {enshrouded} field will have this schema:

```
    {
        "id": {
            "type": "string",
            "description": "{id}, an echo of the unique token ID shown in the metadata URI, leading zero-padded to 64 hex digits (integer quantity but represented as a string)"
        },
        "schema": {
            "type": "string",
            "description": "revision of Enshroud schema used, e.g. v1.0"
        },
        "owner": {
            "type": "address",
            "description": "the address to which this eNFT was minted (included to prevent a valid eNFT from being assigned to a different address)"
        },
        "asset": {
            "type": "string",
            "description": "The actual crypto asset represented by this eNFT (usually an ERC-20 contract address, e.g. WETH, WBTC, etc.)"
        },
        "amount": {
            "type": "integer",
            "description": "The quantity of the named asset, in decimal (not hex), to {decimals} precision"
        },
        "rand": {
            "type": "string",
            "description": "A random salt which makes the eNFT's details harder to guess, 16 bytes base64-encoded (22 bytes)"
        },
        "generation": {
            "type": "integer",
            "description": "The number of eNFTs generated since value deposited"
        },
        "expiration": {
            "type": "integer",
            "description": "OPTIONAL: an expiration date, as a Unix timestamp"
        },
        "growth": {
            "rate": "integer",
            "units": "string, such as percent or bpd (basis points per day)",
            "description": "OPTIONAL: a fixed growth rate, e.g. 1 bpd compound interest"
        },
        "cost": {
```

```
                “rate”: “integer”,
                “units”: “string, such as % APR”,
                “description”: “OPTIONAL: a fixed fee, e.g. 1% APR for gold storage agio fee”
        },
        “signer”: {
                “type”: “string”,
                “description”: “Metadata Validator Oracle (MVO) ID, used to lookup signer’s
    pubkey in the ERC-1155 smart contract’s config data”
        },
        “signature”: {
                “type”: “string”,
                “description”: “privkey signature of Metadata Validator Oracle (MVO) on all of
    the above fields (130 hex char digest)”
        }
    }
```

Example: an eNFT with unique ID 0x12345, representing 1.41421356 of DAI on Eth Mainnet, 3rd generation, no optional fields included:

```
{
    “title”: “eNFT Metadata”,
    “properties”: {
        “name”: “Enshrouded”,
        “decimals”: “18”,
        “description”: “”,
        “image”: “”,
        “properties”: {
            “enshrouded”: {
                “id”:
“0000000000000000000000000000000000000000000000000000000000012345”,
                “schema”: “1.0”,
                “owner”: “0x0123456789aBcDeF0987654321fEDcbA01234567”,
                “asset”: “0x6B175474E89094C44Da98b954EedeAC495271d0F”, // DAI
                “amount”: “1414213560000000000”,
                “rand”:”xzWuANl4fufVsr+1mIFjww”,
                “generation”: “3”,
                “signer”: “MVO-001”,
                “signature”: “<actual signature of {signer} on previous fields, without
whitespace, 130 chars>”
            }
        }
    }
}
```

Note that the {signature} is made on the JSON fields in {enshrouded} above it, and then the whole is encrypted using the AES-256 key and substituted as a Base64 string in place of the {enshrouded} object, shown above as cleartext (in grey). (The reason for the use of Base64 for encrypted data is that it’s the standard for sending such binary data across HTTP connections.)

The key used for encryption of {enshrouded} elements must be a unique ephemeral key created specifically for the encryption of each individual eNFT. The MVO creating and signing the eNFT will apply to an Auditor for a random symmetric key (AES-256) to use in lieu of the pubkey of the owning address. This is necessary because there's no way to obtain the pubkeys for arbitrary payees based on their address. (An address is simply the last 20 bytes of a 32-byte hash of the pubkey.) Also, even if pubkeys were always available, keys utilized for signing (as ECDSA account keypairs are) generally should not also be used for encryption. Auditor nodes will maintain the mapping of eNFT encryption keys in persistent storage, effectively operating a distributed key server. A dApp can submit a signed request (see below) to an MVO to obtain (only) the AES keys associated with each extant eNFT validly owned by their address. In this sense these symmetric keys are "private" keys, in that they cannot be retrieved by anyone else. Using these keys, the dApp/user can always view the actual cleartext contents of their eNFTs, which is required in order to spend them. Decrypted eNFTs together with their assigned keys can optionally be stored in a local wallet file as a backup. Note that ephemeral AES keys relate only to decrypting eNFTs, and do not permit anyone to spend the eNFT itself. (See section VI below on Auditors for further details on how the key store is utilized and maintained.)

Pictorial representation:

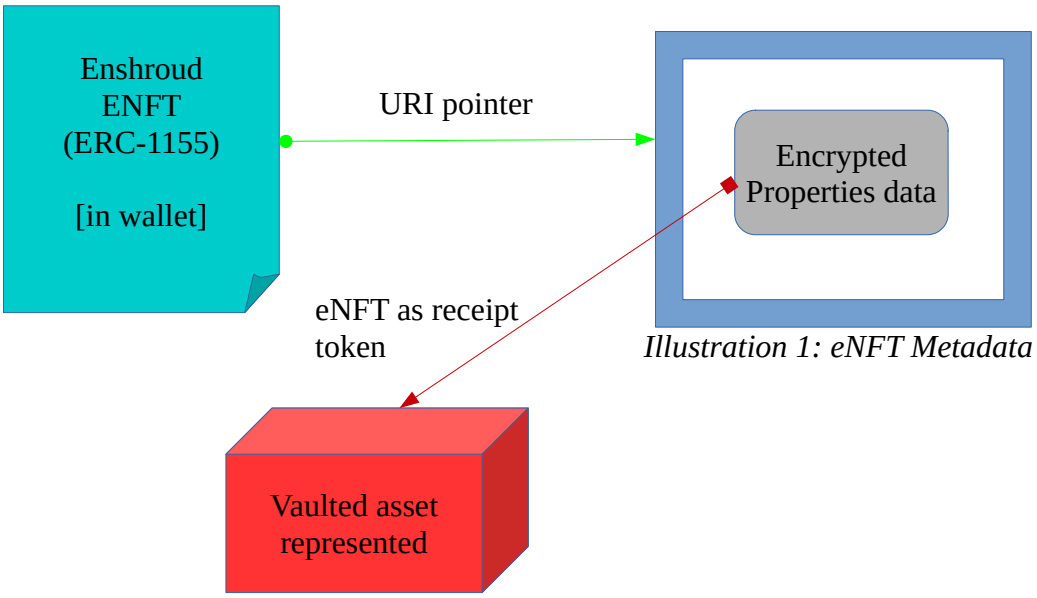


Illustration 1: eNFT Metadata

II. URIs

Metadata URIs are specified by the issuing ERC-1155 compliant smart contract when the NFT is minted, using a URI event emitted to the blockchain event log. Normally there is also a *uri(ID)* view function which will reiterate the same data.

Since the tokens are non-fungible, the account balance quantity will always be set to 1 at minting, as recommended in the standard.

For an *ordinary* NFT, the metadata URI for the token is of the form:

https://token-cdn-domain/
0000000000000000000000000000000000000000000000000000000012345.json

where the filename is the value of {id} with ".json" appended. The "token-cdn-domain" is typically on Web2, something like "*https://s3.amazonaws.com/your-bucket/metadata/*". An IPFS URI could also be used, e.g. "*ipfs://QmWS1VAdMD353A6SDk9wNyvkT14kyCiZrNDYAad4w1tKqT/{id}.json*".

In turn, the "image" URI defined in the token metadata itself points to a stored image or other representation of the artwork or other digital artifact whose ownership is proved by the NFT.

Because in the case of an eNFT there is no external artwork, only a modest amount of JSON data, which even signed and encrypted is probably less than 512 bytes, it becomes possible that the metadata could be stored directly within the state of the smart contract. This is particularly plausible on lower cost blockchains where state storage is inexpensive. In order to support this, the contract would need to add and maintain a mapping like this:

mapping(address => mapping(uint256 _ID => string "JSON metadata, encrypted form"))

This mapping would store all of the eNFTs currently minted to a particular address, indexed by their IDs. Note that the {enshrouded} details of each eNFT are only readable by the owner, since they are encrypted. As eNFTs are burned (spent), they would be removed from the mapping. (Note that an InterableMapping might be required.)

The obvious problem with this mechanism would be gas costs. Since every 32 bytes stored in EVM contract storage costs 20,000 gas, storing an object of ~512 bytes would require 320k in gas! At 10 gwei (above average in the current environment) this is 0.0032 ETH, or roughly US$5 at today's price of approx. $1600 = 1 ETH. This is clearly unacceptable, but might be more plausible on some less congested, lower-cost chains. However one cannot guarantee that such chains will not become more congested or expensive in the future.

It is also possible to store the metadata in a decentralized way off-chain, for example on IPFS or on another blockchain purpose-built for storage, such as Filecoin, Aleph, The Graph, etc. The problems here are twofold: 1) latency, and 2) dependency on a second chain to access asset value. While second-chain storage is adequate for long-lived metadata such as that normally associated with garden variety NFTs, it isn't really suitable for what amounts to a flat-file database utilized on Layer 1.

Thus we arrive at the clear best option: the metadata for eNFTs should be appended to the event log of the Layer 1 blockchain, as a recorded (but not indexed) logging parameter emitted during minting. Because the relevant details are stored encrypted, only the wallet owning the emitted eNFT can decrypt the metadata. The gas profile of this technique is much more conducive: event parameters cost 375 gas per additional item supplied, plus 8 gas per byte of data. So adding the encrypted metadata to an eNFT's URI event would cost 375 + (8 * 512) = 4471 gas. At 10 gwei this is about $0.075, which is quite acceptable. Note that this extra cost is applied for each additional payee, or eNFT generated. This adds only about another 50% to the cost of supplying the encrypted metadata to the SC in calldata.

With the smart contract configured in this way, the return value of the *uri(ID)* function call would logically become the signature of the event which emitted the metadata, such as:

bytes4(keccak256("URI(uint256,string)")).

However this is not useful. Note that smart contracts cannot themselves access the event log, thus they cannot obtain the metadata in order to comply with the recommendation of the standard. Accordingly, in the EnshroudProtocol SC the uri() method has been changed to *uri(string metadata)*, which simply prepends the baseURI (defined as the chainId plus a '-:', e.g. "1-:" for Ethereum Mainnet) to the passed metadata, and is only called internally while preparing the emission of the URI event.

Therefore, the procedure for a wallet client dApp to access a given eNFT found in an account is:

1. Retrieve the JSON metadata, directly from the event log via a URI event filter indexing the ID.
2. Decrypt the {enshrouded} string value using the AES key (retrieved through an MVO, or from a saved local *wallet.dat* file), and parse the JSON object.
3. Using the signer ID, retrieve the pubkey of the MVO that signed from the smart contract.
4. Verify the MVO's signature (*ecrecover()*) on the decrypted {enshrouded} fields above it.
5. Optionally, sort all eNFTs found as desired, and aggregate values for each unique asset type.

Steps 1 & 2 are automated by means of a signed wallet request sent to an MVO (see below).

III. Smart Contract (SC) Configuration

The smart contracts must also define the operating parameters for the Enshroud system. As already hinted, they must provide dApps with the ability to interact with Metadata Validator Oracles (MVOs). (See Sidebar 1 below for a description of the role played by MVOs and how they are accessed.) The contracts must make available to callers on a read-only basis information about all MVOs defined, including:

- A list of the IDs of all MVOs defined
- The URIs at which the API of each MVO listens (typically a *http://IP:port* URL)
- The corresponding ID of each MVO (used for lookups, and specified as {signer} in signatures)
- The account address corresponding to the private key with which each MVO signs replies and {enshrouded} eNFT data (used to confirm MVO signatures by recovering a matching address)
- A ECDSA public key permitting ECIES encryption of client messages to each MVO (separate from the pubkey used to verify signatures; see Sidebar 2 below)
- The quantity of ENSHROUD tokens staked by each MVO, and the staking address (source)
- The last known current status (enabled / disabled) of each MVO
- The definition of *N of M*, that is how many signatures *N* out of *M* total MVOs are required
- The number of confirmation blocks an eNFT must "dwell" after minting before becoming eligible as inputs.

This info is provided by getter functions in the MVOStaking contract (returned as a JSON object), except for the last two items, which are queried directly from the main EnshroudProtocol contract. Updating the MVO-related definitions in the contract state results from MVO operators performing normal operations to stake their $ENSHROUD tokens and define their hosting configs. Modifications are also a privileged function carried out by admin keys only, e.g. to slice an MVO's staking or mark it disabled. Admin operations are multisig and three admins are required (to guarantee that nothing can be changed unilaterally by a dev or other insider.) When configuration changes occur, they are signaled by an event, which can trigger a re-read of the current data by any listeners (something like a HUP signal with a daemon process).

In addition to the MVO configuration, for each deployed chain there's a static supported asset configuration supplied with the dApp. This config consists of a list of predefined "recommended" ERC-20 compatible tokens commonly used on that chain. For each such token this would include:

- The asset name and token symbol, such as WBTC, WETH, DAI
- The type of the asset (e.g. ERC-20 token)
- The best deposit method (ERC-2612 or ERC-20)
- The applicable fee schedule for deposits and withdrawals of the asset.
- Any free, public price feed for determining the USD value of the token on that chain.

The latter (fee schedule) is also defined in a mapping in the EnshroudProtocol contract (overriding the dApp's static config). Three admin keys can make adjustments to deposit and/or withdrawal fees, which are paid to the DAOPool smart contract (for user $ENSHROUD APY staking, 95%) and to the DAO Treasury (5%).

In addition, any user may deploy additional token assets onto the platform, as with user-created liquidity pools on e.g. Uniswap v3 or SushiSwap MISO. User-added tokens must be erc20/erc777/erc4626 compatible and will be established with the system default deposit/withdraw fees, which can be customized subsequently by three admins. Asset supportability, as well as the best deposit method (permit/transfer or approve/transfer) will be determined (by the dApp) via introspection of the token's smart contract address.

IV. Supported Operations

There are three fundamental operations which the SC must support:

1. Depositing a supported {asset} and minting one or more eNFTs of the same asset type.
2. Spending one or more eNFTs (of one or more types) in exchange for one or more new eNFTs.
3. Redeeming one or more eNFTs (all of one type), burned for an equal quantity of {asset}.

In each case, all output eNFTs are always newly minted, and all input eNFTs are always burned. That is, at no time does an existing eNFT get transferred from one wallet account to another. By implication, only case 2 provides meaningful privacy, since cases 1 and 3 involve matching quantities of {asset} moving between addresses on the blockchain, and those transfers are done in the clear. The sum totals of any eNFTs involved in a deposit or withdrawal are therefore apparent.

Sidebar 1:

> Before we continue, we need to define a term. An **MVO**, short for Metadata Validator Oracle, is a quasi-Layer 2 validator node in the Enshroud system. MVOs service requests from dApp clients to perform a supported operation involving eNFTs. Essentially, their role is to proof-read the client request data, making sure the client isn't up to no good (such as by attempting to spend more than they have, etc.). They then provide an instruction block for the dApp, suitably blinded with hashes so that it can go into a block without exposing what's really going on, and attest to those instructions, parameters and details hashes with their own signature. Countersignatures of additional MVOs are then solicited, checking the work of the first. Once the required number of signatures has been generated, the "lead" MVO returns the multi-signed instruction block to the dApp client, which adds its own wallet signature and submits the whole

transaction to the mempool on the blockchain. The MVOs operate a PoS (proof-of-stake) validation system, but don't actually implement a blockchain between them. As the name implies, they act as oracles for signatures approving and attesting to user actions. At no time do they receive or transmit any user funds on-chain.

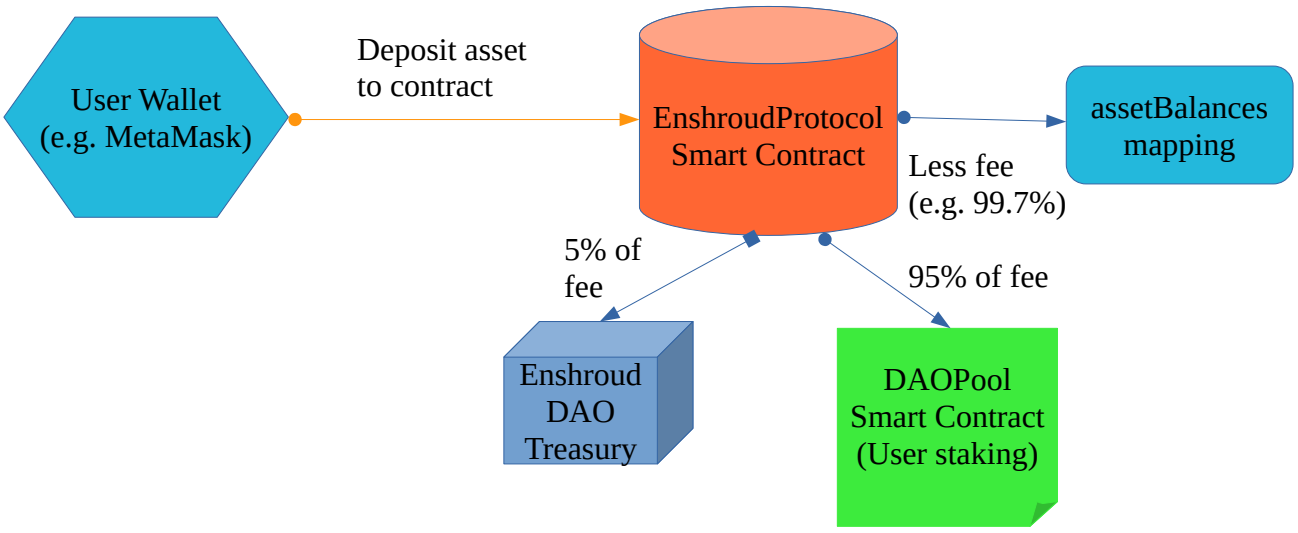
Let us now proceed to examine the details of each of these three basic operations.

1a. Depositing to mint eNFTs

In order to mint eNFTs, a wallet address must have a non-zero balance in the EnshroudProtocol contract's mapping for a given asset:

mapping(address => mapping(address => uint256)) public assetBalances;

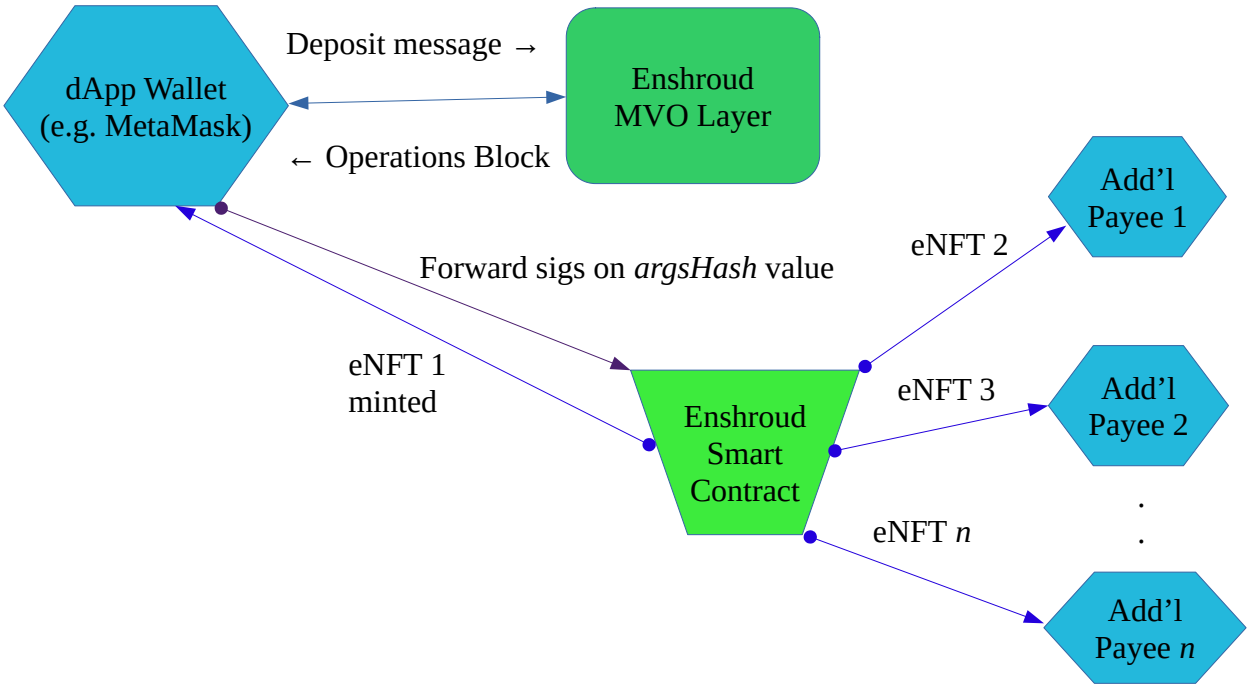
This is achieved by the dApp calling one of these smart contract methods: *depositEth()* (converts ETH to WETH), *depositTokens()* (for ERC20-compatible), or *depositTokensWithPermit()* (for ERC20 tokens which also support EIP-2612). The appropriate deposit fee is deducted by the method, and paid 95% to users who have staked $ENSHROUD in the DAOPool contract, and 5% to the DAO Treasury address:



1b. Minting eNFTs from deposited balance

After transmitting the tokens to the smart contract for balance credit, the dApp must stipulate a breakdown of amounts and addresses to which the new eNFTs will be distributed. Amounts can be specified either as absolute quantities or as percentages, with anything left over implicitly allocated to the active account (i.e. *msg.sender*). Thus in the simplest degenerate case 100% of the deposited value comes back to the depositor in a single eNFT bearing an amount equal to the net deposit. It should be noted that while exact amounts of individual eNFTs will be concealed on the blockchain, their total must by necessity equal the amount of the debit made to the deposited balance. Every eNFT so generated directly from a deposit balance will bear a {generation=1} by definition. Nevertheless, an effect similar to a Taproot transaction may be achieved during a deposit, by specifying multiple payees.

Pictorial representation:



Let's examine the details of the steps diagrammed here. They are similar in the other cases.

First the dApp must select a MVO to which the list of eNFT amounts and addresses will be sent. This selection must be from among the currently enabled MVOs (per the MVOStaking contract's data). The recommended selection mechanism is by random number generation matched to enabled MVOs weighted by their staking quantum. That is, a given MVO with 2X as much $ENSHROUD staked should be chosen twice as often as one with X staked.

Sidebar 2:

> All communications between the dApp and MVO nodes is secured by a custom encryption layer, using an algorithm known as ECIES. In this algorithm the public key for communications listed in the MVOStaking contract's mapping for the selected MVO is used to generate a unique AES-256 encryption key for each individual message. It is a property of ECDSA encryption that, for any two keypairs K1, K2 in the SECP-256k1 curve space, it will be the case that: K1pub.mult(K2priv) == K2pub.mult(K1priv), where "mult()" refers to the multiplier function derived from the private key, not literal multiplication.
>
> To apply this, the dApp generates an ephemeral 256-bit ECDSA keypair and derives the multiplier function from the privkey. It then multiplies the MVO's posted pubkey by the ephemeral privkey to arrive at a 256-bit value. This value naturally becomes a 256-bit AES-256 encryption key, with randomness of the same degree as the ephemeral keypair. The dApp further generates and stores a second AES-256 key from a secure random source, which will be used to encrypt the reply. (This avoids the MVO needing to know the user's pubkey, which cannot be derived from their address.) The dApp builds its message (including the reply key),

and encrypts it with AES-256 in GCM mode with a 128-bit initialization vector and a 128-bit tag (both transmitted alongside the ciphertext). The ephemeral pubkey is supplied in cleartext along with the encrypted data. In order to inform the MVO which chain is involved (and thus which ECDSA private key to use), the POST must include the HTTP header value "encrchain", set to the chainId in decimal. (Note this implies the MVO must support OPTIONS preflights.)

The MVO decapsulates the passed pubkey and uses its own private key and multiplier function to compute the same 256-bit AES key value. It then decrypts the payload (recall the initialization vector and checksum tag were supplied along with the ciphertext) and then parses the dApp's request. Certain error responses may not be encrypted, but any normal reply will always be AES-256 encrypted using the dApp's indicated reply key.

This technique is used in lieu of standard *https://* URLs, quite simply because HTTPS is no longer safe for use in privacy applications. This is not due to flaws in SSL/TLS, but to deficiencies in the fundamental PKI (public key infrastructure) related to domains, certificates and CAs (certificate authorities). If this is news to you, or sounds like an outlandish claim, we suggest that you read this article. The writer in fact naively assumes that major world governments are not *already* conducting MITM attacks on https-secured sites by means of certs issued by compliant captive CAs, which will not be flagged by any modern browser.

Apart from which, communicating with the MVO layer via CA-signed site certificates would require all MVO nodes to obtain such certs, along with associated domains. This would substantially widen the attack surface for node operators. Since Enshroud MVOs are analogous to "Layer 2" nodes, they should follow the same standards for other miners and validators, and utilize bare IPv4/6 addresses over plain HTTP. To enable this while still securing communications, Enshroud took the trouble to deploy on both sides a much more secure method (ECIES) already in use elsewhere in the crypto universe, and adequately supported in library software. *In our considered opinion, other serious privacy projects should do likewise.*

N.B.: The Enshroud dApp will be served over HTTP. Normally http contexts permit http sub-connections. However, users who have configured their browsers to reject "insecure" connections categorically in all contexts will need to allow an exception to use the Enshroud dApp, unless (as recommended) they are accessing the dApp via a IPNS URL. Browsers (e.g. Chrome 121+) have begun to encourage this blanket setting rejecting HTTP.

The message payload will conform to this schema, which assumes context of EIP-712 client signatures:

Mint From Deposit Message to MVO – Schema

```
{{"message": { "requestJson": {
        "chainId": {
                "type": "integer",
                "description": "the chain ID of the blockchain (identifies smart contract)"
        },
        "opcode": {
                "type": "string",
                "description":"nature of operation, one of deposit | spend | withdraw"
        },
```

“sender”: {
“type”: “address”,
“description”: “the wallet address making the deposit, i.e. *msg.sender*”
},
“asset”: {
“type”: “address”,
“description”: “the type of token deposited, e.g. contract address for WBTC”
},
“amount”: {
“type”: “integer”,
“description”: “total deposited balance used (output eNFTs must not exceed this), in wei”
},
“replyKey”: {
“type”: “string”,
“description”: “AES-256 key to encrypt reply, in Base64 format (will be 22 chars)”
},
“payees”: {
“type”: “array”,
“description”: “list of payees to be issued newly minted eNFTs”,
[
{“payeeLabel”: {
“type”: “string”,
“description”: “designation of payee, by default payeeNNN”
},
{“payeeSpec”: {
“address”: {
“type”: “address”,
“description”: “account address of payee”
},
“amount”: {
“type”: “integer”,
“description”: “amount to issue eNFT (wei in decimal)”
},
“units”: {
“type”: “string”,
“description”: “empty if an absolute quantity, % if a percentage (that is, the amount can be specified either as a figure <= {requestJson.amount} or as a fraction of {requestJson.amount})”
},
“rand”: {
“type”: “string”,
“description”: “if present, 16 bytes of Base64-encoded random data, generated by the dApp as a salt (22 chars)”
},
“memo”: {
“type”: “string”,
“description”: “optional baggage field: an identifying note from the {sender} (limited to 1024 chars)”

```
                        }
                    }},
                    {“payeeLabel”: “payee002”: { … }},
                    {“payeeLabel”: “payee003”: { … }},
                    …
                ]
        },
}}},
        “signature”: {
                “type”: “string”,
                “description”: “the privkey EIP-712 signature of the {sender} on the above fields, 130 hex char digest”
        }
}
```

The simplest case would be a single {payee001} in the array, with an empty {address} indicating that the sole eNFT to be issued against the {amount} will be minted to the sender’s own address for the full value, specified as 100%. Alternatively, the {amount} can match {requestJson.amount} in wei.

It will be observed that more complex cases implement a private batch payment scenario, in which a single quantity of deposited {asset} tokens can be “fanned out” to a number of other wallets or contracts. (The latter assuming that contracts existed which know what to do with Enshroud eNFTs; see section IX below.) The EnshroudProtocol smart contract enforces a maximum of 20 output eNFTs at once. Obviously gas costs may impose a smaller practical limit. Client dApps will estimate the gas required before attempting complex operations. (MVOs will tolerate anything up to 20 inputs/outputs.)

Notice also that a deposit mint can be done on one’s own behalf first, taking back a single equal eNFT; whereupon that eNFT can then be used as an input to a spend operation, as described below.

The entire EIP-712 signed message block must then be POSTed by the dApp to the MVO’s interface URL as the payload contained in a x-*www-form-urlencoded* form variable called “*depositspec*”, with header parameter encrchain=chainId. (This connection is HTTP, routed through an autossh proxy but end-to-end encrypted with the MVO as described above in Sidebar 2.)

MVO Algorithm – Minting from Deposits

On receipt of a *depositspec* request, the MVO server will perform the following steps:

- Verify the EIP-712 signature of the {sender} on the JSON data.
- Verify the format and consistency of the message, e.g. all fields present, amounts add up, etc.
- For all output eNFTs, query an Auditor for a randomly generated AES-256 key, indexed by hash “keccak256(chainId+ID+address)”. (The Auditors will store this hash:key entry in a persistent map, and make it available to MVOs only, when authorized by owning user signatures.)
- Construct an “Operations Block” for the dApp, in JSON format as described below. This enumerates all the new eNFTs which must be created, including their full metadata, already encrypted with the appropriate symmetric AES keys for each output eNFT.

- Construct an argsHash for the Operations Block encapsulating the arguments which will be passed to the SC, and sign this with its private key. (This will be verified by the SC.) For a deposit, this *argsHash* is constructed as:

  keccak256(abi.encode(uint256 amount, uint256[] inputIds, uint256[] detailsHashes))

- Sign the entire Operations Block with its private key. (This will be checked by the dApp.)
- Construct a version of the Operations Block with the asset / amount values in the clear, and sends it to the Auditor encrypted with the Auditor's public communications key. (The Auditor nodes are described in section VI below.)
- Collect additional counter signatures of other MVOs, by following this procedure:
  - Select a committee from (N-1) of M fellow MVOs, from those currently enabled and online. The selection algorithm is random and staking-weighted (i.e. same as the dApp's).
  - For each committee MVO, forward them a copy of the original payload from the dApp, plus its own Operations Block that it generated. (These forwards and replies are always encrypted with the recipient MVO's public key and signed by the sender MVO's private key, and sent to *ws://* URLs over a VPN operating between all of the MVOs.)
  - Each committee MVO performs the same validation steps and affixes its own *argsHash* and overall signatures onto the Operations Block if it agrees with the lead MVO's results.
  - Collate the replies from its partner MVOs in the committee.
- Return an aggregate reply to the dApp consisting of the generated Operations Block, together with the independent signatures of all *N* committee MVOs including itself. The reply is AES256-GCM encrypted using the indicated *replyKey* passed by the dApp.
- On any error, return a suitable HTTP error response (code 400, 500, 504, etc.) with a message.
- Generate encrypted receipts (lead-MVO only), and sign/upload them to the receipt storage configured for that chain only if/when the transaction minting the eNFT IDs are actually minted and appear on the blockchain. (Receipts whose eNFTs never appear are eventually purged.)

On a success response, the dApp must invoke the *mintENFTsFromDeposit()* method in the SC, passing the required parameters (including all MVO *argsHash* signatures) as calldata. Upon successful processing the SC will debit the balance pre-deposited by {msg.sender} to fund the eNFT issuance, and mint all the new eNFTs via a *TransferSingle* event, emitting their encrypted metadata via *URI* events.

The eNFT Operations Block for Minting from Deposits – Schema Definition

For a deposit mint operation, the Operations Block constructed by an MVO and co-signed by its peers will adhere to the following JSON schema:

```
{"OperationsBlock": {
        "type": "object",
        "description": "dApp instructions block from MVO layer, plus signatures",

        "MVOsigned": {
                "type" : "object",
                "description": "the complete JSON object as signed by each of the MVOs",

                "opcode": {
                        "type": "string",
```

“description”: “one of these three values: **deposit** | spend | withdraw”
},
“argsHash”: {
“type”: “string”,
“description”: “hash of the arguments which will be passed to the SC (bytes32 as base64)”
},
// Here we will show only the elements required for a “deposit” opcode:
“amount”: {
“type”: “integer”,
“description”: “total of eNFT amounts to be issued (as hex)”
},
// all eNFTs to be minted must be of this asset type:
“asset”: {
“type”: “string”,
“description”: “the contract address to be deposited (such as WETH, WBTC)”
},
“outputs”: {
“type”: “array”,
“description”: “the opaqued details of each eNFT to be minted”,
[
{“001”: {
“address”: {
“type”: “address”,
“description”: “if empty, implicitly the address of (*msg.sender*); otherwise a valid payable address”
},
“id”: {
“type”: “string”,
“description”: “the unique ID for the new eNFT, assigned randomly by the lead MVO, zero-padded to 64 hex digits”
},
// this goes into the SC’s (ID => detailsHash) mapping
“hash”: {
“type”: “string”,
“description”: “detailsHash for the eNFT, constructed as keccak256(abi.encode(address,id,asset,amount,rand))”
},
“metadata”: {
“type”: “object”,
“description”: “full JSON metadata, with {enshrouded} element encrypted to allocated AES key, in base64 format”
},
}},
{“002”: { … }},
{“003”: { … }},
…
{“nnn”: { . . . }}
]

```
            }
        },
        “signatures”: {
            “type”: “array”,
            “description”: “successive signatures for N of M MVOs who signed {MVOsigned}”,
            [
                {“001”: {
                    “signer”: {
                        “type”: “string”,
                        “description”: “the ID of the signing MVO, e.g. MVO-004”
                    },
                    “sig”: {
                        “type”: “string”,
                        “description”: “the private key signature, a 130 hex char digest”
                    },
                    “argsSig”: {
                        “type”: “string”,
                        “description”: “signature on the {argsHash}, 130 hex char digest”
                    }
                }},
                {“002”: { … }},
                {“003”: { … }},
                …
                {“nnn”: { … }}
            ]
        }
}}
```

This {OperationsBlock} is supplied to the dApp, which verifies it and then uses it to build the arguments to be passed to the SC as calldata. To prevent the dApp from making modifications to the calldata, the SC will verify that the passed {argsHash} value matches the hash calculated from the other passed parameters, and then check each MVO's signature on the hash. Essentially, the MVO layer builds and signs the instructions for minting and recording the eNFTs, and the depositor's wallet adds its own signature and forwards these instructions to the SC. To prevent the MVOs from doing anything untoward, before submitting to the SC the dApp must construct each output eNFT's *detailsHash* from known data (as entered by the user) and verify that all hash values are as expected.

The {AuditorBlock} (not shown) is the same except that the actual {owner}, {asset}, {amount}, and {rand} values are included for each payee item in the array, in addition to the {detailsHash}. Also, the {AuditorBlock} is signed and sent to it by each MVO individually as it processes requests. Thus the Auditor will receive multiple copies of the same {AuditorBlock}, one from each of *N* MVOs who are members of the committee, probably before the transaction is ever submitted on-chain. (See sec. VI.)

Smart Contract Algorithm – Minting from Deposits

On invocation of *mintENFTsFromDeposit()*, the SC will perform the following steps:

- Confirms that the {asset} account balance deposited by {sender} meets or exceeds the total given in the arguments. Debits balance in *assetBalances* mapping appropriately.
- Parses, hashes, and verifies the calldata parameters, calculates and confirms the passed *argsHash* value, and verifies each MVO's *argHash* signature, plus overall sufficiency of signatures (at least *N*).
- Verifies that the IDs of all eNFTs to be issued have not been added to the greylist by an Auditor.
- Verifies that no eNFT currently exists with the same ID (IDs must guarantee uniqueness).
- Manipulates the internal mapping of hashed eNFT data (see below).
- Records the eNFT ID in the *enftUnlockTime* mapping, calculating block.number + conf blocks. (This is the block number at which the eNFT becomes usable as a transaction input.)
- Sets *balanceOf[to_address][id]* to 1 (in accordance with ERC-1155) for each Id.
- Issues *TransferSingle* mint events for all new eNFTs (i.e. transfers where *_from* = 0x0).
- Per the safe transfer requirements, calls *ERC1155TokenReceiver.onERC1155Received()* if any *_to[]* eNFT recipient address was itself a smart contract.
- Issues URI log events for each new eNFT, giving the full (enc.) metadata for each new eNFT.
- In the MVOStaking contract, credits the MVOs who signed the OB with transaction rewards (lead MVO gets 3 points, committee members each get 1 point).
- On any error or inconsistency, throw exception and revert the entire transaction.

Drill Down: Tracking Hashed eNFT Data

It would obviously be self-defeating for privacy if the SC's state data contained all of the amounts of the existing eNFTs, or if the amounts were placed into the mempool via open function arguments in calldata. However, it is still essential that the blockchain itself be utilized to prevent double-spending or counterfeiting of circulating eNFTs. For this reason, the SC will maintain a mapping of all extant eNFT IDs against the hash of their pertinent details. To ensure uniqueness of hashes, the details hash is constructed thusly:

bytes32 detailsHash = keccak256(abi.encode({address},{id},{asset},{amount},{rand}));

(N.B.: a Keccak256 hash is an EVM-specific implementation of a SHA-3 hashing algorithm.) That is, the hash of the eNFT's owning address, ID, the asset type, the quantity, and the random salt picked by the dApp. The SC's mapping of IDs to hashes is defined as:

mapping(uint256 => bytes32) internal detailsHash;

The mapping is marked *internal* to deter attempts to solve for the details of an eNFT through hash generation on candidate values. The *detailsHash* value is supplied in the eNFT Operations Blocks for *deposit* and *spend* opcodes for all output eNFTs. The multiple signatures of MVOs will attest to the exact correspondence between the generated details hashes and the constituent owner/asset/amount values. For the *withdraw* opcode, the actual details will be provided in the calldata, and the SC will compute the details hash itself and confirm it matches the hash value in the mapping for each eNFT being redeemed. This compromises no privacy, since all eNFTs being redeemed must belong to the address making the withdrawal, and the successful result will be a public transfer of deposited tokens for the full amount (less any withdrawal fee defined) back to the on-chain balance of the very same address, which is a visible state change.

Sidebar 3:

> It might be objected that an even more secure mechanism could be provided by using zk-SNARKs instead of simple hash verification. In this case the actual proofs could be stored in the mapping (or in a L2 Merkle tree, as Aztec Network does it), and the operations (simple addition/subtraction) could be computed by the MVOs, with the proofs provided to the SC in the calldata. (The MVOs would then become "snarkers" in the parlance of Mina, or sMPCs the way Liminal does it.) However the Enshroud system is intended mainly for deployment on blockchains where nifty capabilities like zero-knowledge proofs and homomorphic encryption do not natively exist. On such naïve chains the gas costs related to verifying proofs are prohibitive, as Railgun and Offshift have discovered. Hashes are ubiquitous and fast, and the system architecture requires computation of eNFT *detailsHash* values in the EVM only when burning eNFTs during a withdrawal.

2. Making Private Spends Using eNFTs

The benefit of spending an eNFT as opposed to the tokens which it represents is that the amount and asset type of the spend is kept private, even on a public blockchain. A potential side benefit is that several spends can be made privately in parallel (similar to a "CoinJoin" transaction). But as noted above, gas limits will gate the practical application of this, although batching could also reduce overall costs by reducing the gross number of transactions.

The simplest spend case is that Bob has one or more eNFTs for a particular token asset in his wallet, and wishes to make a spend to Alice's address. Instead of transferring a portion of his balance from his account to Alice's, as with fungible tokens, an eNFT spend will use his existing eNFT(s) as inputs, and produce two outputs: one new eNFT minted for Alice in the desired amount, and another eNFT returned to Bob as change. The input eNFT(s) Bob used to fund his transfer are all burned. The output eNFTs are all new, including the one for Bob's change. In this sense eNFTs act like UXTOs on Bitcoin. The obvious difference is that no one except the owner can see the value of an eNFT.

Sidebar 4:

> The {generation} field of output eNFTs is always the *lowest* generation of the input eNFTs plus one. So if Bob's sole input eNFT had {generation=5}, the new one for Alice, and his change, would both have {generation=6}. The higher its generation, the farther away the eNFT is from a deposit/mint operation. (All eNFTs minted from a deposit start out with {generation=1}.) This implies that one is spending value that has been involved in a longer series of opaque transactions, making it harder to draw conclusions about values and ownership. Therefore, to maximize privacy dApps should select input eNFTs for spend operations with a bias for the *lowest* generation available (i.e. youngest first). Similarly, for withdraw operations (described below) dApps should select inputs with a bias for the *highest* generation available (i.e. oldest first). This procedure will tend to increase generation values over time, aimed at achieving the hightest possible distance between on-chain deposits and eventual withdrawals. The MVO system will in any case process the supplied inputs exactly as selected by the user.

The principal difference between a spend and the minting mechanism described in the previous section is that in addition to specifying {outputs} describing eNFTs to be minted, a set of {inputs} representing existing eNFTs owned by the spender and used to fund the spend(s) must also be specified.

Thus the data block sent to the randomly selected lead MVO looks like this (assumes EIP-712 sigs):

Spend Message to MVO – Schema

{{“message”: { “requestJson”: {
“chainId”: {
“type”: “integer”,
“description”: “the chain ID of the blockchain (identifies smart contract)”
},
“opcode”: {
“type”: “string”,
“description”:”nature of operation, one of deposit | **spend** | withdraw”
},
“sender”: {
“type”: “address”,
“description”: “the wallet address making the spend, i.e. *msg.sender*”
},
“replyKey”: {
“type”: “string”,
“description”: “AES-256 key to encrypt reply, in Base64 format (will be 22 chars)”
},
“inputs”: {
“type”: “array”,
“description”: “list of eNFTs to be used to fund the transfer(s)”
[
{“inputLabel”: “input001”, “inputSpec”: {
“enshrouded”: {
“type”: “object”,
“description”: “the complete {enshrouded} block, decrypted but with the MVOs signature still attached; note this contains {id}, {asset}, {amount}, {rand}, and {generation}”
},
“key”: {
“type”: “string”,
“description”:”the base64-encoded AES key which decrypts the URI metadata for this eNFT”
}
}},
{“inputLabel”: “input002”: { … }},
{“inputLabel”: “input003”: { … }},
...
]
},
“payees”: {
“type”: “array”,
“description”: “list of payees to be issued newly minted eNFTs”,
[
{“payeeLabel”: {

```
                        “type”: “string”,
                        “description”: “designation of payee, by default payeeNNN”
                    },
                    {“payeeSpec”: {
                        “address”: {
                            “type”: “address”,
                            “description”: “account address of payee”
                        },
                        “amount”: {
                            “type”: “integer”,
                            “description”: “amount to issue eNFT (wei in decimal)”
                        },
                        “units”: {
                            “type”: “string”,
                            “description”: “empty if an absolute quantity, % if a percentage (that is, the amount can be specified either as a figure <= {inputTotal} or as a fraction of {inputTotal})”
                        },
                        “rand”: {
                            “type”: “string”,
                            “description”: “if present, 16 bytes of Base64-encoded random data, generated by the dApp as a salt (22 chars)”
                        },
                        “memo”: {
                            “type”: “string”,
                            “description”: “optional baggage field: an identifying note from the {sender} (limited to 1024 chars)”
                        }
                    }},
                    {“payeeLabel”: “payee002”: { … }},
                    {“payeeLabel”: “payee003”: { … }},
                    …
                ]
        }
}}},
        “signature”: {
            “type”: “string”,
            “description”: “the privkey EIP-712 signature of the {sender} on the above fields, 130 hex char digest”
        }
}
```

The simplest case would be one element in the {inputs} array, and one element in the {payees} array. In this case the change eNFT would be implied by any difference in value. Notice that because the full decrypted details are provided in the clear, the MVOs can independently confirm they have not been modified by checking the incorporated MVO {signature}, and confirm via the event log that the eNFT has not previously been burned.

It is legitimate for a given address to appear multiple times in the {payees} array. This could be done to break up a larger value eNFT into smaller ones (like breaking a $100 into five $20s or two $50s). It

could also be a result of the inclusion of multiple asset types in both inputs and outputs, by which a payee is simultaneously paid some of one token in addition to some of another token. It is legitimate for the {sender} address to appear in the {payees} output list, any number of times. While there must be at least one element to the {payees} array (other than the implicit entry representing change), that payee address could even be that of the {sender}; although this would serve only to split or merge existing eNFTs, and thus bump up the {generation} values. A payee {amount} of 0 is allowed, which would create decoy outputs (for example to make it appear that the {sender} got change back, or to send a private message to the payee).

The entire signed message block must then be encrypted using ECIES (as described in Sidebar 2 above) and POSTed by the dApp to the MVO's interface URL as the payload contained in a *www-form-encoded* form variable called "*spendspec*", along with the header value encrchain=chainId.

MVO Algorithm – Spends

In response to receipt of a *spendspec* JSON block, the selected MVO will perform these steps:

- Verify the EIP-712 signature of the {sender} on the JSON data.
- Verify the format and consistency of the message, e.g. all fields present, amounts add up, etc.
- Verify the MVO signature on each eNFT listed in the {inputs} array.
- Verify that each input eNFT is still validly circulating. To be validly circulating, the eNFT's *balanceOf()* mapping must be != 0 for the {payer} address and it must not appear in a TransferSingle event transferring it to 0x0 (i.e. not burned).
- Verify that each input eNFT's original on-chain URI metadata successfully decrypts using the key retrieved from the Auditor as key server (with the dApp's supplied key as a fallback) and matches the dApp's supplied values.
- For all output eNFTs, query an Auditor for a randomly generated AES-256 key, indexed by hash "keccak256(chainId+ID+address)". (The Auditors will store this hash:key entry in a persistent map, and make it available to MVOs only, when authorized by owning user signatures.)
- Construct an "Operations Block" for the dApp, in JSON format as described below. This enumerates all the old eNFTs which must be burned, plus the new eNFTs which must be minted, including their full metadata, already encrypted with the appropriate symmetric AES keys for each output eNFT.
- Construct an *argsHash* for the Operations Block encapsulating the arguments which will be passed to the SC, and sign this with its private key. (This will be verified by the SC.) For a spend, this *argsHash* is constructed as:

  keccak256(abi.encode(uint256[] inputIds, uint256[] outputIds, uint256[] detailsHashes))

- Sign the entire Operations Block with its private key. (This will be checked by the dApp.)
- Construct a version of the Operations Block with the asset / amount values in the clear, and sends it to the Auditor encrypted with the Auditor's public communications key.
- Collect additional counter signatures of other MVOs, by following this procedure:
  - Select a committee from (N-1) of M fellow MVOs, from those currently enabled and online. The selection algorithm is random and staking-weighted (i.e. same as the dApp's).
  - For each committee MVO, forward them a copy of the original payload from the dApp, plus its own Operations Block that it generated. (These forwards and replies are always encrypted with the recipient MVO's public key and signed by the sender MVO's private key, and sent to *ws://* URLs over a VPN operating between all of the MVOs.)

    - Each committee MVO performs the same validation steps and affixes its own *argsHash* and overall signatures onto the Operations Block if it agrees with the lead MVO's results.
    - Collate the replies from its partner MVOs in the committee.
- Return an aggregate reply to the dApp consisting of the generated Operations Block, together with the independent signatures of all *N* committee MVOs including itself. The reply is AES256-GCM encrypted using the indicated *replyKey* passed by the dApp.
- On any error, return a suitable HTTP error response (code 400, 500, 504, etc.) with a message.
- Generate encrypted receipts (lead-MVO only), and sign/upload them to the receipt storage configured for that chain only if/when the transaction minting the eNFT IDs are actually minted and appear on the blockchain. (Receipts whose eNFTs never appear are eventually purged.)

On a success response, the dApp must invoke the *spendENFTs()* method in the SC, passing the parameters drawn from the Operations Block and all MVO *argsHash* signatures thereon as calldata. Note there are **no fees** (outside of gas, naturally) collected for executing a *spend* transaction. Upon successful processing the SC will mint and burn all the new/old eNFTs via a *TransferSingle* event, emitting the encrypted metadata of newly minted ones via *URI* events.

The eNFT Operations Block for Spends – Schema Definition

The OB schema for a spend operation looks like this:

```
{“OperationsBlock”: {
        “type”: “object”,
        “description”: “SC instructions block from MVO layer, plus signatures”,

        “MVOsigned”: {
                “type” : “object”,
                “description”: “the complete JSON object as signed by each of the MVOs”,

                “opcode”: {
                        “type”: “string”,
                        “description”: “one of these three values: deposit | spend | withdraw”
                },
                “argsHash”: {
                        “type”: “string”,
                        “description”: “hash of the arguments which will be passed to the SC (bytes32 as base64)”
                },
                // Here we will show only the elements required for a “spend” opcode:
                “inputs”: {
                        “type”: “array”,
                        “description”: “the opaqued details of each eNFT to be burned”,
                        [
                                {“001”: {
                                        // this must be in the SC’s (ID => detailsHash) mapping
                                        “id”: {
                                                “type”: “string”,
```

```
                                        “description”: “the unique ID for the input eNFT, zero-padded to 64 hex digits”
                                    },
                                    “hash”: {
                                        “type”: “string”,
                                        “description”: “detailsHash for the eNFT, constructed as keccak256(abi.encode(address,id,asset,amount,rand))”
                                    }
                                }},
                                {“002”: { … }},
                                {“003”: { … }}
                                ...
                            ]
                    },
                    “outputs”: {
                            “type”: “array”,
                            “description”: “the opaqued details of each eNFT to be minted”,
                            [
                                {“001”: {
                                    “address”: {
                                        “type”: “address”,
                                        “description”: “if empty, implicitly the address of payer (msg.sender); otherwise a valid payable address”
                                    },
                                    “id”: {
                                        “type”: “string”,
                                        “description”: “the unique ID for the new eNFT, assigned randomly by the lead MVO, zero-padded to 64 hex digits”
                                    },
                                    // this goes into the (ID => hash) mapping
                                    “hash”: {
                                        “type”: “string”,
                                        “description”: “detailsHash for the eNFT, constructed as keccak256(abi.encode(address,id,asset,amount,rand))”
                                    },
                                    “metadata”: {
                                        “type”: “string”,
                                        “description”: “full JSON metadata, with {enshrouded} element encrypted to allocated AES key, in base64 format”
                                    },
                                }},
                                {“002”: { … }},
                                {“003”: { … }},
                                …
                            ]
                    }
            },
            “signatures”: {
                    “type”: “array”,
```

```
            "description": "successive signatures for N of M MVOs who signed {MVOsigned}",
            [
                {"001": {
                    "signer": {
                        "type": "string",
                        "description": "the ID of the signing MVO"
                    },
                    "sig": {
                        "type": "string",
                        "description": "the actual private key signature, a 130 char digest"
                    },
                    "argsSig": { and matches the dApp's supplied values including MVO signature.
                        "type": "string",
                        "description": "signature on the {argsHash}, 130 hex char digest"
                    }
                }},
                {"002": { … }},
                {"003": { … }},
                …
                {"nnn": { … }}
            ]
        }
}};
```

This {OperationsBlock} is supplied to the dApp, which verifies it and then uses it to build the arguments to be passed to the SC as calldata. To prevent the dApp from making modifications to the calldata, the SC will verify that the passed {argsHash} value matches the hash calculated from the other passed parameters, and then check each MVO's signature on the hash. Essentially, the MVO layer builds and signs the instructions for minting and recording the eNFTs, and the depositor's wallet adds its own signature and forwards these instructions to the SC. To prevent the MVOs from doing anything untoward, before submitting to the SC the dApp must construct each output eNFT's *detailsHash* from known data (as entered by the user) and verify that all hash values are as expected.

The {AuditorBlock} (not shown) is the same except that the actual {owner}, {asset}, {amount}, and {rand} values are included for each payee item in the array, in addition to the {detailsHash}. Also, the {AuditorBlock} is signed and sent to it by each MVO individually as it processes requests. Thus the Auditor will receive multiple copies of the same AB, one from each of *N* MVOs who are members of the committee, probably before the transaction is ever submitted on-chain. (See section VI.)

Notice that the SC has no way to know whether the address/asset/amount values for output eNFTs which produced the corresponding *detailsHash* value are actually correct. This is the purpose of the MVO layer, which is discussed in more detail in section V below. For this reason any eNFTs intended for the spender (*msg.sender*), including change back, must be specified in the {outputs}, since no implicit remainder could be calculated.

Smart Contract Algorithm – Spends

On the *spendENFTs()* function invocation with the Operations Block, the SC will perform the following steps:

- Parses, hashes, and verifies the calldata parameters, calculates and confirms the passed *argsHash* value, and verifies each MVO's *argHash* signature, plus overall sufficiency of signatures (at least *N*).
- Verifies that each input ID has met its confirmation requirement as of the current block.
- Verifies that each input ID has not been added to the greylist mapping by an Auditor node.
- Verifies that *balanceOf[msg.sender][ID]* is non-zero for each input ID.
- Deletes the *detailsHash* mapping entry and *enftUnlockTime* mapping entry for each input ID.
- Burns all input eNFTs by emitting *TransferSingle* events (i.e. transfers where *_to* = 0x0).
- Verifies that the IDs of all output eNFTs have not been added to the greylist by an Auditor.
- Verifies that no eNFT currently exists with the same ID as an output (IDs must guarantee uniqueness).
- Adds the passed *detailsHash* value to the mapping for each output ID.
- Records the eNFT ID in the *enftUnlockTime* mapping, calculating block.number + conf blocks. (This is the block number at which the new eNFT becomes usable as a transaction input.)
- Sets *balanceOf[to_address][id]* to 1 (in accordance with ERC-1155) for each output Id.
- Issues *TransferSingle* mint events for all new eNFTs (i.e. transfers where *_from* = 0x0).
- Per the safe transfer requirements, calls *ERC1155TokenReceiver.onERC1155Received()* if any *_to[]* eNFT recipient address was itself a smart contract.
- Issues URI log events for each new eNFT, giving the full (enc.) metadata for each new eNFT.
- In the MVOStaking contract, credits the MVOs who signed the OB with transaction rewards (lead MVO gets 3 points, committee members each get 1 point).
- On any error or inconsistency, throw exception and revert the entire transaction.

3. Making Withdrawals By Redeeming eNFTs

As noted above, this operation comes with no expectation of increased privacy. This is because the account making the withdrawal must own all of the input eNFTs, which must all represent the same asset being withdrawn from the contract, and total up to at least the amount withdrawn. Therefore, all of the operations performed are done "in the clear," with actual amounts disclosed in the calldata submitted to the mempool. Clients are nonetheless required to submit their transaction first to the MVO layer, in order to offload additional verification processing from the SC, saving gas and improving security.

The principal difference between a spend described in the previous section and a withdrawal is that instead of specifying {outputs} describing eNFTs to be minted, an {asset} and {amount} are given. Naturally the {inputs} representing existing eNFTs owned by the spender used to fund the withdrawal must also be specified. If there is any {inputs} overage, a singleton "change" eNFT will be generated representing the difference and returned to the {sender}.

Thus the data block sent to the randomly selected lead MVO looks like this (assumes EIP-712 sigs):

Withdraw Message to MVO – Schema

{{“message”: { “requestJson”: {
“chainId”: {
“type”: “integer”,
“description”: “the chain ID of the blockchain (identifies smart contract)”
},
“opcode”: {
“type”: “string”,
“description”:”nature of operation, one of deposit | spend | **withdraw**”
},
“sender”: {
“type”: “address”,
“description”: “the wallet address making the withdrawal, i.e. *msg.sender*”
},
“asset”: {
“type”: “string”,
“description”: “the token symbol of the asset being withdrawn (all eNFTs in {inputs} must correspond to this type)”
},
“amount”: {
“type”: “integer”,
“description”: “the total amount being withdrawn (in decimal; the sum of the eNFTs in {inputs} must be >=)”
},
“replyKey”: {
“type”: “string”,
“description”: “AES-256 key to encrypt reply, in Base64 format (will be 22 chars)”
},
“inputs”: {
“type”: “array”,
“description”: “list of eNFTs to be used to fund the withdrawal”
[
{“inputLabel”: “input001”, “inputSpec”: {
“enshrouded”: {
“type”: “object”,
“description”: “the complete {enshrouded} block, decrypted but with the MVOs signature still attached; note this contains {id}, {asset}, {amount}, {rand}, and {generation}”
},
“key”: {
“type”: “string”,
“description”:”the base64-encoded AES key which decrypts the URI metadata for this eNFT”
}
}},
{“inputLabel”: “input002”: { … }},
{“inputLabel”: “input003”: { … }},

```
                    ...
                ]
        },
}}},
        "signature": {
                "type": "string",
                "description": "the privkey EIP-712 signature of the {sender} on the above fields, 130 hex char digest"
        }
}
```

The simplest case would be one element in the {inputs} array, equaling the {amount}. In this case no change eNFT would exist. Notice that because the full decrypted details are provided in the clear, the MVOs can independently confirm they have not been modified by checking the incorporated MVO {signature}, and confirm that each eNFT has not previously been burned. The SC will compute and validate the *detailsHash* value (normally it cannot do this because all the datums are not supplied).

The entire signed message block must then be POSTed by the dApp to the MVO's secure interface URL as the payload contained in a *www-form-encoded* form variable called "*withdrawspec*", along with the POST header value encrchain=chainId.

In response to receipt of a *withdrawspec* JSON block, the selected MVO will perform these steps:

MVO Algorithm – Withdrawals

- Verify the EIP-712 signature of the {sender} on the JSON data.
- Verify the format and consistency of the message, e.g. all fields present, amounts add up, etc.
- Verify the MVO signature on each eNFT listed in the {inputs} array.
- Verify that each input eNFT is still validly circulating. To be validly circulating, the eNFT's *balanceOf()* mapping must be != 0 for the {sender} address and it must not appear in a TransferSingle event transferring it to 0x0 (i.e. not burned).
- Verify that each input eNFT's original on-chain URI metadata successfully decrypts using the key retrieved from the Auditor as key server (with the dApp's supplied key as a fallback) and matches the dApp's supplied values.
- If there is an output eNFT, query an Auditor for a randomly generated AES-256 key, indexed by hash "keccak256(chainId+ID+address)". (The Auditors will store this hash:key entry in a persistent map, and make it available to MVOs only, when authorized by owning user signatures.)
- Construct an "Operations Block" for the dApp, in JSON format as described below. This enumerates all the old eNFTs which must be burned, plus the new eNFT which must be minted (if any), including its full metadata, already encrypted with the appropriate symmetric AES key.
- Construct an *argsHash* for the Operations Block encapsulating the arguments which will be passed to the SC, and sign this with its private key. (This will be verified by the SC.) For a burn/withdraw, this *argsHash* is constructed as:

  keccak256(abi.encode(uint256[] inputIds, uint256 amount, uint256[] outputIds, uint256[] detailsHashes))

  where the *outputIds* and *detailsHashes* are length 1 if a change eNFT is needed, else absent.

- Sign the entire Operations Block with its private key. (This will be checked by the dApp.)
- Construct a version of the Operations Block with the asset / amount values in the clear, and sends it to the Auditor encrypted with the Auditor's public communications key.
- Collect additional counter signatures of other MVOs, by following this procedure:
  - Select a committee from (N-1) of M fellow MVOs, from those currently enabled and online. The selection algorithm is random and staking-weighted (i.e. same as the dApp's).
  - For each committee MVO, forward them a copy of the original payload from the dApp, plus its own Operations Block that it generated. (These forwards and replies are always encrypted with the recipient MVO's public key and signed by the sender MVO's private key, and sent to *ws://* URLs over a VPN operating between all of the MVOs.)
  - Each committee MVO performs the same validation steps and affixes its own detailsHash and overall signatures onto the Operations Block if it agrees with the lead MVO's results.
  - Collate the replies from its partner MVOs in the committee.
- Return an aggregate reply to the dApp consisting of the generated Operations Block, together with the independent signatures of all *N* committee MVOs including itself. The reply is AES256-GCM encrypted using the indicated *replyKey* passed by the dApp.
- On any error, return a suitable HTTP error response (code 400, 500, 504, etc.) with a message.
- Generate encrypted receipts (lead-MVO only), and sign/upload them to the receipt storage configured for that chain only if/when the transaction minting the eNFT IDs are actually minted and appear on the blockchain. If there's a change eNFT, this will key on that eNFT being minted. If not, receipt finalization will key on input eNFTs being burned. (Receipts whose upload is never triggered will eventually be purged.)

On a success response, the dApp must invoke the appropriate withdraw method in the SC, passing the eNFT Operations Block and all MVO signatures thereon as calldata. Note there is typically a fee collected for executing a *withdraw* transaction. This fee will be deducted from the {withdrawTotal} paid out by the SC.

The eNFT Operations Block for Withdrawals – Schema Definition

The OB schema for a withdrawal operation looks like this:

```
{"OperationsBlock": {
        "type": "object",
        "description": "SC instructions block from MVO layer, plus signatures",

        "MVOsigned": {
                "type" : "object",
                "description": "the complete JSON object as signed by each of the MVOs",

                "opcode": {
                        "type": "string",
                        "description": "one of these three values: deposit | spend | withdraw"
                },
                "argsHash": {
                        "type": "string",
                        "description": "hash of the arguments which will be passed to the SC (bytes32 as
base64)"
```

```
// Here we will show only the elements required for a “withdraw” opcode:
“sender”: {
	“type”: “address”,
	“description”: “the address to pay {asset} to (normally msg.sender)”
},
“asset”: {
	“type”: “string”,
	“description”: “contract address of token being withdrawn, e.g. WETH, WBTC”
},
“amount”: {
	“type”: “integer”,
	“description”: “the total amount of {asset} to be credited to the {sender} balance within the contract (in hex), in the appropriate precision (before any withdrawal fee)”
},
“inputs”: {
	“type”: “array”,
	“description”: “the clear details of each eNFT to be burned”,
	[
		{“001”: {
			// these 4 items (plus {sender}) support verification vs. the
			// (ID => hash) mappings
			“id”: {
				“type”: “string”,
				“description”: “the unique ID for the input eNFT, zero-padded to 64 hext digits”
			},
			“asset”: {
				“type”: “string”,
				“description”: “the token type/symbol represented by this eNFT (must match {asset})”
			},
			“rand”: {
				“type”: “string”,
				“description”: “the random salt value of this eNFT”
			},
			“amount”: {
				“type”: “string”,
				“description”: “the quantity value of this eNFT, in decimal”
			}
		}},
		{“002”: { … }},
		{“003”: { … }},
		...
	]
},
“outputs”: {
	“type”: “array”,
	“description”: “the clear details of each eNFT to be minted”,
```

```
                    // N.B.: if no change eNFT required, this array can be empty (no payees)
                    [
                        {“001”: {
                            // N.B.: address is implicitly that of {sender}
                            “id”: {
                                “type”: “string”,
                                “description”: “the unique ID for the new eNFT, assigned randomly by the lead MVO, zero-padded to 64 hex digits”
                            },
                            // N.B.: asset is implicitly that of {asset}
                            “amount”: {
                                “type”: “string”,
                                “description”: “amount of change eNFT, in decimal”
                            },
                            // N.B.: the lead MVO will generate the value for {rand}
                            “rand”: {
                                “type”: “string”,
                                “description”: “16 byte random salt, base64 encoded”
                            },
                            // keccak256(abi.encode(sender,id,asset,amount,rand)) goes  into
                            // (ID => hash) map
                            “metadata”: {
                                “type”: “string”,
                                “description”: “full JSON metadata, with {enshrouded} element encrypted with assigned AES key, in base64 format”
                            },
                        }},
                        {“002”: { … }},
                        {“003”: { … }},
                        …
                    ]
                }
            },
            “signatures”: {
                “type”: “array”,
                “description”: “successive signatures for N of M MVOs who signed {MVOsigned}”,
                [
                    {“001”: {
                        “signer”: {
                            “type”: “string”,
                            “description”: “the ID of the signing MVO”
                        },
                        “sig”: {
                            “type”: “string”,
                            “description”: “the actual private key signature, a 130 hex char digest”
                        },
                        “argsSig”: {
                            “type”: “string”,
```

```
                                        “description”: “signature on the {argsHash}, 130 hex char digest”
                                }
                        }},
                        {“002”: { … }},
                        {“003”: { … }},
                        …
                        {“nnn”: { … }}
                ]
        }
}}
```

It could be legitimate to have more than one payee in {outputs}, but since they are all being issued to the same party by way of change back from the withdrawal transaction (all of whose input amounts have been revealed), it’s probably not very useful; and so in practice it isn’t supported.

Smart Contract Algorithm – Withdraws

On seeing the withdraw function invocation with the Operations Block, the dApp will invoke the EnshroudProtocol contract’s *redeemENFTsAndWithdraw()* method which performs the following steps:

- Checks passed argument array lengths for consistency.
- Recreates and verifies the Operations Block *argsHash*, and confirms each MVO signature and overall sufficiency of signatures (*N of M*).
- Verifies that all input eNFTs are found in its mappings and are unburned, i.e. *balanceOf[msg.sender][ID] != 0.*
- Verifies all input eNFTs are confirmed (sufficient number of blocks has elapsed since mint).
- Verifies all input eNFTs have not been greylisted by an Auditor node.
- Verifies that the stored details hash for each input agrees with the hash constructed from the actual values presented by the client.
- Deletes the *detailsHash* mapping entry and *enftUnlockTime* mapping entry for each input ID.
- Burns all input eNFTs by emitting *TransferSingle* events (i.e. transfers where *_to* = 0x0).
- Sums the value of all burned inputs, and confirms it meets or exceeds the passed {amount}.
- Performs the following 8 steps if and only if an output change eNFT is specified:
- 1) Verifies that the ID has not been added to the greylist by an Auditor.
- 2) Verifies that no eNFT currently exists with the same ID (uniqueness).
- 3) Adds the passed *detailsHash* value to the mapping for the ID.
- 4) Records the ID in the *enftUnlockTime* mapping, calculating block.number + conf blocks. (This is the block number at which the new eNFT becomes usable as a transaction input.)
- 5) Sets *balanceOf[to_address][id]* to 1 (in accordance with ERC-1155) for the ID.
- 6) Issues *TransferSingle* mint events for the new eNFT (i.e. transfer where *_from* = 0x0).
- 7) Per the safe transfer requirements, calls *ERC1155TokenReceiver.onERC1155Received()* if the *_to[]* eNFT recipient address was itself a smart contract. (Unlikely since {sender} is a EOA.)
- 8) Issues URI log event for the new eNFT, giving the full (encrypted) metadata.
- Calculates the amount to withdraw net of fees paid to DAOPool and Treasury.
- Using SafeTransferLib methods, pays {sender}, DAOPool, and Treasury their respective amounts due of the {asset}.
- In the MVOStaking contract, credits the MVOs who signed the OB with transaction rewards (lead MVO gets 3 points, committee members each get 1 point).

- On any error or inconsistency, throw exception and revert the entire transaction.

Notice that in this case, alone among the three operations, the SC verifies the existence and hash correctness of each input eNFT, based on their actual asset/value/salt inputs which generate those hashes. The cost of this additional security is that no part of a withdrawal transaction is truly private. For this reason, supplying eNFT payees other than the {sender} is not supported.

This {OperationsBlock} is supplied to the dApp, which verifies it and then uses it to build the arguments to be passed to the SC as calldata. To prevent the dApp from making modifications to the calldata, the SC will verify that the passed {argsHash} value matches the hash calculated from the other passed parameters, and then check each MVO's signature on the hash. Essentially, the MVO layer builds and signs the instructions for minting and recording the eNFTs, and the depositor's wallet adds its own signature and forwards these instructions to the SC. To prevent the MVOs from doing anything untoward, before submitting to the SC the dApp must construct each output eNFT's *detailsHash* from known data (as entered by the user) and verify that all hash values are as expected.

The {AuditorBlock} (not shown) is the same except that the actual {owner}, {asset}, {amount}, and {rand} values are included for each payee item in the array, in addition to the {detailsHash}. Also, the {AuditorBlock} is signed and sent to it by each MVO individually as it processes requests. Thus the Auditor will receive multiple copies of the same AB, one from each of *N* MVOs who are members of the committee, probably before the transaction is ever submitted on-chain. (See section VI.)

4. Downloading Wallet eNFT Data

As described above, the metadata for eNFTs is encrypted using randomly generated AES-256 symmetric keys, and stored in a mapping of hashes vs. the Base64Url-encoded keys by the Auditors. (See section VI below for more details on Auditors.) The hash indexes are *keccak256(chainId+ID+ address),* where ID is that of the eNFT, and address is the owning wallet to which the eNFT was minted. Since this index of decryption keys is permanently maintained by the Auditors, there is no imperative need for Enshroud users to maintain copies of their own keys. However, as the adage "not your keys not your coins" has it, users may wish to take personal custody of the keys required to decrypt and access their eNFTs, for backup purposes against loss or damage of the Auditors' persistent map. (Note that the dApp's keys transmitted with {inputs} act as a fallback in case the Auditor keyring no longer has them available, as described above.)

To facilitate this task, the MVO layer's JSON interface supports a query which will download any or all of an account's eNFTs in plaintext format, together with the required decryption keys to decrypt each one from the copy permanently stored in the blockchain's event log. In order to separate message traffic related to such housekeeping procedures from traffic related to actual transactions, MVOs listen for wallet download requests (along with similar receipt requests described below) on a separate secure listen port. The JSON schema for wallet contents requests, delivered encrypted with ECIES via an http POST payload called *walletDownload,* along with the POST header value encrchain=chainId, is the following (assumes EIP-712 signatures):

```
{{"message": { "requestJson": {
        "chainId": {
                "type": "integer",
```

```
            “description”: “the chain ID of the blockchain”
        },
        “sender”: {
            “type”: “address”,
            “description”: “the wallet address making the request, i.e. msg.sender”
        },
        “replyKey”: {
            “type”: “string”,
            “description”: “AES-256 key to encrypt reply, in Base64 format (will be 22 chars)”
        },
        “IDList”: {
            “type”: “array”,
            “description”: “the eNFT IDs to be retrieved, or an empty array to request all”
            [
                “{id}”: {
                    “type”: “string”,
                    “description”: “unique ID assigned to the eNFT”
                },
                ...
            ]
        },
}}},
        “signature”: {
            “type”: “string”,
            “description”: “the privkey EIP-712 signature of the {sender} on the above fields, 130 hex char digest”
        }
}
```

The dApp could populate the IDList[] by searching past event logs for *EnshroudProtocol.TransferSingle* events with topic *_to={address}*, and then subtracting all those IDs also referenced in TransferSingle’s with *_from={address}*. Passing “*IDList: []*” (as the current dApp always does) will cause the receiving MVO to do exactly this in order to return all currently circulating eNFTs belonging to {address}. Note that in order to request keys from an Auditor node, the MVO must attach an EIP-712 signed request from the address that owns the eNFT IDs requested (otherwise it will be denied access).

The MVO’s reply JSON payload will be structured like this:

```
{
        “status”: {
            “type”: “string”,
            “description”: “either the word ‘success’ or the text of an error message”
        },
        “eNFTs”: {
            “type”: “array”,
            “description”: “the eNFT ID and its contents in each case”
            [
```

```
                        {“eNFT001”: {
                                “type”: “object”,
                                “description”: “encrypted contents plus decryption key”,

                                “enshrouded”: {
                                        “type”: “object”,
                                        “description”: “decrypted data, per {enshrouded} schema above”
                                },
                                “key”: {
                                        “type”: “string”,
                                        “description”: “the AES-256 key which decrypts the record on the event log, in base64url format”
                                }
                        }},
                        {“eNFT002”: { … }},
                        …
                ]
        },
        “signature”: {
                “type”: “object”,
                “description”: “signature of replying MVO on the above fields”,

                “signer”: {
                        “type”: “string”,
                        “description”: “the ID of the signing MVO”
                },
                “sig”: {
                        “type”: “string”,
                        “description”: “the actual private key signature, 130 hex char digest format”
                }
        }
}
```

The dApp should verify the MVO’s signature on the reply. It should then parse every returned eNFT and check the signature of whichever MVO signed the eNFT, using the signing key posted for that MVO Id in the MVOStaking contract’s mapping. Additionally, the dApp could subsequently verify each piece of data returned, by fetching the matching record from the blockchain event log, (basic) Base64-decoding it, then decrypting it with the indicated key (using AES-256/GCM with no padding, a 128 bit authentication tag, and utilizing the first 12 bytes of the ciphertext as the initialization vector), and comparing the results to the data which was just returned. The dApp may then request permission from the user to store the validated data in a local file, possibly encrypted with a passphrase. (Note the currently implemented dApp does not perform these additional steps, instead facilitating a save of the eNFT data and keys returned by an MVO.)

At that point it will always be possible to populate *spend* or *withdraw* requests with all decrypted eNFT metadata, as required by the schemas. Such a local backup is similar to a *wallet.dat* file in clients like Bitcoin Core or Litecoin Core, except that having access to the wallet passphrase does not provide the ability to spend any of the eNFTs: doing that also requires access to the privkey for the owning

account. For this reason, a downloaded *enshroud.json* file (or a subset thereof) can also be made public as proof-of-ownership of a stipulated amount of value held in eNFTs.

5. Receipts and Transaction History

The reader may have noticed that all of the *TransferSingle* events generated for the blockchain's event log are either mints or burns. That is, all of them either have *from=0x0* and *to={address}* (a mint), or *from={address}* and *to=0x0* (a burn). At no time is an existing eNFT transferred from one wallet address to another. All inputs are always destroyed and new outputs generated in all transactions.

It's also worth observing that the MEV (miner extractable value) from an Enshroud transaction will always be zero.

It is of course possible to construct a rough transaction history by viewing all the relevant *TransferSingle* events and noting the *operator* addresses shown as the counterparties. But this of course only addresses the metadata, not the substance of any payments. It's also the case that placing transaction history on the blockchain is problematic for privacy in the first place, simply because blockchains, like diamonds, are forever. Ideally transaction history should be capable of being purged when no longer required or wanted. Plus, detailed transaction history should be accessible only to the actual parties to a transaction (payer and payee) – this is an absolute requirement.

In order to accomplish all of this, the lead MVO performs one other service. Right at the point where it is ready to return the {OperationsBlock} to the client (with all committee signatures attached), it will also pause to generate receipts, one for every wallet address involved in the transaction. These receipts conform to the following schema:

Transaction Receipts – Schema Definition

```
{"ReceiptBlock": {
        "type": "object",
        "description": "user transaction receipt, for payers and payees, signed by an MVO",

        "receiptId": {
                "type": "string",
                "description": "random sequence number assigned by the lead MVO, used to obtain the ephemeral privkey needed to decrypt the {receipt} section"
        },
        "receipt": {
                "type" : "object",
                "description": "the complete JSON object as signed by the lead MVO",

                // For a receiptType=sender, this is the user's own address.
                // For a receiptType=recipient, this is the payer's address.
                "source": {
                        "type": "address",
                        "description": "the wallet that provided the funds for the transaction"
                },
```

```
“receiptType”: {
        “type”: “string”,
        “description”: “one of these two values: sender | recipient”
},
“chainId”: {
        “type”: “integer”,
        “description”: “blockchain on which the following {block} number occurs”
},
// N.B.: a block number implicitly designates a timestamp
“block”: {
        “type”: “integer”,
        “description”: “the block number in which the transaction appeared”
},
“tagID”: {
        “type”: “string”,
        “description”: “the unique eNFT ID, whose appearance in a mint/burn operation within a block provides the indication that the transaction has been mined in that block (this field is not emitted within a signed receipt)”
},
“destinations”: {
        “type”: “array”,
        “description”: “the addresses that received funds in the transaction”,

        [
        // For a receiptType=sender, this array will include all payees.
        // However for privacy, individual IDs are not shown, only totals per {asset}.
        // For a receiptType=recipient, this array will include only the user’s address.
        // However, the actual unique IDs are shown for each eNFT received.

                {“payee001”: {
                        “address”: {
                                “type”: “address”,
                                “description”: “the recipient’s address”
                        },
                        “asset”: {
                                “type”: “string”,
                                “description”: “the crypto token type”
                        },
                        // for sender type receipts, this will be a total paid to the address
                        “amount”: {
                                “type”: “integer”,
                                “description”: “the amount of {asset} paid (in decimal)”
                        },
                        // N.B.: not shown for sender type receipts
                        “id”: {
                                “type”: “string”,
                                “description”: “the unique ID assigned to the eNFT”
                        },
                        // optional field
```

```
                                        “memo”: {
                                                “type”: “string”,
                                                “description”: “the memo line (if any) supplied by the
{source} to this recipient (limited to 1024 chars)”
                                        }
                                }},
                                {“payee002”: { … }},
                                …
                        ]
                },
                “signature”: {
                        “type”: “object”,
                        “description”: “signature of MVO on {receipt} data”,

                        “signer”: {
                                “type”: “string”,
                                “description”: “MVO Id of the lead MVO signing the receipt”
                        },
                        “sig”: {
                                “type”: “string”,
                                “description”: “actual private key signature, a 130 hex char digest”
                        }
                }
        }
}}
```

The following rules govern the generation and interpretation of these receipts:

1. For a receipt generated for the sender (payer), the {source} will be their own address. Unique payees are generated, one per separate recipient {address} per {asset}. The {amount} field will be a total paid for that asset type (consolidating multiple eNFTs together if necessary), and the individual IDs of the output eNFTs are not included. This is done to make it harder to trace recipient eNFTs (via transaction receipt log[] entries) in the event that a payer’s receipt was ever disclosed to a third party.

2. For receipts generated for recipients, each recipient gets their own, on which any other simultaneous payees are (naturally) not shown. However every individual eNFT they received will be enumerated, including the unique ID values. This makes it possible for the user’s wallet software to determine which eNFTs in their account were received from the {source} in a particular transaction.

3. On a burn/withdrawal operation, the {source} and sole {destinations} array element are necessarily the same address. In this case, because the output is a token spend on-chain (i.e. to the {source}), the change eNFT will have a special {memo} field indicating it was “auto-generated as change.” (The same memo line is also used in other contexts.)

4. Once the MVO has constructed all of the receipts for a transaction (for both payer and payee(s)), it will generate a random ID string for their {receiptId} field and place them in a temporary queue. It is important that the MVO not upload the receipts until the relevant

transaction has actually taken place by being mined. In order to confirm this, the MVO will watch the event log for a mint or burn operation involving the {tagID} in each receipt. For sender receipts, the tagID should be one of the output eNFT IDs which the transaction is supposed to mint. (Ideally, that of the sender's own change eNFT.) An input ID should not be used, because another transaction could also use it prior to finalization. For a recipient receipt, the tagID will similarly be set to the output eNFT minted to them.

5. Once the {tagID} value condition has been met, the MVO performs the following steps:
   - Note the block number in which the tag appeared, and record it in {block}.
   - Delete the {tagID} field from the structure (as the users do not need to see it).
   - Sign the entire finalized {receipt} block.
   - Query any Auditor for an AES-256 key with which to encrypt the {receipt} section, indexed by the hash "keccak256(chainId+address+{receiptId})". The Auditors will store this hash:key pair in a persistent map (separate from the map for eNFT keys).
   - Encrypt the {receipt} element with the returned symmetric key and a random initialization vector, prepending the IV as the first 12 bytes of the ciphertext. This is then encoded with (basic) Base64.
   - Trigger uploading the receipt to the storage area (see point 6).
   - Once the upload is confirmed as successful, remove the receipt from the queue.

   After X blocks have elapsed without seeing the tagID appear (the value of X dependent on the blockchain, equal to 100x the number of blocks required for confirmation dwell time), purge a receipt from the queue without uploading. This is because at that point the "receipt" represents a proposed transaction that was never finalized, and likely will never be.

6. A base URI is defined in MVO configuration data for the receipt storage location for each supported blockchain. This may be physically resident on any persistent web2 or web3 service. A folder may be initialized for each user's {address}, named for the hash of the {address}. That is, receipts could be stored in a folder located at, for example:

   https://<receiptStorage>/chainID/sha256sum(address)/

   The individual receipt filenames can assume any values, but *{receiptId}.json* is suggested. These files are encrypted to the allocated AES key. The dApp or other user wallet software should decrypt the {ReceiptBlock}, validate the MVO's signature on {receipt}, and then present all of this data in human-readable form for each receipt found. Anything which does not decrypt, is not a receipt, or does not pass signature check, must be ignored or error-flagged. (Note that the *hash(address)* uses a standard SHA2-256 algorithm, because it isn't meant to be accessed except via an MVO, and never appears on-chain.)

   In this way it becomes impossible to determine the {address} belonging to a given storage folder, since hashes are not reversible. Likewise one could not obtain a list of Enshroud user addresses by scanning the base URI for the folder names. However it is possible to compute *sha256sum(address)* and discover whether a given {address} has any receipts posted. They are however encrypted, and thus can only be read by someone who has access to the private key for the address (and hence can request the list of receipts from an MVO). There is also an access capability requirement; but at present this is simply set to SHA2-256({address}).

All MVOs will initially support the storage of receipts in a SQL database (shared among MVOs), in a table indexed by {receiptId} and {chainId}, in which the {owner} is SHA2-256 hashed and the {receipt} text is AES-256 encrypted as described above. While this mechanism is centralized, it has the advantage that user-demanded receipt deletions are atomic. Users are meant to download any receipts they care about, and then encouraged to delete them from the MVOs' database. Doing so also causes the Auditor nodes to remove the corresponding ephemeral AES key for the receipt from the keystore. After that has occurred, no one can decrypt the receipt.

In future, distributed storage for receipts may be implemented (for example on IPFS, Aleph.im, Filecoin, TheGraph, or Storj). Again, note that the storage utilized is configurable per blockchain.

7. Users have the ability to download all or selected decrypted receipts onto their local device, in JSON formats (though CSV would perhaps be easier to import into spreadsheets). This can be useful to establish proof of payment to a certain {address} on a certain date, for example.

   They likewise have the ability to request the permanent deletion of any or all selected receipts, by sending a signed deletion request to any MVO. For security reasons, only MVOs will have access to the storage area. Once a receipt has been purged, it's gone for good and cannot be restored. Note that since receipts are never stored on-chain, deletion costs no gas.

Sidebar 5:

This capability for permanent deletion of some or all of a wallet's transaction history is significantly more protective of user privacy than the mechanisms contemplated by other privacy services that use Zero-Knowledge methods, such as Aztec or Railgun. On those services the transaction history is permanently embedded on-chain in Merkle trees, and can be accessed by the wallet owner at any time, for "compliance purposes." This means any user could be compelled to produce their own transaction history, contrary to the fundamental human right to avoid self-incrimination. A "compliant" privacy system is an oxymoron. Notice too that when receipts are deleted, the ephemeral keys are also purged, making it physically impossible for anyone to decrypt them even if they had a saved copy. (Unlike with downloaded wallet eNFTs, downloaded receipts do not include their ephemeral AES keys. Therefore an MVO must always decrypt a retrieved receipt, and cannot do so after the owner has deleted it, triggering the purge of the corresponding key.)

Receipt-related dApp Messages to MVOs

In order to ensure the deletability of receipts, even read access must require credentials. Otherwise copies could be made by third parties, and retained even after deletion. The storage folder for any given wallet will therefore be accessed with capability bits. The folder index will be *hash(address)*, and the value for the access capability will be *sig(address)*, that is, the address signed by the private key of the address. All client access requests will require *(address, sig(address))* as credentials, to access folder *hash(address)*. The MVOs will be the only parties who can store receipts, to prevent the insertion of bogus receipts, spam or other irrelevant data into the folders. The schema for dApp JSON requests to MVOs related to receipts is as follows. In a http POST payload parameter called *receiptRequest*, with header param encrchain={chainId} (and assuming EIP-712 sigs):

```
{{“message”: { “requestJson”: {
        “chainId”: {
                “type”: “integer”,
                “description”: “the chain ID of the blockchain (identifies receipt storage URI)”
        },
        “sender”: {
                “type”: “address”,
                “description”: “the wallet address making the request, i.e. msg.sender”
        },
        “capability”: {
                “type”: “string”,
                “description”: “the separate signature of the {sender} on the {sender} address”
        },
        “opcode”: {
                “type”: “string”,
                “description”:”nature of operation, one of: list | get | delete”
        },
        “replyKey”: {
                “type”: “string”,
                “description”: “AES-256 key to encrypt reply, in Base64 format (will be 22 chars)”
        },
        // for get and delete opcodes, specific file specifications must be provided as follows:
        “filespecs”: {
                “type”: “array”,
                “description”: “the receipt files to be retrieved (get) or removed (delete)”
                [
                        “filename”: {
                                “type”: “string”,
                                “description”: “relative filename, format {block}-N.json”
                        },
                        ...
                ]
        },
}}},
        “signature”: {
                “type”: “string”,
                “description”: “the privkey EIP-712 signature of the {sender} on the above fields, 130 hex char digest”
        }
}
```

The MVO’s reply JSON payload will be structured like this:

```
{
        “opcode”: {
                “type”: “string”,
                “description”: “one of: list | get | delete (echo)”
        },
```

```
	“status”: {
		“type”: “string”,
		“description”: “either the word ‘success’ or the text of an error message, such as file not found”
	},
	// for a list reply, all filenames present will be returned; for a delete, only those removed
	“filespecs”: {
		“type”: “array”,
		“description”: “the filenames present (list) or confirmed deleted (delete)”
		[
			“filename”: {
				“type”: “string”,
				“description”: “relative filename, format {block}-N.json”
			},
			… more filenames ...
		]
	},
	// for a get opcode only, the actual receipt file contents are also returned:
	“receipts”: {
		“type”: “array”,
		“description”: “the filename and its contents in each case”
		[
			{“receipt001”: {
				“type”: “object”,
				“description”: “relative filename + file contents”,

				“filename”: {
					“type”: “string”,
					“description”: “relative filename, format {receiptId}.json”
				},
				“receiptData”: {
					“type”: “string”,
					“description”: “decrypted data, per {ReceiptBlock} schema above”
				}
			}},
			{“receipt002”: { … }},
			…
		]
	},
	“signature”: {
		“type”: “object”,
		“description”: “signature of replying MVO on the above fields”,

		“signer”: {
			“type”: “string”,
			“description”: “the ID of the signing MVO”
		},
		“sig”: {
```

```
                    "type": "string",
                    "description": "the actual private key signature, 130 hex char digest format"
                }
        }
}
```

The MVO handling the request will silently decrypt each receipt using the appropriate key, obtained from the Auditors' persistent map. To offload processing of receipt operations away from actual value transactions, MVOs listen for these requests on a separate port and URL.

V. Metadata Validator Oracle (MVO) Layer and Environment

The purpose of the MVO layer is twofold:

1. To provide security against fraud, hacking, and exploitation attempts.

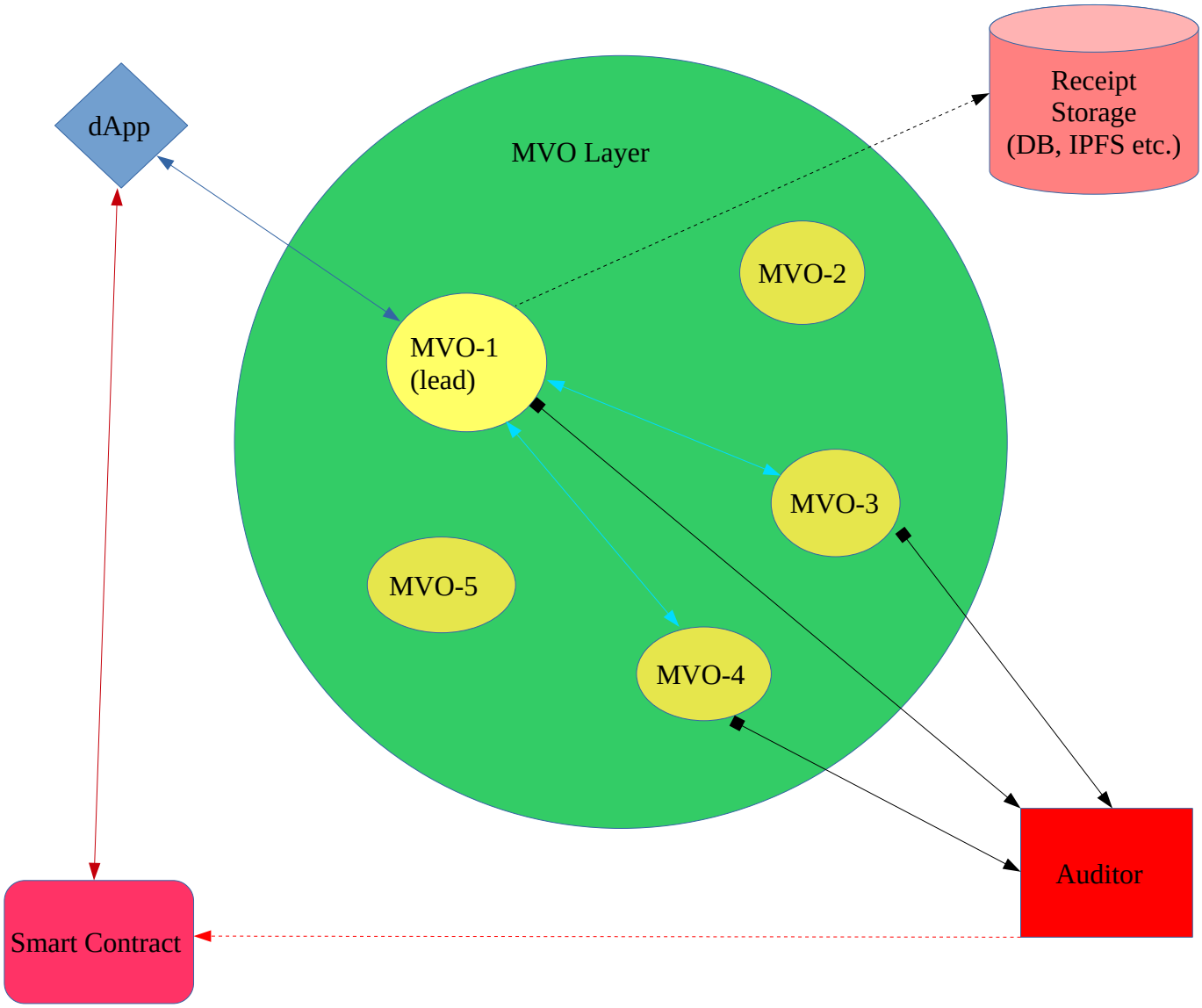
2. To offload significant processing from the smart contract to save gas.

As described above, in order to secure privacy on a blockchain without zk-Snarks or native homomorphic encryption, complete transaction details must be obscured on-chain via hash blinding. The blockchain itself is still utilized to record and log transactions, and prevent double-spending issues. In addition, doing most of the work in parallel off-chain both reduces gas costs and potentially augments transaction throughput, in that capacity scaling can be supplied in the off-chain layer. Random number generation (e.g. for non-sequential eNFT ID values) can also be provided in this layer, obviating the need for a blockchain Oracle supplying VRFs (verifiable random functions).

The Enshroud MVO layer is not a blockchain itself, or even a side-chain. There is no chain, database, or transaction event log, decentralized or otherwise. Therefore the layer is not a true "Layer 2" in the sense of Optimism, Polygon, Avalanche, Aztec, etc. Nor is it a bridge between chains, like ThorChain or Polkadot. It is merely a collection of parallel validators which appropriately "pre-process" dApp requests before they are sent to the smart contract, which performs the actual recordation.

In order to prevent collusion between a dApp client and a single MVO, the MVOs operate in parallel, countersigning each other's work, with all their signatures (on *argsHash* values) validated by the SC on-chain. MVOs stake $ENSHROUD tokens as a further bond against malfeasance, and earn newly minted $ENSHROUD for validating user transactions. Note MVO rewards are completed transaction rewards, not block rewards. Because it serves only as an intermediary between dApp clients and the SC, and interacts only with clients, the MVO layer is blockchain-independent and can support transactions made on any number of distinct chains. As a final failsafe against coordinated fraudulent collusion among a group of MVOs, a passive Auditor function is provided; see section VI below.

The software function of the MVO layer can be illustrated pictorially as follows:



Assumptions in diagram:

- N=3 and M=5; that is, 3 of 5 MVOs must sign off on a user transaction.
- The dApp client randomly selects MVO-1 to interact with.
- MVO-1 randomly selects MVO-3 and MVO-4 to be its (N-1) cosigning committee.
- The arrows represent messages sent between entities.
- Read-only (view) accesses of the SC's state data made by all parties are not shown.

MVO Layer Procedure

The operations performed, in order, are as follows:

1. The dApp client sends a JSON message over HTTP to MVO-1, according to one of the three message schemas stipulated above (for Deposit, Spend, or Withdraw operations).
2. This JSON message is encrypted using ECIES encryption to the MVO's public ECDSA key as specified in the SC for that MVO. Therefore, HTTP is sufficient and if HTTPS were used this

would represent a second layer of encryption. (Note that MVO replies are {OperationsBlock} objects which partly go into mempool calldata in any case and as such are not private anyway.)

3. MVO-1 decrypts the message using its corresponding private ECDSA key to solve for the AES-256 key used, validates the request, constructs the matching eNFT Operations Block JSON object, according to the schemas described above, and signs it.
4. MVO-1 forwards the dApp's request along with its signed Operations Block to its selected committee members, MVO-3 and MVO-4 (asynchronously). These messages are encrypted and signed among MVOs using RSA-4096 keypairs, over ws:// (websocket) connections.
5. MVO-3 and MVO-4 independently check MVO-1's work, and append their own signatures to the Operations Block, and return the block + ECDSA signature to MVO-1.
6. MVO-1 collates the replies from MVO-3 and MVO-4, and when all are received and validated, returns a single response to the dApp, containing the signatures of all 3 MVOs on the data.
7. Immediately after signing the OB, each MVO also creates an Auditor Block (the OB but with the hashed data shown in the clear), signs it, and broadcasts it to the Auditor.
8. If the Auditor notices any discrepancies in an Auditor Block, it will greylist the output eNFT IDs involved by notifying the SC, as soon as the suspect ID gets minted. (See section VI.)
9. The dApp verifies the OB as signed by all 3 MVOs, extracts the necessary data from the OB to build its parameters passed to the appropriate SC method in calldata, and invokes that method, passing the argsHash value and the MVO commitee signatures. The SC validates the MVO signatures attesting to the dApp's passed parameters, and performs the indicated mint/burn operations.

The transaction capacity of the MVO layer can be scaled by adding MVOs (+*M*). The security level can be increased by increasing the number of signatures required (+*N*). The number of internal communications among MVOs is a function of the ratio of *N* to *M*. The obvious security concern is collusion between >=*N* corrupt MVOs, which could alter output eNFTs amounts to facilitate an allied client dApp in withdrawing a value which exceeds the aggregate value of the input eNFTs. Because of the Auditor function, this will always be detected and prevented. The values set for *N of M* are recorded in the SC's configuration and are set by the constructor (i.e. by the SC's owner at deployment). There are a number of trade offs involved in setting these values, such as:

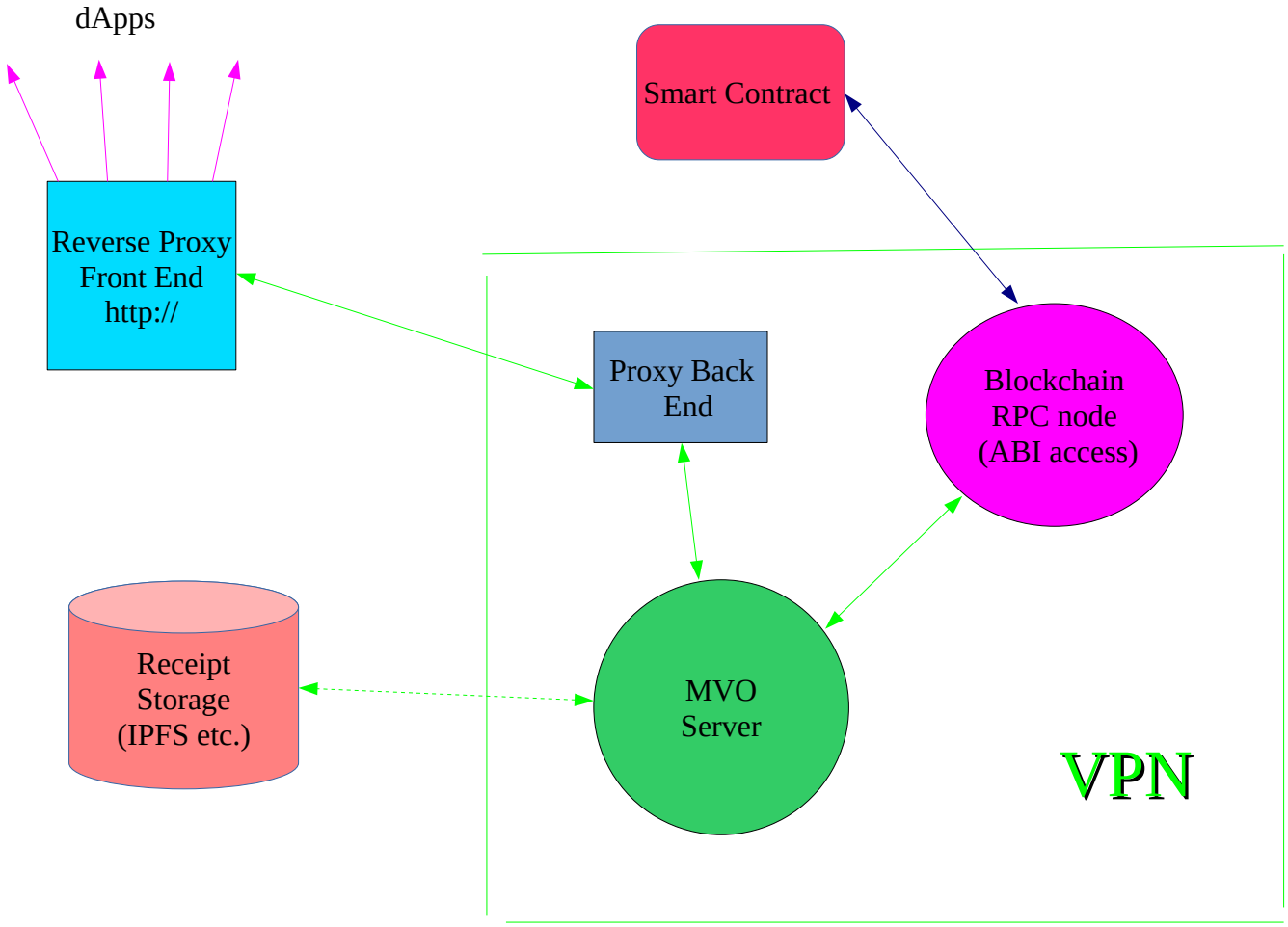
- Increasing *N* will significantly increase gas costs on the chain due to more *ecrecover()* calls.
- Increasing *N* potentially increases latency, i.e. worsens user response time.
- Increasing *N* increases traffic between MVOs (and the Auditors), as well as load.
- Increasing *M* without increasing *N* adds processing capacity.
- At most *M*-*N* MVOs can be offline at any time without blocking transactions.
- Too high *M* for user demand will result in high idle time for MVOs, and could reduce earnings.
- A given MVO is selected (both as the lead and as a sub) at random but weighted by its staking amount. This will select for MVOs with more "skin in the game," but could result in an oligarchy if the value of *M* is set too low (or too high).
- Given sufficient capacity, the same set of MVOs could support traffic on >1 blockchain.

An estimated minimum deployment basis is: *3 of 20* MVO signatures required.

Let us now examine the network operating environment for each individual MVO.

MVO Layer Network Structure

Each “MVO” entity shown above is actually a small cluster of virtual machines performing different tasks. The entire MVO layer, plus the Auditors, is deployed on a WireGuard VPN (virtual private network). This allows intra-layer communications (between MVOs) to happen over a third layer of encryption, and without being monitored on the public internet. The architecture required is as follows:



Observations:

The clients’ communications with the MVO layer take place solely through an HTTP proxy. Each MVO must define one or more Front End Proxies. These listen on a specific URL (which dApps can obtain by querying the SC), such as: *http://IPv4:10443/* What is meant by “reverse proxy” is that the Front End has no idea how to contact the Back End. Instead, the Proxy Back End machine connects to each Front End it services, via a VPN exit IP. In this way the Back End system does not need a public IP address at all, and the Front End, were it compromised, cannot be used to discover the location of the Back End server. The Back End does nothing more than maintain a persistent channel (itself encrypted using autossh) to each Front End it is supporting, and forwards all traffic received, over the VPN to the MVO Server. Also note that the Back End sees all traffic as originating from the Front End’s IP, while the MVO Server sees all traffic as originating from the Back End (which is presumably a private network IP). *This prevents MVOs from ever seeing a client’s real IP address.*

The MVO Server system is likewise without a public IP address requirement. It runs the MVO node software, and communicates only on the VPN to its Proxy Back End, peer MVOs, its Blockchain RPC, and Auditors. When it needs to generate JSON receipt files, it will upload to storage via the VPN exit(s). (To storage servers, all MVOs will therefore look like the same client.) The VPN may support its own internal DNS servers to prevent DNS leakage, but the MVO Server should never need to look up any external domains, other than to generate external receipt files or to install software updates.

Note further that due to the use of ECIES encryption (see Sidebar 2 above) between the dApp and the lead MVO, it is not required that an MVO obtain a domain or site certificates. That is, a plain http://IP4 or http://IP6 URL will be sufficient. This avoids a potential identity compromise for the MVO's operator, in that domain names cannot be registered anonymously. The dApp client's connection is secured end-to-end with the MVO Server using ECIES, thus making all user data traffic across the Front/Back proxies opaque both to the proxies and to VPN routers.

The Blockchain JSON-RPC Node is the only machine which would benefit from dual ethernet controllers. One is configured to talk only on the VPN; the other has public internet access, to remove the latency across the VPN for quick access to the blockchain. (Note that it should not have any domains associated with its public IP, nor listen on any other public ports, and it should appear indistinguishable from any other random full node on that blockchain.) This node exists only to provide fast read-only access to the SC and other blockchain data. For example, when the MVO wishes to confirm the validity of eNFTs, it will look it up in a local cache constructed by making a filter request of the RPC Node at startup, along with ongoing subscriptions to *TransferSingle* and *URI* events. Being a full node in possession of all blocks and event history, it should be able to provide an immediate and authoritative response. This prevents the MVO from needing the services of a third party node (such as a connection through Infura). However, no submissions to the mempool (requiring gas) will ever be made by the MVO's Blockchain RPC Node, as all transactions are actually submitted and paid for by dApp clients.

For reduced latency, it may be advantageous to have all three of these VPN components co-located in a single data center, so they can communicate at LAN speed. In theory all three of these machines could be run as vserver instances on a single physical server, but they are logically separate and the MVO Server in particular may benefit from clustering, because its implementation is multi-threaded. For security, the Front End Proxy should never be co-located alongside the VPN nodes; rather, an entirely separate data center and hosting company should be used for the Front End Proxy. As an additional optimization, encrypted SSD filesystems can be used on any or all of the servers. Note however that the only permanent records stored on these three hosts are public chain data kept on the Blockchain RPC Node, which by definition is not secret.

The goal is to maximize throughput and uptime without compromising the security of the platform. The described configuration largely limits the impact of DDOS attacks to Front End Proxies, and removes all associations between the actual processing machines and internet domain names. Care should be used in registering for all hosting arrangements (such as using domain privacy, shell company registrants, pay fees with crypto, etc.).

The MVOStaking SC natively supports MVO's creating and joining/leaving MVO Pools. Such a Pool can be operated as a profit center by any party willing to provide MVO network infrastructure, by means of dues defined for the Pool and automatically paid by MVO operators when claiming rewards.

VI. The Auditor Function

Auditors do not directly participate in processing transactions for users. However, they monitor everything done by all MVOs, and compare it against SC state data. This monitoring is passive, with no actions taken unless a problem is discovered. An Auditor server has its own keypair used for communications, and open websockets to all active MVOs. Messages broadcast to an Auditor by an MVO resemble the eNFT Operations Block generated for the SC, except that actual amounts, assets, and rand salts of eNFTs are always shown in clear. MVO broadcasts to Auditors are encrypted to each Auditor's public key (RSA-4096). Because MVOs need to know the status of Auditor servers, these connections are not RESTful. A continuous websocket connection with keep-alive pings is maintained from each Auditor to every MVO, so that connection drops will be noticed. If no Auditors are available, MVOs are required to immediately reject all transaction requests until such time as at least one Auditor server is online. To prevent a SPF (Single Point [of] Failure), the system must have multiple Auditor servers, so that individual Auditors can go offline without disabling user transactions.

The software and hardware of Auditor servers are provisioned and controlled by the DAO only. (That is, there are no third parties who can operate an Auditor.) All Auditors are deployed within the VPN cloud, and communicate only with MVOs and their own captive Blockchain RPC Node. Since they do not interact with users, they do not require Front or Back End Proxies. The Auditors maintain a shared list of eNFT IDs they have seen, for each blockchain, along with a pass/fail status (boolean value). Like MVOs, they do not have public IP addresses or listen ports. The logic for Auditors is as follows:

Auditor Algorithm

- Decrypt each broadcast from an MVO using its own private RSA key, and confirm MVO's sig.
- Validate the eNFT Operations Block, including the MVO's signature on the block.
- Validate the dApp's user signature on the original request to the lead MVO (forwarded).
- Obtain all eNFT metadata from the event log via the adjacent Blockchain RPC Node. This of course includes ensuring that the AES key for each eNFT exists in the keyring mapping.
- Confirm that all input eNFT IDs are on its known list, marked as passing. Any marked as having previously failed are rejected. Any failed input invalidates all outputs.
- Confirm that the details hashes of the input eNFTs recorded in the SC's mapping match the hashes calculated from the values passed in the signed {AuditorBlock}. If any do not match, all output IDs become invalid.
- Confirm that stored encrypted metadata for each output eNFT matches the value passed in the {AuditorBlock} JSON. Note the IDs of any invalid outputs.
- Calculate the details keccak256(abi.encode({owner},{id},{asset},{amount},{rand})) for every output eNFT, using the values passed in the {AuditorBlock}. Store this in a shared map, indexed by the eNFT ID, with status="valid".
- If any output IDs fail tests, log the failure including the blockchain ID, token ID, MVO ID, and the details of the issue found. The token ID is added to the permanent list of known failures.
- For all failures which actually get minted by the SC, invoke the auditorGreyList() method on the SC, adding the suspect ID(s) to a mapping. This is done via the Blockchain RPC Node, and requires the Auditors (unlike MVOs) to have gas available to modify state. The SC keeps a private list of keys belonging to Auditors which are specifically permitted to manipulate the greylist mapping. (Auditors can add to the greylist, but only admins can effect removals.)
- For each sub-MVO that sends a duplicate {AuditorBlock} with its own signature, this same procedure is followed again, requiring the stored details hashes to match for all outputs.

- The Auditor then listens for mint events emitted from the SC (T*ransferSingle*). When one is received, the ID looked up in the list of eNFTs with "valid" status. If the detailsHash is not found or does not match, the ID is immediately marked with "suspect" status. An attempt to greylist with the SC is queued, and on completion the ID is marked with "blocked" status.
- If mint events occur which the Auditor was not expecting (because no {AuditorBlock} messages were broadcast to it by any MVOs), it will queue an auditorGreyList() invocation on the {outputs}. This situation may be indicative of a collusion attack by *N* or more MVOs.
- The Auditor also listens for burn events from the SC, and removes them from the "valid" list (by marking them "deleted"). Missing or deleted IDs seen as {inputs} in the future are automatically rejected, marking all transaction {outputs} as "suspect" and greylisting them should they ever appear in a mint operation.
- Auditors maintain lists of bad output IDs in "buckets," all of which will be greylisted at once should any ID in the bucket get minted. In the event that these buckets are large enough to exceed the max number of IDs which can be greylisted simultaneously (20), multiple Auditors will begin working from opposite ends of the bucket to get through the list more quickly by working in parallel without overlap.
- Some configurable length of time after an eNFT is burnt, its AES key is deleted from the keystore map. This helps make forensic analysis impossible, except by wallet owners who might retain downloaded local copies of their burned eNFT keys.

In order to prevent a race condition developing between the finalization of a fraudulent user transaction at the SC with the Auditor's effort to greylist its outputs, by rule no eNFT can be used as an input in a transaction until *X* blocks have elapsed since its minting. The value of *X* (known as the "dwell time" of "conf blocks") will vary between blockchains, assuming larger values where blocks occur more rapidly, but can never be set below 1. For example, on Ethereum POS the custom is to wait at least 64 blocks (2 epochs, or about 12.8 minutes) dwell time between transactions. Enshroud currently uses 80. Should colluding MVOs maliciously fail to communicate with the Auditor altogether, this will be detected via monitoring of the blockchain's event log. MVOs must broadcast to all available Auditors, and buffer up any transmissions that could not be sent due to Auditor downtime. On coming online, Auditors will reconnect to all available MVOs and request a sync-up. An Auditor must complete its resync before greylisting any eNFT IDs for being unknown. Auditors should also sync their token ID results lists (additively) against one another, and against each blockchain's event log (for mints and burns), as a double-check for completeness. The former sync will be accomplished via a shared database mapping among the Auditors. The latter is achieved via a dynamic cache rebuild at startup.

Auditor Key-server Functions

As described above, the actual eNFT {enshrouded} metadata stored on the event log is always encrypted with ephemeral AES-256 keys. That is, a unique key is generated for each eNFT as it gets minted, and that key will continue to exist until after the eNFT is eventually burned. (The key is thus "ephemeral" in the sense that, unlike an account keypair, it doesn't need to outlive the eNFT it's related to.) To avoid giving individual MVOs control over the generation or storage of the keys (which could be abused), Auditors will generate keys upon requests from lead MVOs during transaction processing. They will also store the keys for later retrieval. Storage will be in a mapping:

keccak256(chainId+ID+address) => base64url(AES-256 secure-random key)

If an index hash value does not exist in the map, one will be created, using a secure random number generator (i.e. one seeded with entropy). If a key does exist for the index, the current value will be

returned. The keys will be stored using Base64.Url encoding (i.e. without padding using Table 2 of the spec, which does not employ the '/' character).

In order to keep all Auditors in sync using consistent data, the Map will be stored in a database cluster to which every Auditor node belongs. Each Auditor will access the DB and synchronize to it at startup. The effect of this is that an MVO can ask a given Auditor for a new key, and then another MVO can ask a different Auditor for the same hash index and receive the same key value, instants later. (A situation which actually arises frequently given the way MVO committee verification works.)

A separate Map will also exist for receipts:

keccak256(chainId+receiptID+address) => base64url(AES-256 secure-random key)

The Auditors will purge entries from the eNFT key Map time *T* after burn events being recorded on the event log. Entries will be purged from the receipt key Map following explicit delete requests made by clients through MVOs. (The signed deletion request from the dApp will be forwarded.) This key deletion feature works to frustrate forensic analysis involving no-longer relevant eNFT or receipt objects, by removing all copies of the only key in the universe which could have decrypted them. It is possible that old copies of receipts may be stored locally by dApp users, and similarly that eNFT keys may be preserved in *enshroud.json|dat* files downloaded and retained by users.

Because of the way the distributed Map works within the cluster, MVOs may make all key-related requests of any random Auditor node, and safely expect identical results. The RPC-JSON key request schema between a MVO and an Auditor is as follows:

```
{
	"mapping": {
		"type": "string",
		"description": "which mapping keys are wanted for, one of: eNFT | receipt"
	},
	"opcode": {
		"type": "string",
		"description": "one of: new | get | delete (delete applies only to {mapping}=receipt)"
	},
	"hashes": {
		"type": "array",
		"description": "the hash indices for which keys are wanted"
		[
			"hash": {
				"type": "string",
				"description": "keccak256(hash data)"
			},
			… more hashes ...
		]
	},
	// forward of original user's receipt-related request
	"receiptRequest": {
		"type": "object",
```

```
            "description": "user's signed request to the MVO, as per receiptRequest as shown in Section IV above, which must match and authorize what the MVO is requesting"
        },
        "signature": {
            "type": "object",
            "description": "signature of sending MVO on the above fields",

            "signer": {
                "type": "string",
                "description": "the ID of the signing MVO"
            },
            "sig": {
                "type": "string",
                "description": "the actual private key signature, 130 hex char digest format"
            }
        }
}
```

The Auditors will log the requested key operations made by each MVO. This does not compromise privacy, because the index hashes are not reversible. But it does make it possible to identify forensically which MVOs have accessed a particular eNFT or receipt object, for example in the event that private information about a certain item was ever improperly revealed.

The Auditors' reply block will conform to this schema:

```
{
        "status": {
            "type": "string",
            "description": "either the single word 'success', or an error message"
        },
        "keys": {
            "type": "array",
            "description": "the list of AES keys returned"
            [
                {"mapEntry001": {
                    "type": "object",
                    "description the individual index/key map entry"

                    "hash": {
                        "type": "string",
                        "description": "echo of the hash index sent by MVO"
                    },
                    "key": {
                        "type": "string",
                        "description": "the base64Url-encoded AES-256 key"
                    }
                }},
                {"mapEntry002": { … }},
                …
```

```
            ]
        },
        “signature”: {
            “type”: “object”,
            “description”: “signature of replying Auditor on the above fields”,

            “signer”: {
                “type”: “string”,
                “description”: “the ID of the signing Auditor”
            },
            “sig”: {
                “type”: “string”,
                “description”: “the actual RSA-4096 private key signature, base64 format”
            }
        }
}
```

N.B.: Auditors sign replies to MVOs using their private RSA keys of 4096 bits. Requests sent to them by MVOs are always encrypted using the corresponding public RSA key. This additional layer of encryption also mitigates against a MITM attack carried out against a WebSocket connection, because the impostor would also need the MVO or Auditor’s private RSA key. An Auditor server uses its ECDSA account private key only to sign greylisting requests sent to the SC on each supported blockchain.

VII. Phase Two A: Smart Contract Support API

Once the system architecture described above is launched, the obvious extension to it (beyond deployment on additional sans-privacy blockchains), is to enable smart contracts from other projects to receive, read, and manipulate eNFTs. In this way DeFi services such as DEXs, lending platforms, and liquidity farming could be conducted under a shroud of privacy as well, without the need to migrate the protocols and assets to a Layer 2 providing such privacy. Instead, projects desiring to “go dark” will be able to integrate the Enshroud Smart Contract API.

Because eNFT metadata is stored only in the event log, and SC code cannot directly access event log data, nor interface with the MVO layer, the proposed API will define an Oracle service.

To achieve this, a new server type will be added inside the VPN, the API Gateway. This Gateway will manage the access of subscribing smart contracts, and monitor a queue initialized within its own SC for each client contract. A subscriber SC will use its API keys to submit requests onto this queue. The Gateway will continuously monitor for state changes made to its queues, and respond appropriately.

To preserve privacy, client requests will always refer to eNFTs by ID only. For example, a client contract may submit a request for the sum of the values in asset type {asset} of an array of eNFT IDs, through making a call such as:

EnshroudGateway.queueRequestForTotal(APIKey, “WETH contract address”, “[ID1, ID2, ID3 ...]”, <signature>);

The Gateway's SC function will check the sender address, the validity of the APIKey, and the signature, and if all is okay add the request to the subscriber's queue and return a *tId* (transaction ID) for the request. The Gateway's monitor server will notice the appearance of the request on the queue when the transaction is mined.

In order to process this request, the Gateway must be able to decrypt the {enshrouded} metadata for each referenced ID of an eNFT belonging to the client contract. To do this it will make requests of an Auditor for eNFT keys, using RPC-JSON very similar to that shown above in section VI for MVO-Auditor key requests. As with other key requests, ownership must be demonstrated to elicit data.

To accomplish this, in setting up its subscription the client SC must register proxy account credentials (both the public *and* the private key) for use by the Gateway to manipulate the client contract's eNFTs. Essentially, the Gateway will act as a proxy for making requests of the MVO layer, just as the dApp does for live human users, with the difference that the dApp utilizes the actual keypair of the user's wallet (after the user has connected their wallet to the dApp and approved it for reading the account address's contents). Using the established shared keypair, the Gateway proxy will drive the Enshroud protocol much as the dApp does, except that communications between the "user" (in this case a SC) must be conducted via a queue in the API's smart contract rather than over an HTTP connection.

So in the case of our request for a total value above, the Gateway will fetch all of the eNFT metadata for the referenced IDs, decrypt each, find the sum, and call the API reply function, which must be incorporated into the client SC's own code. For example:

clientContract.EnshroudAPI.onTotal(tId, 3141592600000000000, <sig of Gateway on tId>);

It should be noted that there is no need for notifications of eNFTs paid to the client SC to pass through this interface, as those are already handled by the client contract's (required) implementation of *ERC1155TokenReceiver.onERC1155Received()*, which is invoked whenever eNFTs are minted to the contract's address. However, logically the client's code implementing that function should queue up appropriate API Gateway requests to ascertain exactly what was just received.

Notice that while MVOs will accept requests signed by the proxy credentials when they originate from a Gateway, the Gateway cannot actually spend or burn any eNFTs, because they belong to the client SC's address, not to the proxy account credentials. All it can do is fulfill the role of the dApp layer in communicating with the MVO layer to pass data as instructed via the queue, to trigger the generation of {OperationsBlock} objects, as described above for the three fundamental operations of *deposit, spend, and withdraw*. Once passed back up to the client SC via the reply functions (e.g. *EnshroudAPI.onSpendOperationsBlockReceived()*), the client must then invoke the appropriate function in the Enshroud SC itself, signing with the operating key of its address which contains the eNFTs. This should be done in the context of a subsequent end user client dApp action, so that gas is paid by the end user of the client SC. The implications for privacy are exactly the same as for end user {OperationsBlock} objects generated via the Enshroud dApp.

The detailed functional definition of all methods required in the interfaces *EnshroudGateway* (for queuing requests to the Gateway) and *EnshroudAPI* (for handling the replies) is beyond the scope of this document. This functionality belongs to a second phase which will be designed after launch.

As with the Auditors, the Gateway nodes shall be operated and maintained by the DAO, in sufficient numbers that at least one will always be online. Like the Auditor nodes, they will each consist of a dedicated server inside the VPN with continuous connections to all MVOs, together with a private full blockchain node. Because gas is required in order to use the on-chain queue mechanism for communications, subscribing contracts will need to fund their API accounts with $ENSHROUD. (Similar to Chainlink price Oracles which must be seeded with a quantity of $LINK.) The net $ENSHROUD earned from API Gateway subscriptions (according to a fee schedule established by the DAO) will be paid out to $ENSHROUD DAOPool stakers, after deducting equivalent gas costs incurred by the interface.

Lastly, note that dwell times must still be observed. For example, a client contract may not, upon notification of receipt of an eNFT paid to it by one of its users (such as in a deposit context), immediately utilize the API Gateway to spend that eNFT prior to the dwell time in conf blocks having elapsed. Replies to queued requests will be processed by the Gateway as quickly as possible, but to prevent abuse no reply shall be sent earlier than in the next block after the one in which the request was mined. Dapp code of client contracts should monitor the event log for API replies on its end of the queue, and offer their users further actions only after those replies are seen.

Phase Two B: MVO Load Balancing

As usage grows, individual MVO servers may become saturated, especially if their share of transactions (based on their relative staking quantum) is large. To prevent any bottlenecks from developing, a new Request Router Node can be deployed. The dApps would utilize the Router to send all requests, and the Router would forward them on to lead MVOs, according to a modified selection algorithm. This new algorithm would select the lead MVO at random skewed by relative MVO staking size, as before, but capped by load average. When a given MVO system reached the cap (say ~80% of capacity), any selection of a load-capped MVO would be replaced by a random re-selection made excluding all capped nodes. The same algorithm could also be used by the lead MVO in selecting its committee (i.e. allow the Router to pick them). This implicitly requires that the Router be continuously aware of each MVO's load average.

To prevent the Request Router from itself becoming a single point of failure (SPF), in the event that it was not reachable or timed out, the dApp would fall back to the original algorithm, itself selecting a lead MVO at random skewed by staking size.

VIII. Conclusion

Privacy is a right, not a privilege extended only to those who can prove they aren't up to something that might displease those who claim to be their masters. Historically, physical cash and coin have provided privacy by default. It's extremely unfortunate that in the digital age most cryptocurrencies, the closest modern analog to cash, inherently provide no meaningful privacy, merely a weak form of pseudonymity. *The Enshroud protocol fixes this.* If Bitcoin, and crypto in general, fixes the issuance of money by privatizing and decentralizing it, private circulation fixes crypto's single largest defect.

The threat of "govcoin" in the form of CBDCs issued by central banks (which will have zero privacy by design) creates an imperative for private cryptos and their users to adopt privacy-enhancing measures preemptively. Enshroud enables privacy as a service, building a privacy layer on top of

existing blockchains without any need to alter their operation. In this way users who value their right to privacy will have the option to exercise it through the use of enshrouded eNFTs.

Private transactions are only the beginning, of course. Once smart contracts are written that use the Enshroud API to hold and manipulate eNFTs, a private DeFi layer becomes possible on existing public chains. Liquidity pools, lending, AMMs, staking with eNFT receipt tokens, and much more can be reimagined to incorporate native privacy. The Enshroud mechanism is extensible without requiring special pro-privacy chains. It may also assist with scaling issues, by offloading certain processing to a validator network without the sync overhead of a sidechain or shards linked by a beacon chain. Since withdrawal/burn transactions provide no privacy, ideally they will one day be performed mainly in rollups by smart contracts or exchanges, rather than by individual users.

In sum, Enshroud provides a platform which can help the crypto ecosystem evolve to survive and thrive in a landscape of increasing government market friction, and (soon) state-sponsored competition. Owning $ENSHROUD is like taking out a hedge against looming future regulation, which potentially increases its yields with every new encroachment committed by governments held captive by established financial and political interests. Join us as we help to build out the post-political financial system!

Thank you for your interest and kind attention.