timelock.zone

A public time-locked cryptographic service
enabling anyone to encrypt data for decryption in the future,
with support for the most common cryptographic schemes

Aragon ZK Research

June 2023 - ***Draft***

Abstract

In this document we present our *timelock.zone* service, which will enable anyone to encrypt data for decryption in the future, or to commit data for opening in the future. It has the following key characteristics:

1. Public keys for periods far into the future are always available;
2. Support for many cryptographic schemes;
3. Relies on trusted randomness (drand beacon) published by the League of Entropy;
4. Possibility of public participation;
5. The correctness and security of the scheme is guaranteed as long as a single party participating in the public key computation is honest;
6. These parties do not need to be present when the private key is revealed.

We present two protocols that can be used for a timelock service. The first one 4.1 is quite simple, but the parties participating in the key-generation process are required to send a ZK proof of the correctness of its parameter. The second protocol 4.2 uses a key concept from the first one but is designed in a way that does not require a ZK proof. Our timelock service will use this second protocol.

1 Introduction

Time-locked cryptograpy (TLE) refers to cryptographic systems which guarantee that ciphertexts will be decipherable only at a certain time in the future. Such systems are also variously referred to as time-lapse, time-based, time-dependent, delayed unlocking etc. cryptographic protocols. TLE cipher-texts remain locked for a specified period of time, after which even an encoder cannot prevent decryption.

Time-lapse cryptography has many applications. One of them is in e-voting systems: ballots are encrypted during the voting period, and can be publicly decrypted after the voting period is over. This allows for transparent and independent verification of results. Indeed, the *timelock.zone* is the result of our work on a voting system for which we needed a time-lock service.

In this paper, we present both the outline of a future public serice, called *timelock.zone* as well as a generic protocol that can be used to to implement it. In our protocol, we specify some encryption system, such as an encryption scheme, commitment, signature, or any public key protocol. The master public key of this particular protocol is generated in a distributed way among a set of parties. The security of the scheme is guaranteed as long as there exist at least a single honest party that participates in the key generation protocol.

2 Description of the Timelock service

2.1 Overview

2.1.1 User interface

There will be a user interface at `https://timelock.zone` to query any keys published by the Timelock service.

2.1.2 Timelock periods

For every suported scheme, we plan to make available public keys for times in the future as follows: every hour for the next 2 years, and every day for the next 10 years [1].

2.1.3 Supported schemes

We plan to support a wide variety of schemes to cover use cases not only in web3, but also in more traditional settings. Among the curves supported are Curve 25519, P-256, P-384, P-521, P-256, P-384, P-512, secp256k1, BabyJubJub, MNT6[2] etc.

2.1.4 The timelock.zone blockchain

The timelock.zone blockchain will be used to store the store the shares of the various public keys, and also to ensure that consensus is reached as to which shares are used to compute each public key. It is being built using Gears, a Rust implementation of the Cosmos SDK.

2.2 Lifecycle of a public/private keypair

This outlines the different phases of the process for a specific keypair.

Contribution phase To generate a private key, we need several parties to generate "shares" of the private key. These "contributors" make their shares publicly available on the timelock blockchain. At least one of these parties needs to be honest, meaning it discards the private data generated during the computation of the share. Note that the software for generating shares will be open source, and it can also be used to verify if shares are valid.

Private key generation The validators are the parties who will compute the public key. This group can be different from the group of contributors, but in practice we expect that all validators will also be contributors. Once the contribution phase is over, the role of the validators is decide which shares to include in the public key generation process. Obviously, only valid shares should be included. Once the set of shares is decided, the public key can be generated from these shares and published on the blockchain.

Waiting period During this phase, the private key corresponding to the previously published public key is not yet available. The shares corresponding to the public key are visible on the blockchain and can be used to verify the validity of the public key.

Publication of the private key Once the corresponding drand signature (cf. infra) has been published, the private key can be computed by anyone as long as all shares are available on the blockchain. The original contributors do not need to be involved. The validators jointly publish the private key on the PBB. Once this is done, the shares are no longer needed, although they will remain available on archival nodes.

[1] These parameters are still subject to change

[2] We consider $MNT6-753$ curve generated in [2]

2.3 Key generation

A contribution or share is represented by the tuple $(R, \mathsf{scheme}, \mathsf{pp})$, where:

- R is a round, which corresponds to a UNIX epoch
- scheme is the identifier of the scheme
- pp dataset that can be used to compute a public key, by aggregating it with other datasets

The contribution does not include information about who submitted the contribution, but that information will be available on the blockchain. A contribution is "valid" if:

- it has been submitted by a validating node, or by a whitelisted participant (conditions for whitelisting will be determined in the future)
- the scheme is supported
- it has been submitted during the contribution period for R
- verification of data is successful

A timelock dataset is represented by (R, scheme, Vec(data), PK, SK):

- Vec(data) is a vector of participant contributions submitted during the submission period
- PK is the public key, computed using Vec(data) once the contribution period ends
- SK is the secret key, computed using Vec(data) and $sig_L(R)$ once the latter become available

A timelock item gets initialised when the first valid contribution for (R, scheme) is received.

2.4 Contribution periods

As stated above, we aim to publish public keys as follows: every hour for 2 years, and every day for 10 years, with a contribution period of 14 days. For a given date D and full hour there is a well-defined contribution period:

- For D:12 (noon) the contribution period will be (D - 10y - 14d):12 to (D - 10y):12
- For D:hh (hh not 12) the contribution period will be (D - 2y - 14d):hh to (D - 2y):hh

This means that

- at D:12 (noon of day D):
  - start contribution period for (D + 10y + 14d):12
  - end contribution period for (D + 10y):12
- at D:hh (top of the hour hh of day D, with hh not 12):
  - start contribution period for (D + 2y + 14d):hh
  - end contribution period for (D + 2y):hh

Obviously we will need to compute numerous public keys when launching the service.

Contributions will be accepted iff the block timestamp of the last block is within the contribution period. Public keys for a round R (and a given scheme) are computed as soon as there is a block whose timestamp is after the end of the contribution period. Finally, the private keys are computed as soon as the drand signature for round R is published.

2.5 Expected data volumes

We estimate that a single share will be roughly 50 kB, or 0.05 MB [3]. If we want to generate keys on an hourly basis for one year in advance, with 10 contributors and for 10 schemes, this will represent approximately 365*24*10*10*0.05 kB = 43,800 MB = 43.8 GB of data. This number will grow in proportion to the number of supported schemes, the number of contributors and the number of years for which keys will be generated in advance. To have keys for 20 schemes 2 years in advance with 20 contributors, we will need to store 350 GB of data. To this, we need to add shares of daily keys for periods exceeding 2 years. Clearly, such data volumes cannot be stored on a public blockchain.

3 Random Beacon

We assume that there exists a trusted party that generates and publishes a random beacon at regular intervals. In our case, this trusted party will be the League of Entropy (LoE) which will be described below 5.

3.1 Beacon Parameters

The private parameter of the Beacon is its private key $\mathbf{sk}_L \in \mathbb{Z}_p$. Its public parameters are:

$$\mathsf{pp}_L = \langle p, \mathbb{G}_1, g_1, \mathbb{G}_2, g_2, \mathbb{G}_T, \mathbf{e}, H, \mathsf{PK}_L \rangle$$

- $\mathbb{G}_1$, $\mathbb{G}_2$ and G_T are groups of order p.
- $\mathbb{G}_1$ and $\mathbb{G}_2$ are generated by g_1 and g_2 respectively.
- $\mathbf{e} : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is a non-degenerate pairing function.
- $H : \mathbb{Z}_p \to \mathbb{G}_1$ is a "number to point" hash function.
- $\mathsf{PK}_L = g_2^{\mathbf{sk}_L}$ is the public key.

Note that in case of a symmetric pairing we will have $\mathbb{G}_1 = \mathbb{G}_2$ and $g_1 = g_2$.

3.2 Beacon Output

At regular time intervals, the Beacon publishes the signature $H(R)^{\mathbf{sk}_L}$. R is a round number which corresponds to a UNIX epoch at which the value is published. The publication time is known in advance.

4 Protocols for a Public Timelock Service

Our goal is to define a protocol in which a set of participants cooperate in computing a public key of some encryption scheme Π^{enc}, so that the corresponding private key can be obtained at some future time, but not before. The participants have access to the data from the Random Beacon.

The scheme Π^{enc} is defined over a group $\mathbb{G} = \langle g \rangle$ of order q, and we assume that there exists a hash function $\mathscr{H} : \mathbb{G}_T \to \mathbb{Z}_q$ that acts as a random oracle.

[3] The exact size will depend on the scheme and on the compression algorithm used, if any.

4.1 Protocol 1

The first protocol is quite simple but requires a ZK proof. It has three non-overlapping phases for each round R. During the *contribution phase*, participants submit their public parameters. Once that first phase is over, the master public key $\mathsf{MPK}(R)$ is computed and the *waiting phase* starts. The third phase, the *decryption phase*, begins after the Beacon has published the signature $H(R)^{\mathbf{sk}_L}$. The secret key $\mathbf{sk}(R)$ is computed, and any data encrypted using $\mathsf{MPK}(R)$ can be decrypted.

4.1.1 Participant contribution

Every participant follows the protocol as shown in table 1. This algorithm is run by each $\mathsf{party}^{(i)}$ but for simplicity the index is omitted. Note that in line 6, $\mathbf{sk}$ must be represented as a bitstring using a suitable serialisation method.

The contribution of each participant are the public parameters $\mathsf{pp} = (\mathsf{PK}, T, y, \pi)$. After computing these public parameters, the private values $\mathbf{sk}, \mathbf{t}$ and $\mathbf{Z}$ must be deleted from memory.

In the following, we will denote the public parameters of participant i by $\mathsf{pp}^{(i)} = (\mathsf{PK}^{(i)}, T^{(i)}, y^{(i)}, \pi^{(i)})$ and the private parameters by $\mathbf{sk}^{(i)}, \mathbf{t}^{(i)}, \mathbf{Z}^{(i)}$.

	public	private
1		$\mathbf{sk} \xleftarrow{\$} \mathbb{Z}_q$
2	$\mathsf{PK} \leftarrow g^{\mathbf{sk}}$	
3		$\mathbf{t} \xleftarrow{\$} \mathbb{Z}_p$
4	$T \leftarrow g_2^t$	
5		$\mathbf{Z} \leftarrow \mathbf{e}(H(R), \mathsf{PK}_L)^{\mathbf{t}}$
6	$y \leftarrow \mathscr{H}(Z) \oplus \mathbf{sk}$	
7	proof π	

Table 1: Protocol 1

The proof π is a ZK proof for the following inputs and assertions:

- generic public inputs: $\mathsf{PK}_L, H(R)$
- participant-specific public inputs: PK, T, y.
- private inputs: $\mathbf{sk}, t$
- assert $PK = g^{\mathbf{sk}}$
- assert $T = g_2^t$
- assert $y = \mathscr{H}(\mathbf{e}(H(R), \mathsf{PK}_L)^{\mathbf{t}}) \oplus \mathbf{sk}$

4.1.2 Public key computation

Once all participants have published their contributions, the master public key will simply be the product of the individual public keys:

$$\mathsf{MPK}(R) = \prod_i \mathsf{PK}^{(i)} \tag{1}$$

This master public key can then be used for encryption, commitments and signatures with the chosen cryptographic scheme.

4.1.3 Private key computation

At round R, the Beacon publishes the value $H(R)^{\mathbf{sk}_L}$. For every participant i we can compute:

$$\begin{aligned}\mathbf{e}(H(R)^{\mathbf{sk}_L}, T^{(i)}) &= \mathbf{e}(H(R)^{\mathbf{sk}_L}, g_2^{\mathbf{t}^{(i)}}) \\ &= \mathbf{e}(H(R), g_2^{\mathbf{sk}_L})^{\mathbf{t}^{(i)}} \\ &= \mathbf{e}(H(R), \mathsf{PK}_L)^{\mathbf{t}^{(i)}} \\ &= \mathbf{Z}^{(i)}\end{aligned}$$

We then obtain $sk^{(i)}$ via:

$$\mathbf{sk}^{(i)} = \mathscr{H}(\mathbf{Z}^{(i)}) \oplus y^{(i)}$$

Once this is done for all participants, the private key for round R is:

$$\mathbf{sk}(R) = \sum_i \mathbf{sk}^{(i)}$$

4.2 Protocol 2

The second protocol is seemingly more complex, but avoids general-purpose NIZK proofs. It is this protocol that will be used by the timelock service. The integer k is the security parameter, and we define $\mathcal{H}_k$ as the first k bits of a cryptographic hash function $\mathcal{H}$. The phases of this protocol are the same as in the previous one.

4.2.1 Participant contribution

The non-interactive algorithm for participating in the key generation for round R is shown in table 2. As before, for simplicity the index of the party is omitted in that table. We define:

$$\overrightarrow{\mathsf{PK}_0} = (\mathsf{PK}(1,0), \mathsf{PK}(2,0), ..., \mathsf{PK}(k,0))$$

The vectors $\overrightarrow{\mathsf{PK}_1}$, $\overrightarrow{T_0}$, etc. are defined in a similar way. The output of party i is then:

$$\mathsf{pp}^{(i)} = \left(\mathsf{PK}^{(i)}, \overrightarrow{\mathsf{PK}_0}^{(i)}, \overrightarrow{\mathsf{PK}_1}^{(i)}, \overrightarrow{T_0}^{(i)}, \overrightarrow{T_1}^{(i)}, \overrightarrow{y_0}^{(i)}, \overrightarrow{y_1}^{(i)}, \overrightarrow{t^*}^{(i)}\right)$$

4.2.2 Interactive protocol

In the interactive version, the participant first executes steps 1 through 14 and sends

$$(\mathsf{PK}, \overrightarrow{\mathsf{PK}}_0, \overrightarrow{\mathsf{PK}}_1, \overrightarrow{T}_0, \overrightarrow{T}_1, \vec{y}_0, \vec{y}_1)$$

to the verifier. The verifier then sends a random bitstring $\vec{b}$ of length k to the participant, who returns $\overrightarrow{t^*}$.

	public	private
1		$\mathbf{sk} \xleftarrow{\$} \mathbb{Z}_q$
2	$\mathsf{PK} \leftarrow g^{\mathbf{sk}}$	
		$\forall j \in [k]$
3		$\mathbf{sk}(j,0) \xleftarrow{\$} \mathbb{Z}_q$
4		$\mathbf{sk}(j,1) \leftarrow \mathbf{sk} - \mathbf{sk}(j,0)$
	$\forall j \in [k]$	
5	$PK(j,0) \leftarrow g^{\mathbf{sk}(j,0)}$	
6	$PK(j,1) \leftarrow g^{\mathbf{sk}(j,0)}$	
		$\forall j \in [k]$
7		$t(j,0) \xleftarrow{\$} \mathbb{Z}_p$
8		$t(j,1) \xleftarrow{\$} \mathbb{Z}_p$
	$\forall j \in [k]$	
9	$T(j,0) \leftarrow g_2^{\mathbf{t}(j,0)}$	
10	$T(j,1) \leftarrow g_2^{\mathbf{t}(j,1)}$	
		$\forall j \in [k]$
11		$\mathbf{Z}(j,0) \leftarrow \mathbf{e}(H(R), \mathsf{PK}_L)^{\mathbf{t}(j,0)}$
12		$\mathbf{Z}(j,1) \leftarrow \mathbf{e}(H(R), \mathsf{PK}_L)^{\mathbf{t}(j,1)}$
	$\forall j \in [k]$	
13	$y(j,0) \leftarrow \mathscr{H}(\mathbf{Z}(j,0) \oplus sk(j,0)$	
14	$y(j,1) \leftarrow \mathscr{H}(\mathbf{Z}(j,1) \oplus sk(j,1)$	
15	$\vec{b} \leftarrow \mathcal{H}_k(\mathsf{PK}, \overrightarrow{\mathsf{PK}}_0, \overrightarrow{\mathsf{PK}}_1, \overrightarrow{T}_0, \overrightarrow{T}_1, \vec{y}_0, \vec{y}_1)$	
	$\forall j \in [k]$	
16	$t^*(j) \leftarrow \mathbf{t}(j, b_j)$	

Table 2: Protocol 2

4.2.3 Validity check

The algorithm for checking the validity of data submitted by a participant is shown in table x.

Figure 1: Algorithm to verify the validity of public parameters sent by Participants

Require: pp is well-formed
1: **for** $j = 1, \ldots, k$ **do**
2: **assert** $PK(j,0) \cdot PK(j,1) == \mathsf{PK}$
3: **assert** $T(j,b_j) == g_2^{t^*(j)}$
4: $\hat{Z}(j) \leftarrow \mathbf{e}(H(R), PK_L)^{t^*(j)}$
5: $\hat{s}(j) \leftarrow \mathscr{H}(\hat{Z}(j)) \oplus y(j,b_j)$
6: **assert** $g^{\hat{s}(j)} == \mathsf{PK}(j,b_j)$
7: **end for**
8: **Output** accept

We start by checking that pp is well-formed, i.e. it has the required number of group/field elements.

In line 2, we check that all the public keys are compatible. This also means that the participant must know the values $\mathbf{sk}(j,0)$ and $\mathbf{sk}(j,1)$, except with negligible probability.

Now remember that if the participant generated pp correctly, then $t^*(j) = \mathbf{t}(j,b_j)$. If that is the case we will have $g_2^{t^*(j)} = g_2^{\mathbf{t}(j,b_j)} = T(j,b_j)$ and the assert in line 3 will pass. We will also have $\hat{Z}(j) = \mathbf{Z}(j,b_j)$ and $\hat{s}(j) = \mathbf{sk}(j,b_j)$, and as a result the assert on line 6 will pass.

4.2.4 Public key computation

Once all participants have published their contributions, the master public key is computed in the same way as in protocol 1:

$$\mathsf{MPK}(R) = \prod_i \mathsf{PK}^{(i)} \tag{2}$$

4.2.5 Private key computation

At round R, the Beacon publishes the value $H(R)^{\mathbf{sk}_L}$. For every participant we can now compute:

$$\begin{aligned}\mathbf{e}(H(R)^{\mathbf{sk}_L}, T(j,0)) &= \mathbf{e}(H(R)^{\mathbf{sk}_L}, g_2^{\mathbf{t}(j,0)}) \\ &= \mathbf{e}(H(R), g_2^{\mathbf{sk}_L})^{\mathbf{t}(j,0)} \\ &= \mathbf{e}(H(R), \mathsf{SK}_L)^{\mathbf{t}(j,0)} \\ &= \mathbf{Z}(j,0)\end{aligned}$$

We then obtain $\mathbf{sk}(j,0)$ via:

$$\mathbf{sk}(j,0) = \mathscr{H}(\mathbf{Z}(j,0)) \oplus y(j,0)$$

The values of $\mathbf{Z}(j,1)$ and $\mathbf{sk}(j,1)$ are computed similarly, and we therefore obtain $\mathbf{sk}(j) = \mathbf{sk}(j,0) + \mathbf{sk}(j,1)$ which should be equal to $\mathbf{sk}$ for all $j \in [k]$ if the participant was honest.

Once this is done for all participants, the private key for round R is then again:

$$\mathbf{sk}(R) = \sum_i \mathbf{sk}^{(i)}$$

4.2.6 Soundness error

The soundness error will be 2^{-k} so there is a trade-off: we have freedom to choose k as large as possible to reduce the soundness error and as small as possible to make the system more efficient. Any polynomial dishonest prover can cheat with prob 2^{-k}. This is true only for the interactive version.

When done non-interactively, the soundness is no longer statistical because the prover can make brute force attacks. If TLCS keys are supposed to be used for short periods we consider that a security parameter between 80 and 100 should be sufficient for most applications. Observe that the error also scales down with the number of parties participating in the protocol. So concretely, we suggest a security parameter of $k = 80$ for a small number of participants (e.g. less than 10) and keys with periods of validity of a few months and $k = 100$ for hundreds of participants and keys with a validity of several years.

4.3 On extending the protocol to generic one-way functions

In the two protocols presented above, we have assumed the existence of a group $\mathbb{G} = \langle g \rangle$ of order q as the basis of our cryptographic scheme. We can generalise these protocols to generic one-way functions (OWF) as long as these have a suitable homomorphic property. All groups have this homomorphic property, as $g^x g^y = g^{x+y}$, but it also holds for RSA and for certain post-quantum OWFs. This means that we can apply our protocols to these schemes.

	public	private
1		$\mathbf{sk} \xleftarrow{\$} A$
2	$\mathsf{PK} \leftarrow f(\mathbf{sk})$	
3		$\mathbf{t} \xleftarrow{\$} \mathbb{Z}_p$
4	$T \leftarrow g_2^t$	
5		$Z \leftarrow \mathbf{e}(H(R), \mathsf{PK}_L)^t$
6	$y \leftarrow \mathscr{H}(Z) \oplus \mathbf{sk}$	
7	proof π	

Table 3: Protocol 1 for OWF

Let $f : (A, *) \to (B, \circ)$ be such a one-way function, with the property $f(x) \circ f(y) = f(x * y)$. We can then modify Protocol 1 as shown in table 3. The only requirement is that elements of A can be represented as bitstrings of a suitable length. If n denotes the number of participants, the master public key is then obtained as:

$$\mathsf{MPK}(R) = \mathsf{PK}^{(1)} \circ ... \circ \mathsf{PK}^{(n)} = f(sk^{(1)} * ... * sk^{(n)})$$

Once $H(R)^{\mathbf{sk}_L}$ is published, we can recover $sk^{(i)}$ for all participants as before, because it is represented as a bitstring. The private key for round R is then

$$\mathbf{sk}(R) = \mathbf{sk}^{(1)} * ... * \mathbf{sk}^{(n)}$$

Protocol 2 can also be adapted in a similar way.

Additional Note. In order to make the protocol quantum secure, we could replace the cryptosystem with homomorphic post-quantum ones such as schemes proposed in [3] and [1]. In this stage, it is yet to be ascertained whether these schemes are compatible with the TLS protocol, warranting further investigation in future research.

4.4 Security Analysis

A timelock protocol should satisfy two main security properties. The first one, which we call *soundness*, should guarantee that if a contribution is accepted and published on the blockchain during an execution of the protocol for a round R then the master public key $\mathsf{MPK}(R)$ obtained by aggregating all contributions is such that at round R, $\mathsf{MPK}(R)$ can be inverted to obtain $\mathbf{sk}(R)$ using as auxiliary information the secret key SK_L published by the beacon. A protocol that is not sound would be one in which a malicious adversary is able to inject a contribution that makes $\mathsf{MPK}(R)$ not invertible at round R. Our main protocol satisfies soundness since the interactive version of our protocol is indeed a sigma protocol and as such is knowledge sound.

The other main security property that a TLCS protocol should satisfy is what we call *privacy*. We have seen that the algorithm requires the participants to delete the private variables $\mathbf{sk}$, $\mathbf{t}$ and $\mathbf{Z}$ from memory at the end of the computation. If only a single participants acts honestly and deletes these private variables, then the secret key $\mathbf{sk}$ corresponding to $\mathsf{MPK}(R)$ is protected until round R, assuming that the majority of LoE members are honest. This means that the only added assumption beyond LoE is that there is a single honest party. Observe that, if participation in the protocol were to be opened to the public, then if someone does not trust the system they can decide to participate in the protocol themselves.

5 League of Entropy

The League of Entropy (LoE) is a collaborative project to provide a verifiable, decentralized randomness beacon accessible via `https://drand.love/`. Its founding members are Cloudflare, École Polytechnique Fédérale de Lausanne, Kudelski Security, Protocol Labs and the University of Chile. As of June 2023, the number of participants has risen to 18.

LoE uses the BLS12-381 curve for signing. It started generating its "pedersen-bls-chained" beacon in July 2020. This beacon produces a random value every 30 seconds and is not suitable for time-based encryption. In March 2023, LoE launched its "bls-unchained-on-g1" beacon, which is generated every 3 seconds and is suitable for time-based encryption.

Useful links:

- HTTP API reference
- Chain hashes of running networks
- Chained beacon (sine July 2020): info round 1 latest round
- Unchained beacon (since March 2023): info round 1 latest round

5.1 The LoE unchained beacon

The "bls-unchained-on-g1" beacon generated its first round at UNIX epoch $1,677,685,200$, which corresponds to 1st March 2023 at 15:40:00 UT. A signature is published every 3 seconds. Here is some example Rust code to convert between epochs and rounds.

```
const LOE_GENESIS: u64 = 1677685200; // 01MAR2023 16:40 UT

fn get_round(epoch: u64) -> u64 {
    assert!(epoch >= LOE_GENESIS); // before genesis
    assert!(epoch % 3 == 0); // time not divisible by 3
    (epoch - LOE_GENESIS)/3 + 1
}

fn get_epoch(round: u64) -> u64 {
    assert!(round > 0); // there is no round 0, the first round is 1
    LOE_GENESIS + (round - 1)*3
}
```

The public key is:

```
a0b862a7527fee3a731bcb59280ab6abd62d5c0b6ea03dc4ddf6612fdfc9d01f01c31542541771903475eb1ec6615f8d
0df0b8b6dce385811d6dcf8cbefb8759e5e616a3dfd054c928940766d9a5b9db91e3b697e5d70a975181e007f87fca5e
```

The following is the query to obtain the signature of round 1:

`https://api.drand.sh/dbd506d6ef76e5f386f41c651dcb808c5bcbd75471cc4eafa3f4df7ad4e4c493/public/1`

This returns a JSON with the following content, in which the "signature" corresponds to $H(1)^{\mathbf{sk}_L}$, while the "randomnes" is just the SHA256 hash of the signature:

```
{"round":1,
"randomness":"ef076e4d0b9320bf3f50cb2940777ae6bbee79c3d620d8efc04195bfc0568486",
"signature":"9544
    ddce2fdbe8688d6f5b4f98eed5d63eee3902e7e162050ac0f45905a55657714880adabe3c3096b92767d886567d0"}
```

Note that a query for round 0 will return the signature for the latest round.

References

[1] Khin Mi Mi Aung, Hyung Tae Lee, Benjamin Hong Meng Tan, and Huaxiong Wang. Fully homomorphic encryption over the integers for non-binary plaintexts without the sparse subset sum problem. *Theor. Comput. Sci.*, 771:49–70, 2019.

[2] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Scalable zero knowledge via cycles of elliptic curves. *Algorithmica*, 79(4):1102–1160, 2017.

[3] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*, volume 8042 of *Lecture Notes in Computer Science*, pages 75–92. Springer, 2013.