Ruby: a private data management layer for Web 3.0 in a multichain world

Ruby Labs
info@ruby.io

Version 1.0.1

Contents

1 Introduction

1.1 What Problem Ruby Network Solves

The existing internet economic model relies almost exclusively on the monetization of personal data. The recent scandals involving internet companies mishandling individual data, such as Facebook's Cambridge Analytica scandal has prompted many individuals to awaken to the fact that in the current internet economy they are the product. People are tired of the fact that all the value produced from their data is accrued into the hand of a few monopoly companies that care little about their privacy. On the other hand, Web 3.0, as an emerging new model for the internet economy is calling for redistribution and democratization of value flowing in the internet.

Web 3.0 revolution has brought us many interesting concepts and products. Decentralized finance (DeFi) allows any user to have an access to sophisticated financial services without the need of opening a bank account. As the size of the cryptocurrency market grows, we expect many traditional institutions to join the crypto market in the near future. A natural requirement of institutional money is regulatory compliance. Due to the sensitive nature of know-your-client (KYC) information, users, especially newcomers to DeFi are usually reluctant to give out their private information to the DeFi apps. On the other hand, the traditional financial (TradFi) system has gathered tremendous KYC information regarding their customers. How to leverage this private information while addressing the users' privacy concerns so that more TradFi users can join the sphere of DeFi has become a central challenge. As a matter of fact, many traditional institutions such as HSBC or Lloyds have already joined the open banking movement, in which consumers are able to grant their KYC data to third-party providers in a fine-grained manner. A private data sharing mechanism between TradFi institutions and DeFi apps that enables the users to have a fine-grained access control on their private data will be in high demand as the regulatory compliance requirement for DeFi grows.

Another interesting concept that represents the spirit of Web 3.0, non-fungible token (NFT), first as a new way of monetizing of digital artwork, has gradually been transformed into a new frontier of redefining decentralized identifiers (DID). NFT can play various roles in a decentralized projects. For instance, numerous decentralized projects are starting to issue NFTs to represent how involved a particular community member is with the project. An artist can issue an NFT as an exclusive membership token for his/her fans. In many cases, the ownership of an NFT also signifies one's identity or status. As a matter of fact, there is a growing trend using NFT as a premise for gated access to either digital or physical experience [nftGatedAccess]. For instance, imagine you wish to attend an NFT-gated online concert on a Metaverse platform. You could only attend the concert if you are either the owner of at least one Crypto Punk or at least two Bored Apes. Here the online concert could be replaced with an art exhibition in the Metaverse or a DAO zoom meeting, and it could also be a physical service such as renting a bike [pavemotors]. The access policy could be any logical formula that connects all kinds of NFT tokens.

Compared with traditional public-key encryption where the message receiver gets either nothing

or all, functional encryption allows the message sender to precisely define what information on the encrypted message is allowed to be revealed to the receiver. In fact, functional encryption was originally proposed as a tool to enforce fine-grained access control over encrypted data [GPSW2006]. Therefore, it is a natural solution to all the aforementioned access control problems. Take the gated NFT case as an example, the access key to the online concert could be encrypted under the policy "1 Crypto Punk & 2 Bored apes". Only those who have a secret key corresponding to at least two Bored apes or 1 Crypto Punk is able to decrypt the secret key entering the concert. Similarly, functional encryption can also serve as a handy tool for the private data-sharing platform between TradFi and DeFi.

With this context, Ruby will design and implement a fine-grained personal data management framework, which would serve as a second-layer/middleware protocol interacting with the multi-chain ecosystem. The framework will enable a data owner or an entity representing the owners to enforce a fine-grained access control policy over the encrypted private data using Functional Encryption (FE). The access control policy will be built into the smart contracts, and the relevant monetary transaction will also be executed via the smart contracts. This is why this solution is defined as a second-layer/middleware protocol.

The major technical contributions of this project will be:

- We propose a novel watermarking technique and Functional Encryption methodology to build a private data management framework by leveraging the power of blockchain technology and smart contracts, at a fine-grained level. The proposed system will enable a fine-grained data management layer for the multi-chain world and Web 3.0, including private data sharing platform between TradFi and DeFi, NFT-gated access, etc.
- We devise an access control mechanism to ensure that legitimate buyers, on receiving data owner's permission, can get access to the target data, be it the private KYC data in the case of private data sharing between TradFi and DeFi, or the access token of NFT-gated access system.
- We introduce a payment system, which will allow data trading between sellers and buyers on our platform. A review mechanism, with incentivization to buyers, is also put in place to attract more sellers and buyers in the system, and to motivate them to behave honestly.
- Finally, we propose a Fine-grained access control policy that will be built into the smart contracts that will be deployed in multi-chain, and the data monetization transaction will also be executed via the smart contracts.

1.2 Why Choose Functional Encryption

There are three relevant projects to Ruby: the first one is perhaps the Enigma project, a privacy protocol that enables the creation of decentralized applications (DApps) that guarantee privacy. The protocol Enigma is based on secure multi-party computation (MPC). The second one, Insights Network, is a data exchange protocol based on combining blockchain technology, Substrate module, and MPC. It is based on the EOS blockchain and a custom MPC system. The third

one, NuCypher, is a cryptographic infrastructure for privacy-preserving applications. Its main technology is threshold proxy re-encryption and fully homomorphic encryption.

There are several different ways of implementing an MPC protocol: threshold homomorphic encryption, garbled circuit, and secret sharing. The general idea of MPC is to outsource private data (either in the form of secret shares or homomorphic encryption) to a few separate computing parties so that they can perform confidential computation over the encrypted data. Directly applying MPC to fine-grained data monetization is problematic in the sense that once the data is outsourced, the data owner does not retain any control over what type of computation can be performed by the computing party. In other words, individual privacy is now at the mercy of these computing parties, which is against the owner-centric ethos of fine-grained data monetization, where the access control policy should be defined by the data owner and enforced by the algorithm.

On the other hand, functional encryption was specifically proposed and tailored for enforcing fine-grained access control over encrypted data. Compared with traditional public key encryption scheme that only allows the message receiver to either decrypt the whole data set or nothing, functional encryption allows the sender to determine exactly which part of the message can be decrypted by exactly which kinds of receivers. By allowing the data owner to define the access control policy, the owner has full control over what type of access the data purchaser can have over the encrypted data. The only decryption result the data purchaser will be able to retrieve is the predefined function evaluation.

1.3 Competitive Edge

Data privacy is always high on the agenda when it comes to the Crypto community. Since 2015, different projects have been working to improve that on three directions:

1. Anonymous currency, represented by Zcash, Monero, etc. They mainly meet the needs of anonymous payment.
2. The privacy projects for crypto payment and transaction, represented by Tornado, zkSync, Starkware, Raze Network, etc. They mainly meet the privacy needs of smart contracts.
3. Data-based privacy computing projects, represented by Ruby Network, Oasis, and PlatON. They mainly meet the privacy computing needs of the data transaction market.

On the track of data privacy computing, Ruby's main competitors are Oasis, PlatON, Phala, Enigma, ARPA, etc. Ruby's main differentiated advantage lies in proposing a privacy computing solution based on user data attributes for Functional Encryption computing. Users can have fine-grained control of their data so that data buyers can obtain the computing results while the user's specific data content remains private.

	Encryption Tech	Value Position	Whether Fine-Grained	Built Data Marketplace	Owner Centric
Oasis	Trusted Execution Environment	Layer 1 public chain for encryption Apps	No	No	No

	Encryption Tech	Value Position	Whether Fine-Grained	Built Data Marketplace	Owner Centric
PlatON	The Mix of Multiparty Computation, Zero-Knowledge Proof, Verifiable Computing, Secret Sharing, Homomorphic Encryption	Layer 1 public chain for encryption Apps	No	No	No
Phala	Trusted Execution Environment	Layer 1 public chain for encryption Apps	No	No	No
Ruby	Functional Encryption	Layer 2 Application, focus on building data marketplace	Yes	Yes	Yes
Enigma	Secret Sharing and Multiparty Computation	Layer 1 public chain for encryption Apps	NO	NO	No
ARPA	Multiparty Computation	Layer 1 public chain for encryption Apps	No	No	No

The aforementioned projects are mostly based on cryptographic solutions. There also exist several projects, such as Mintlayer [Mintlayer] or lit protocol [litprotocol], that try to implement access control based on non-cryptographic solutions such as access control list (ACL). However, there are mainly two problems with ACL-based solution:

- First, it does not provide confidentiality protection on the access control policy. On the other hand, functional encryption not only can enforce confidentiality requirements on the underlying message but also guarantee the secrecy of the access control policy.
- It is also hard to extend the access control policy to the real world beyond blockchain platforms given most of these ACL can only be implemented as smart contracts that run on the smart contract platforms. In contrast, a functional-encryption based solution can be adjusted to the access control policy that is defined on terms outside of the blockchain world. For instance, a user of TradFi can obtain secret keys corresponding to a certain KYC information from a TradFi institution to authenticate his/her identity to a DeFi app. In other words, functional encryption is a perfect tool to build a private data management middle-layer bridging the Web 2.0 world and Web 3.0.

1.4 Stakeholders

1) Validator
   In the Ruby protocol, Validators produce blocks and package transactions. They are responsible for running the nodes of the Ruby chain.
2) Nominator
   Nominators can delegate their RUBY tokens to Validators to gain staking rewards.
3) Data Owner

Data Owner refers to users who provide data at source. They could be either the TradiFi users or a host of NFT gated event. The Data Owner profits from the fine-grained access control of a certain data.

4) Data Buyer

Data Buyer refers to users who wishes to gain access to the data of data owner. It could be the TradiFi users' private KYC data or the access key to a NFT gated event. It publishes data calculation function f(x) out of its needs and purchases the data from Data Owner and Data Collector. The Buyer pays in accordance with the agreement to the Data Owner and Data Collector.

5) Data Collector

Data Collector is the Market Maker of Ruby Data Marketplace. It helps Data Buyer to purchase data. The Collector can also buy from various Data Buyers on the market. Data Collector helps to improve the data transaction efficiency of Ruby Data Marketplace.

6) RubyDAO Governor

RubyDAO Governor is responsible for the governance of RubyDAO. It consists of the core developers of Ruby Protocol, popular Crypto KOLs, well-known big data experts and scholars, and data transaction legal experts. An important task of RubyDAO governor is to admit and register new key authorities.

2 Functional Encryption

2.1 Introduction

Function encryption was first proposed by Boneh, Sahai and Waters in 2011. This encryption algorithm broke the original "All-or-Nothing" data query mode. The computing results of functional encrypted data can be obtained without revealing the plaintext content.

A classic Functional Encryption Scheme contains four algorithms, namely:

- FE.Setup: $(\text{PK}, \text{MSK}) \leftarrow \text{Setup}(1^\lambda)$. The setting algorithm is used to generate the master key (MSK) and public key (PK)
- FE.KeyGen: $\text{SK} \leftarrow \text{KeyGen}(\text{MSK}, k)$. The key generation algorithm is used to generate the key for the function f_k.
- FE.Enc: $C \leftarrow \text{Encrypt}(\text{PK}, x)$. Encryption algorithm is used to encrypt data.
- FE.Dec: $y \leftarrow \text{Decrypt}(\text{SK}, C)$. The decryption algorithm is used to safely calculate the function value $y = f_k(x)$.

The above is a general form of function encryption. In order to realize that users can achieve fine-grained control of their data, that is to say, the data owner can specify who can access the encrypted data and has complete control over the data, Ruby will use the Ciphertext-Policy ABE (CP-ABE) algorithm in the Functional Encryption algorithm to encrypt their personal data.

CP-ABE is a special form of Functional Encryption. It does not need to be similar to other encryption methods, such as asymmetric encryption, where the recipient's identity information is

needed for each encryption and must be encrypted multiple times when sent to multiple users. CP-ABE only needs to set the access policy to perform encryption only once. When the attributes of the user meet the policy described by the encryptor, the data user can decrypt it.

CP-ABE also solves the problem of key leakage caused by symmetric encryption key transmission and protects the privacy of both data owners and users.

2.2 CP-ABE Algorithm

CP-ABE embeds the desired data strategy (f_k) into the ciphertext and embeds the attributes into the user key. The ciphertext maps an access structure, the key of an attribute set. Decryption is valid only if the attributes in the attribute set can satisfy the access structure. CP-ABE attribute-based encryption uses a password mechanism to protect data. The data owner specifies the strategy for accessing the ciphertext and associates the attribute collection with the access resource. The data user can access the ciphertext information according to his/her own authorized attributes to achieve sharing of private data.

Since CP-ABE embeds the strategy (f_k) into the ciphertext, it means that the data owner can determine which attributes can access the ciphertext by setting the strategy. In other words, the data is fine-grained controlled for encrypted access so that it can be encrypted and stored on the public cloud for fine-grained sharing.

Based on the CP-ABE algorithm, Ruby will support the revocation and restoration of user attributes. The execution includes 7 steps:

1. **FE.Setup**: The initialization algorithm is a randomization one, and the initialization only generates the system public key *PK* and the system master secret key *MSK*.
2. **FE.Setup.Cancel**: The input parameter is *PK*, a prime number field *P* is generated, and *list=1* is calculated for each attribute att. The algorithm outputs the initialized *P*, *map<userGID,prime>* and *map<att,list>*. The list is not a common one but a large, prime number, which is used to record whether it has been revoked.
3. **FE.KeyGen**: By the set *S* provided by *PK*, *MK* and data requesters, the trusted authorization center generates the user secret key *UK* associated with the attribute set for the data requesters. Apply a prime number for the user in the prime number field, delete this prime from *P* to ensure that the primes obtained by different users are varied, and then store the prime number in the mapping table *map<userGID,prime>*.
4. **FE.Enc**: The input parameters of the encryption algorithm (randomization algorithm) are *PK*, the message to be encrypted *M*, the access control structure associated with the access policy, and the ciphertext based on attribute encryption is output.
5. **FE.Dec**: Decryption is a deterministic algorithm, executed by the data requester. Decryption is divided into two steps. The first step is to access the leaf node of the policy tree, let $i = att(x)$, where x represents the leaf node of the ciphertext policy access tree. If $i \in S$, the algorithm obtains the revocation list corresponding to this attribute list and modulo *list% prime*. If the value is not 0, it means that the user has not been revoked. If it is 0, it means

that it has been revoked and the decryption is over. Step 2: After the verification of Step 1 is passed, the algorithm enters *UK* and ciphertext *M*. If the attribute set satisfies the access Strategy, the algorithm can successfully decrypt the ciphertext *M*.

6. **User.Attribute.Cancel**: *att* is the algorithm input parameter; *list* is the cancellation list corresponding to the attribute; and prime is the prime number prime corresponding to the user. When att of a user is cancelled, *DO* takes out the prime corresponding to the user and the list corresponding to the attribute from the mapping tables *map<user,prime>* and *map<att,list>* respectively, and calculates *list'=listprime*. The revocation list corresponding to the attribute *att* is updated to *list'*.
7. **User.Attribute.Recovery**: *att* is the algorithm input parameter; *list* is the cancellation list corresponding to the attribute; and *prime* is the prime number corresponding to the user. The cancellation of the user may be temporary. When the user re-owns the attribute *att*, *DO* takes out the *prime* number corresponding to the user and the *list* corresponding to the attribute from the mapping tables *map<user,prime>* and *map<att,list>* respectively, and calculate *list'=listprime*. The revocation list corresponding to the attribute *att* is updated to *list'*.

2.3 Attribute and Function Management

1) User management

Basic information owned by the user (minimum data set): GID (provided by CA or business side, but overall uniqueness must be guaranteed), CP-ABE user attributes (provided by the business side), unique prime number (provided by the CA user registration unit), private key (generated by CA user private key generation unit).

The user submits information (GID, CP-ABE user attribute, unique prime number) to the CA, applies for registration and generates a user private key, the user private key is stored in the CA, and the CA provides an interface to query the user and obtain the user private key.

2) Modification of strategy function

In practical engineering applications, better choose the cancellation scheme with low complexity, low computing cost, and simple implementation. Therefore, the strategy cancellation scheme adopted in this design is as follows: Revoke users through a fixed-length cancellation list record, so that the cancellation process does not need to update the secret keys of the system and related users, which can greatly reduce the computational burden caused by the cancellation.

3) Cancellation

User attribute cancellation: *att* is the algorithm input parameter; *list* is the cancellation list; *prime* is the prime number corresponding to the user prime. When the *att* of a user is cancelled, *DO* takes out the *prime* corresponding to the user and *list* corresponding to the attribute from the mapping tables *map<user, prime>* and *map<att, list>* respectively, and calculates *list'=listprime*. The cancellation list corresponding to the attribute *att* is updated to *list'*.

4) Recovery

User attribute restoration: *att* is the algorithm input parameter; *list* is the cancellation list corresponding to the attribute; *prime* is the prime number corresponding to the user. The cancellation of the user may be temporary. When the user re-owns the *att*, *DO* takes out the *prime* number and the cancellation *list* from the mapping tables *map<user,prime>* and *map<att,list>* respectively, and calculate *list'=listprime*. The cancellation list corresponding to the attribute *att* is updated to *list'*.

2.4 Concrete scheme

Functional encryption was originally proposed as a fine-grained access control mechanism over encrypted data [GPSW06]. A data owner encrypts a message x to generate ciphertext, which could be stored in an untrusted cloud server, like IPFS. A data purchaser might wish to compute a function f over the encrypted message. With the consent of the data owner, a data purchaser would receive a secret key *sk_f* from a key distributor. The data purchaser can then use the private key *sk_f* to decrypt the ciphertext to compute *f(x)*. Note that the data purchaser cannot retrieve any other information on the underlying message except the final decryption result *f(x)*.

Here is a more concrete example for functional encryption: the message here could be the user's genomic data, and f could be a statistical analysis algorithm. The data purchaser could be a research institute intended to perform private statistical analysis over one's genomic data. The secret key might be generated by the hospital that collects the users' genomic data or the data owner themselves. At the end of this transaction, the data owner will receive economic compensation for the contribution to the computation result, while the data purchaser will receive the statistical analysis results. In other cases, the data could also be one's social network data and the function could be an analytics algorithm for online advertisement. We envision the monetary exchange between the data owner and purchaser will be executed as a Substrate module.

This project will implement a second-layer/middleware protocol running on Polkadot that enables individuals to monetize their private data in a fine-grained fashion. The delivery will include:

- A cryptographic library that implements the inner product functional encryption and quadratic polynomial functional encryption.
- A Substrate module that implements the encryption of the cryptographic keys of the functional encryption scheme and the associated zero-knowledge proof for its legitimacy.
- A micropayment scheme running on the Polkadot blockchain that allows individual users to monetize their data.

Here are the basic principles behind the overall architecture design of the fine-grained personal data monetization framework. We aim to keep the workload of the data owner minimum. This implies that the online time of the data owner should also be kept to a minimum. The optimal case would be that the data owner is left alone after he/she generates and uploads the ciphertext. Ideally, the encryption workload of the data owner should also be kept as minimal as possible. The following scheme can be viewed as a generalization of the knowledge monetization scheme proposed in [TZLHJS2017].

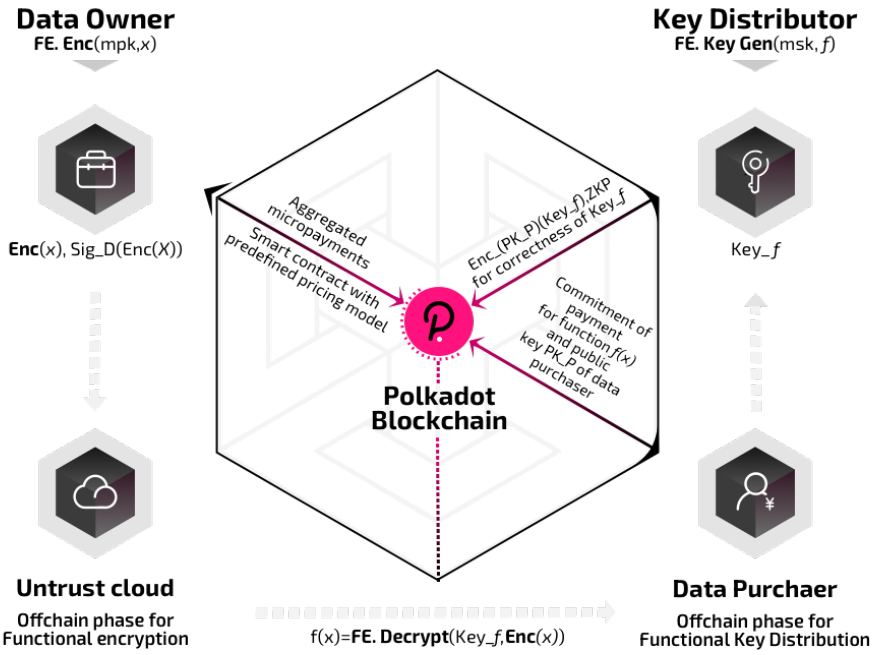
The general idea is shown in Fig. 1. A functional encryption scheme usually consists of four algorithms: *FE.Setup(1^\lambda), FE.KeyGen(msk, f), FE.Enc(mpk, x) and FE.Decrypt(Key_f, FE.Enc(mpk, x)):*

1) At the beginning of the system, the key distributors run the *FE.Setup(1^\lambda)* algorithm to generate the master public key *mpk* and master secret key *msk*. Here *\lambda* denotes the security parameter.

2) When a data owner wishes to sell his/her data, he/she encrypts the data x by running the *FE.Enc(mpk, x)* algorithm, and uploads the output ciphertext *Enc(x)* to the untrusted cloud, like IPFS.

3) The owner then specifies the pricing model with respect to different functions *fs* in a Substrate module *SC*, and posts it to the Polkadot blockchain.

4) When a data purchaser intends to calculate *f(x)*, he/she will first generate a commitment of an appropriate amount of RUB coin based on the pricing of the function f and his/her own public key *PK_P* as a transaction to call the module *SC*.



5) The Substrate module should return a receipt to the data purchaser, who will in turn present it to the key distributor. The key distributor, after verifying the receipt and the respective RUB coin commitment, runs the *FE.KeyGen(msk, f)* algorithm to generate the function key *Key_f* for the data purchaser.

6) Simultaneously, the distributor will send SC the encryption of *Key_f* under the data purchaser's public key *PK_P*, denoted as *Enc_(PK_P)(Key_f)*, and an associated Zero-knowledge Proof proving *Key_f* is indeed a well-formed function key corresponding to *f*.

7) SC with the inputs from both the key distributor and data purchaser is then verified by the Polkadot blockchain. Once the verification for both sides passes, confirming that the amount of

the committed RUB coin is sufficient to pay for the decryption result *f(x)* and the gas fee, and the associated zero-knowledge proof is correct, the payment will be released to the data owner.

8) The data purchaser can then download the ciphertext and decrypt *f(x)* by running the *FE.Decrypt(Key_f, FE.Enc(mpk, x))* algorithm.

Note that once the verification of SC is passed, the SC will be executed and payment released to the data owner instantaneously. To ensure fairness towards the data purchaser, the data owner should provide verifiable evidence to prove the data source. *FE.Enc(x)* is accompanied by a certified signature *Sig_D(FE.Enc(x))* to prove the data is coming from the right source and thus the data quality can be guaranteed. Here D denotes the private key of the data source.

The design of the underlying functional encryption scheme requires that the private key should correspond to the function chosen by the data purchaser. Since the encryption of the private key should be presented as evidence (as shown in Fig. 1) and verified on the blockchain, the private key should be of minimum size, preferably constant size. To keep the workload of both the data owner and key distributor minimal, the encryption time and key generation time should be as short as possible.

There exist several functional encryption schemes with constant key size such as the one presented in [SC2017, RDGBP2019, B2017, MSHBM2019]. General predicate encryption allows the data owner to encrypt the raw data items tagged with various attributes, and a data purchaser to query different parts of the data repository using a function key corresponding to a predicate. Inner product predicate encryption [CGW2015] is a special kind of predicate encryption, where the data purchaser could retrieve the data records if the inner product of their attribute vector y and the predicate vector x specified by the data purchaser is equal to 0. For instance, the data purchaser could potentially ask for the data records corresponding to a conjunctive predicate such as "Age"AND"gender"AND"Income" from a personal data repository. One of the most efficient inner product encryption schemes [CGW2015] has a private key consisting of four elliptic curve group elements, and its key generation is dominated by four modular exponentiations.

While the predicate encryption allows the data purchaser to retrieve different parts of a database based on a predefined predicate, a more general functional encryption allows the data purchaser to calculate an arbitrary function over the input *x*. This project will focus on a slightly narrow set of functional encryption schemes: inner product encryption and quadratic polynomial function encryption [MSHBM2019]. In an inner product encryption scheme, for encryption of a vector x, the data purchaser with a private key corresponding to another vector y will be able to compute the inner product between x and y. On the other hand, in a quadratic polynomial functional encryption scheme, the data owner will encrypt two vectors *v_1 \in \mathbb{Z}_n* and *v_2 \in \mathbb{Z}_n*, a data purchaser with a secret key corresponding to a matrix *H \in \mathbb{Z}^{n*n}* is allowed to decrypt the quadratic product of *x_1, x_2*, and *H*, i.e., *x_1^T\cdot H \cdot x_2*. Both inner product encryption and quadratic polynomial functional encryption can support sophisticated privacy-preserving machine learning tasks, such as classification [LCFS2017, SGP2018] and neural networks [RDGBP2019].

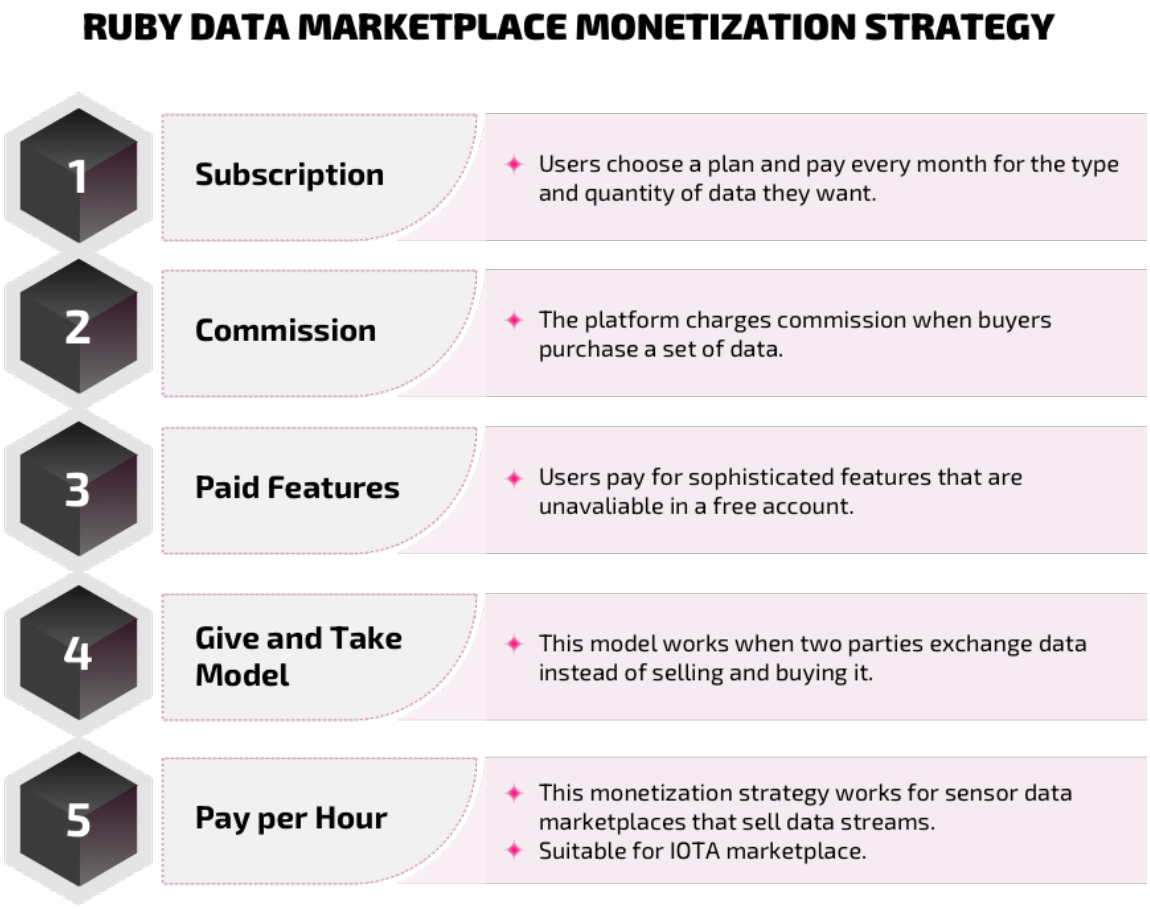
The benchmark results for various inner product encryption and quadratic polynomial function encryption schemes can be found in [MSHBM2019]. The private key of inner product encryption consists of one elliptic curve group element, which is of 256 bits under 128-bit level security. The key generation for a vector of 100 elements takes 0.4149s, and the encryption time for the data owner is around 0.2103s for a vector of the same size. The private key of quadratic polynomial functional encryption also only consists of one elliptic curve group element. The average key generation and encryption time for each coordinate of the message vector is 0.001s and 0.025s.

In terms of the accompanied zero-knowledge proof scheme, the statement of the zero-knowledge proof scheme should be to convince the verifier that the encrypted content is a well-formed function key with regards to a predefined function in the module. Since the verification of the zero-knowledge proof should be verified by the blockchain, we need to make sure that the verification is efficient and the proof size is as small as possible. The candidate zero-knowledge proof scheme for this is zk-snark implementation such as ZeroPool. The proof generation time is comparatively short for the previously mentioned encryption schemes since the respective statement (determined by the key generation algorithm and the public key encryption) is quite simple.

2.5 Monetization Strategy

After designing the data transaction logic of the market, it is also necessary to design the Monetization Strategy between the Data Owner and the Data Buyer. In the Ruby Data Marketplace, Ruby will provide the following 5 data charging modes for Data Owner for users to choose freely:

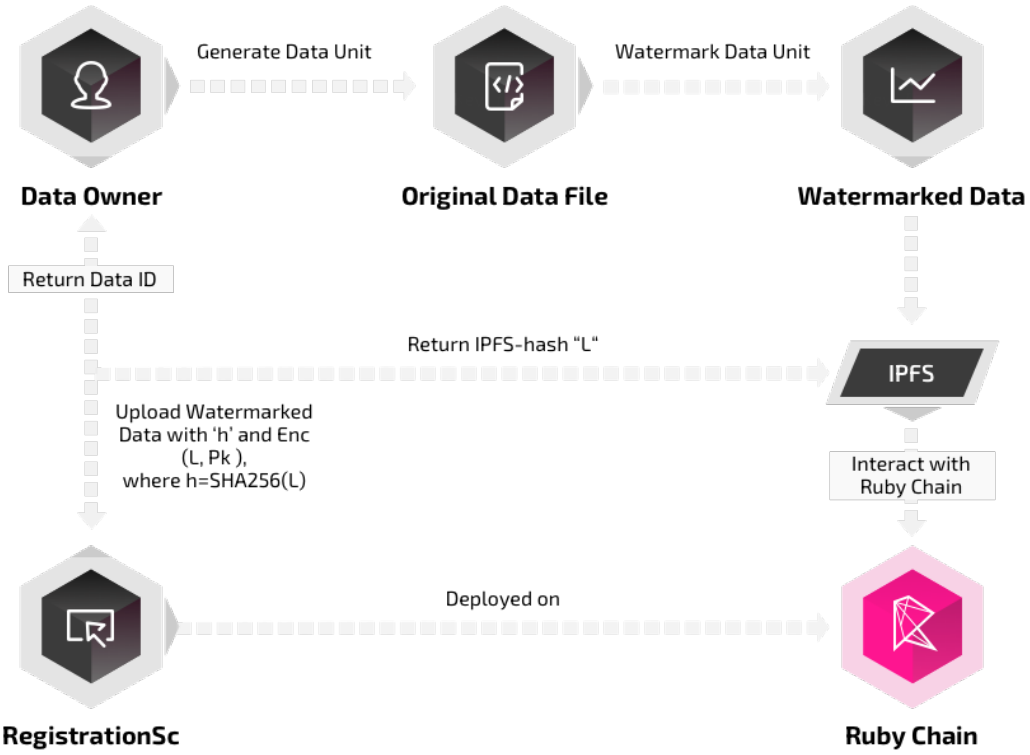
- **Subscription.** Users choose a plan and pay monthly for the type and quantity of data they want.
- **Commission.** Data owners can also charge commission when buyers query or acquire data. The commission is proportionate to the transaction volume.
- **Paid Features.** By this mode, the Data Buyer can acquire part of the data for free, but for more data, they need to pay for the service.
- **Give-and-Take Model.** This model works when two parties exchange data instead of selling and buying it.
- **Pay-per-hour.** Users can also choose to charge according to the time period of data acquisition. For example, if you need to determine the hourly price of data services, the Data Buyer can choose the data services they need from time to time to determine the final purchase price.



3 Data Copyright Protection

3.1 Structure Overview

In Ruby Data Marketplace, only when the uniqueness and accuracy of the data source are fully guaranteed can the data copyright of the original owner of data be secured during the circulation of data. Therefore, Ruby has a Data Copyright Protection mechanism in place, including four functional modules: Registration, Watermarking, Storage, Registering Watermarked-data. The structure is shown in the figure below:

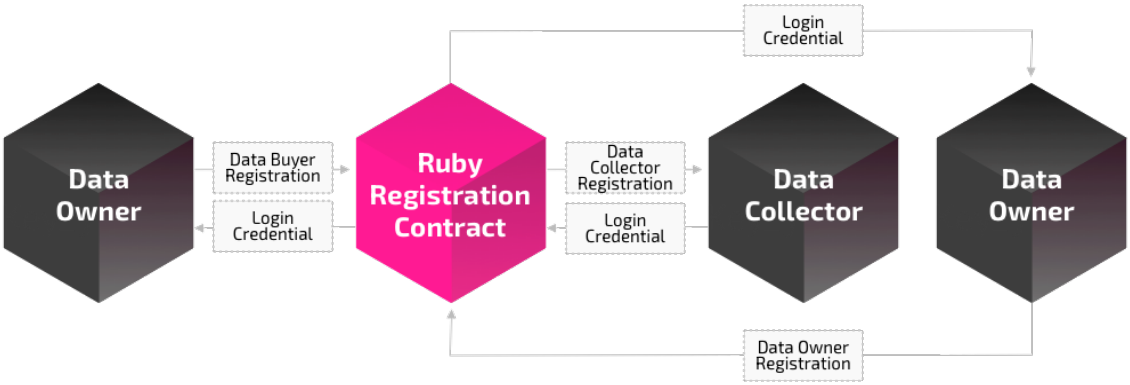


3.2 Registration

All participants of the Ruby Data Marketplace, including the Data Owner, the Data Collector, and the Data Buyer, must register first to join the market. In the registration, Ruby does not require users to fill in Personal Identity information in order to ensure user anonymity, but it needs to set basic ID information, such as user name, email address, information introduction, data introduction, the hash value of the data after running the data encryption algorithm, etc.

After the user fills in the information, they need to sign through the Ruby wallet and obtain the login credentials, which will be used for authentication later. The smart contract RegistrationSc stores participant details and generates login credentials. The login credentials include a unique identifier and password, which are stored on the Ruby blockchain.

The interaction logic between the Data Owner, the Data Collector, the Data Buyer and the RegistrationSc is shown in the figure below:



3.3 Watermarking

When a user completes registration, he needs to watermark the original data he owns using Ruby's watermarking technology tools, and embed the signature of his wallet address into the watermark information. This ensures the binding between the data and the user information.

Ruby will use the AHK algorithm (Agrawal et al., 2003; Agrawal & Kiernan, 2002) to add a watermark to the data. The watermark does not affect the value and use of the original content, and cannot be noticed or detected by the human perception system. It can only be extracted through a dedicated detector or reader. The watermark information includes the user ID registered by the Data Owner in the Ruby Data Marketplace, wallet address signature information, text with special meaning, etc. The information is used to identify the source, version, original author, owner, and issuer of data.

Given a database relationship $R\ (P,\ A_0,\ A_1\ ,\ A_2\ ....\ A_{v\text{-}1})$, where P is the main attribute, $A_0\ A_1\ A_2\ ....\ A_{v\text{-}1}$ is the candidate attribute for marking. The watermark embedding algorithm is shown in Algorithm 1 figure below:

The watermark detection algorithm is shown in the below Algorithm 2 figure:

Given a database relation$R\ (P,\ A_0,\ A_1\ ,\ A_2\ ....\ A_{v\text{-}1})$ where P is the primary key attribute and $A_0\ A_1\ A_2\ ....\ A_{v\text{-}1}$ are candidate attributes (numeric) used for marking. The embedding and detection of watermarks are depicted in *Algorithms 1* and *2* respectively. In *Algorithm 1*, once a tuple r is found eligible for marking at step 2, the index i of the candidate attribute and its corresponding LSB position j are computed at steps 3 and 4, respectively. Step 5 invokes Mark function which flips the bit at jth LSB position in ith attribute depending on the hash of the private key concatenated with the primary key of the corresponding tuple. Observe that H and F are one-way hash and MAC functions, respectively.

The detection algorithm (*Algorithm 2*) considers a suspicious relation S and computes at step 6 the total number of tuples (denoted by *total-count*) already marked in the insertion algorithm. The '*Match*' function at step 7 checks whether the marked bits at LSB position are present and accordingly computes the total number of successful detection (denoted by *match-count*). If *match-count* is more than the threshold τ (computed based on α and *total-count*), then the watermark detection is considered as successful (shown in steps 10–13).

Algorithm 1: Watermark Insertion Algorithm

```
// The private key K and the parameter γ, ν and ξ are known only to the owner of the
   database.
// 1/γ: Fracton of tuples marked.
// ν: Number of attributes available for marking.
// ξ: Number of least significant bits available for marking in an attribute.
for tuple r ∈ R do
    if F(r · P) mod γ equals 0 then
        attribute_index i = F(r · P) mod ν;
        bit_index j = F(r, P) mod ξ;
        r · A_i = Mark(r · P, r · A_i, j);
    end
end
Mark(primary_key pk, number v, bit_index j) return number
    first_hash = H(K ∘ pk);
    if first_hash is even then
        set the j^th least significant bit of v to 0;
    else
        set the j^th least significant bit of v to 1;
    end
    return v;
return
```

Algorithm 2: Watermark Detection Algorithm

```
// α (where 0 < α < 1): Test significant level.
// τ: Minimum number of correctly marked tuples needed for detection.
total-count = match-count = 0;
for tuple s ∈ S do
    if F(s · P) mod γ equals 0 then
        attribute_index i = F(s · P) mod ν;
        bit_index j = F(s, P) mod ν;
        total-count = total-count + 1;
        match-count = match-count + Match(s · P, s · A_i, j);
    end
end
τ = Threhold(total-count, α);
if matchcount ≥ τ then
    suspect piracy;
end
Match(primary_key pk, number v, bit_index j) return int
    first_hash = H(K ∘ pk);
    if first_hash is even then
        set the j^th least significant bit of v to 0;
    else
        set the j^th least significant bit of v to 1;
    end
    return v;
return
```

3.4 Storage

To make sure that the data stored by the user is secure, Ruby will upload and store user data processed by watermarking technology in the distributed storage network IPFS instead of a centralized

cloud service. During the user upload process, Ruby Network will automatically tag and create attributes based on the content uploaded by the user to facilitate the running of the Functional Encryption algorithm. When the upload is completed, IPFS will return to the user a hash value L of the uploaded data.

3.5 Register Watermarked-data

When the user obtains the hash value L, he/she needs to call RegistrationSc on the registration interface of the Ruby Data Marketplace to upload that. When the user completes the upload, he will get the unique data ID corresponding to the hash value L. At the same time, Ruby will run the SHA256 algorithm to encrypt L and store it on the Ruby block.

4 Consensus Mechanism

4.1 NPoS Consensus

The Ruby Network protocol is developed based on the Substrate blockchain development tool and belongs to the Polkadot ecology. So, Ruby will adopt the same NPoS (Nominated Proof of Stake) consensus mechanism as Polkadot.

4.2 Inflation Rate

In the Ruby network, the inflation rate (I) can be calculated by the below formula:

$$I = I_{NPoS} + I_{treasury} - I_{slahing} - I_{tx\text{-}fees} - I_{burn}$$

I_{NPoS} refers to the inflation rewards distributed to validators and nominators. $I_{treasury}$ inflation refers to the rewards distributed to the Treasury. $I_{slahing}$ is the Slash penalty due to the malicious behaviour of the Validator, and $I_{tx\text{-}fees}$ refers to the destruction of transaction fees in the network. I_{burn} represents the repurchase and burning of Ruby Protocol revenues.

In the above formula, I_{NPoS} is a core variable, which is directly related to the Staking rate of the network. The sum of I_{NPoS} and $I_{treasury}$ is a fixed parameter of 10%. That means that the highest possible inflation rate in the Ruby Protocol network is 10%, but this does not mean that the Ruby network will always be in inflation.

In order to reduce the overall inflation rate of the Ruby network, Ruby will also regularly use part of the network's revenue for secondary market repurchase and destruction, which is represented in the formula as I_{burn}. This is accompanied by the default destruction mechanisms $I_{slahing}$ and $I_{tx\text{-}fees}$ in the NPoS consensus mechanism. When the ecosystem in Ruby is prosperous enough, transaction and repurchase & burning plans may offset the inflation rewards in the network, making the system deflationary.

4.3 NPoS Inflation

Under the NPoS consensus mechanism, the rewards for validators and nominators mainly come from the further issuance of tokens, which is also the main cause of network inflation. The inflation rate of Ruby Network is not adjusted by anyone manually but through an algorithm. The formula of the NPoS inflation model is shown below:

$$I_{NPoS}(x) = \begin{cases} I_0 + x\left(i_{ideal} - \frac{I_0}{\chi_{ideal}}\right) & \text{for } 0 < x \leq \chi_{ideal} \\ I_0 + (i_{ideal} \cdot \chi_{ideal} - I_0) \cdot 2^{(\chi_{ideal} - x)/d} & \text{for } \chi_{ideal} < x \leq 1 \end{cases}, \text{ and}$$

$$i(x) = I_{NPoS}(x)/x.$$

χ_{ideal} is the expected Staking rate, which represents the ideal Staking rate in the Ruby network; I_{ideal} is the expected annualized rate of return. R_0 represents the inflation rate when the Staking rate is 0 and that's the default inflation rate; d represents the decay rate of the inflation rate.

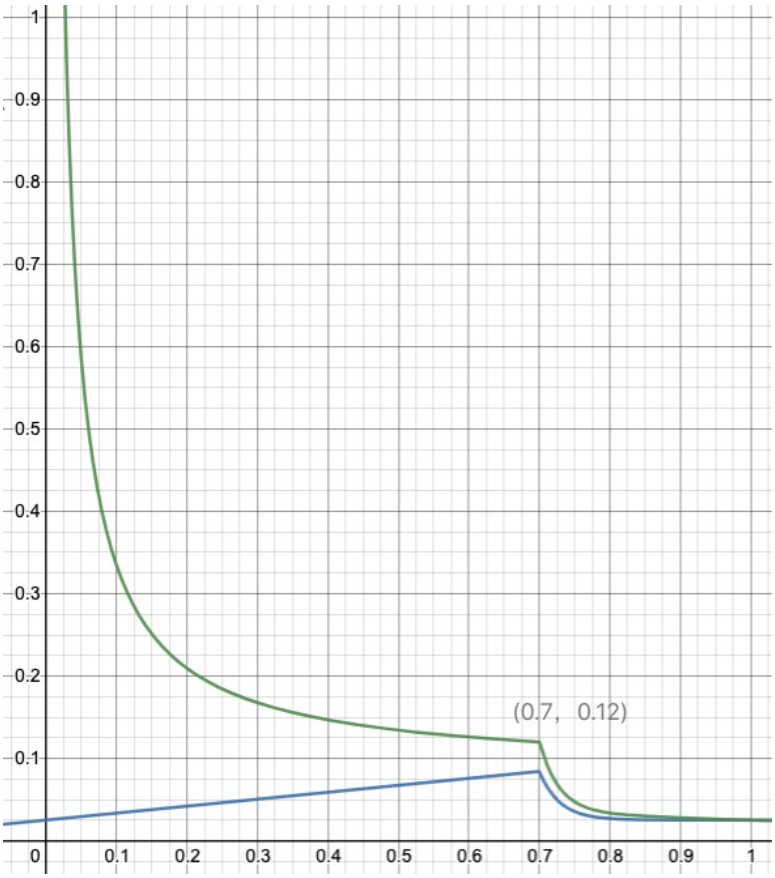
After a comprehensive evaluation, the algorithm formula parameters of the inflation rate and the rate of return in Ruby Network are as follows:

- $\chi_{ideal} = 0.7$
- $I_{ideal} = 0.12$
- $R_0 = 0.025$
- $d = 0.02$

Under this circumstance, the balance between the Staking rate and rate of return is as follows:

- When the Staking rate is lower than 70%, the average annualized rate of return of Staking will continue to increase, closing to 12%. Staking of more tokens will be encouraged.
- When the Staking rate is equal to 70%, the average annualized rate of return of Staking will be 12%;
- When the Staking rate is greater than 70%, and the average annualized rate of return of Staking is less than 12%, the Redemption instead of Staking will be encouraged.

The following figure shows the relationship between the simulated Ruby Network inflation rate, Staking rate, and annualized rate of return:



The X-coordinate is the Staking rate, the blue Y-coordinate is the annual inflation rate, and the green Y-coordinate is the annualized rate of return.

4.4 Block Reward

Every time a block is generated on the Ruby mainnet, the system will issue a certain amount of Ruby Tokens as block reward. The reward of each block will be determined by the real-time inflation rate of the network. The inflation rate, which is flexible, will be determined by the inflation formula. It will be affected by the actual Staking rate of the network.

Block rewards will be distributed in the following way:

- 80% for validators and nominators;
- 20% for the Ruby network Treasury, which will be used for community operation, project development, ecosystem incentives, etc.

4.5 Block Slash

When a validator misbehaves, such as off-line, double-signing, etc., it will be punished - their Staked Ruby will be fined for a certain amount alongwith their nominators.

The Ruby penalties will be affected by the status of the network. For example, if a high proportion of validators in the entire network is off-line or double-signing, the penalty for a single

validator will be higher.

In extreme cases, for example, if over 1/3 of validators are double-signing, the Slash will be 100% of the Staked Ruby.

5 RubyDAO

Ruby Network will establish a community autonomous organization, namely RubyDAO. It will be based on Polkadot's on-chain governance mechanism. Any Ruby token holder will be able to initiate governance proposals, adjust governance parameters, or apply for grants.

However, the RubyDAO governance level needs to be adapted to the development of the project. Especially in the early development stage when the product and business are still immature, the development will be hampered if the governance relies solely on the community autonomy. Market opportunities will also be missed. Therefore, the autonomy level of RubyDAO will continue to improve with the stage of development. Finally, the community will enjoy full autonomy.

Holders of Ruby tokens can make proposals for the following aspects in accordance with the rules:

1) System code upgrade;

2) Adjustment of system governance parameters, such as handling fee, amount of Staked Ruby, etc.;

3) Modification of governance rules;

4) Addition or modification of product features;

5) Financial support and incentive plan;

6 Treasury

In order to ensure the sustainable development and decentralization of the protocol, Ruby Network has set up a Treasury to incentivize individuals or teams who contribute to the project. Any use of the Treasury is approved through online governance. First, developers can submit their proposals to the community. Those voted and approved will be entitled to claim financial support from the Treasury.

The revenue of the treasury mainly come from the following two ways:

1) 20% of the Commission collected by RubyDAO from the transaction revenue of Data Marketplace.

2) 20% of the block reward.

7 Token Economics

7.1 RUBY Token

RUBY token is the native token of Ruby Protocol. The initial total supply is 1 billion. It will be used for the security of the Ruby Protocol chain, and the construction and incentives of the data transaction market.

In Ruby Network, the application scenarios of RUBY Token are as follows:

1) Scenario 1: When the Data Owner uploads data, he/she needs to pay RUBY token to obtain watermarking technical services and data storage services;

2) Scenario 2: Data Owner can use RUBY token for promotion, such as obtaining better system recommendations, launching Data Buyer review incentive activities, etc.;

3) Scenario 3: Data Buyer can use RUBY Token to deduct the data purchase fee from the Data Owner or the Data Collector;

4) Scenario 4: RUBY Token holders can delegate RUBY to Validators to obtain staking rewards;

5) Scenario 5: RUBY Token holders can participate in the governance voting of RubyDAO;

6) Scenario 6: When the Ruby Data Marketplace product goes live, mining incentives will be launched. Users who upload data will receive RUBY Token rewards;

7.2 Token Distribution

Total supply of RUBY tokens is 1 billion, with the distribution plan as the following:

Token Allocation	**%**	**Vesting**
Seed Round	6.0%	Subject to 24-month vesting schedule with a 3-month cliff and quarterly vesting in 4–24 months
Private Round I	8.0%	5% on TGE, quarterly vesting for 18 months, starting one quarter after the listing
Private Round II	10.0%	10% on TGE, quarterly vesting for 12 months, starting one quarter after the listing
Public Sale	8.0%	25.0% on TGE, quarterly vesting for 6 months, starting one quarter after the listing
Parachain Bond Funding	15.0%	10% on TGE, monthly vesting for 48 months
Ecosystem Development	10.0%	10% on TGE, monthly vesting for 24 months
Foundation Reserve	15.0%	10% on TGE, monthly vesting for 24 months
Founding Team	10.0%	Subject to 48-month vesting from network launch, with a 12-month cliff and monthly vesting thereafter
Partners and Advisors	5.0%	Subject to 24-month vesting from network launch, with a 6-month cliff and monthly vesting thereafter

Token Allocation	%	Vesting
Developer Adoption Program	5.0%	Subject to vesting 2-year monthly linear vest. Unused tokens can be used for future sales
Early Staking Reward	4.0%	Subject to 2-year vesting schedule from network launch with a 6-month cliff and monthly vesting thereafter
Community Reward	4.0%	Subject to 2-year vesting schedule from network launch with a 6-month cliff and monthly vesting thereafter

7.3 Value Capture

The big scale of data transactions in the Ruby Data Marketplace will provide an important value capture scenario for RUBY token, acting as the core value support of RUBY token:

1) RUBY token is the fuel in the Ruby network. All related transactions on the Ruby chain will require RUBY token as commission;

2) In the data upload, storage, transaction, promotion, etc., of the Ruby Data Marketplace, RUBY tokens have real use values, which will greatly encourage users to hold them.

3) 70% of RubyDAO revenue will be used for repurchase and burning in the secondary market, which will reduce the inflation rate of RUBY tokens. The whole system may even become deflationary.

4) When submitting an on-chain proposal, community members need to stake a certain amount of RUBY tokens.

7.4 Community Engagement

- Bounty Program for General Community: We will reward users who contribute positively to community building and content creation through an Ambassador Program. The community management team will be available 24/7 to answer questions.
- Incentive Program for Data Monetization: After the main functions are completed, we will provide incentives for users to monetize their data on our platform. This is an encouragement for users to provide the data and purchase the data.
- Parachain Loan Offering Campaign: We may hold a Parachain Loan Offering and reward users for helping in our auction with Ruby Protocol tokens.
- Affiliated Program of Cryptographic Infrastructure: It is proven effective for user growth and can be integrated into Ruby's cryptographic infrastructure.

8 Roadmap

2020 Q3

- Project Establishment

2020 Q4

- Proof of Concept
- Protocol Establishment

2021 Q1

- Draft of the Whitepaper

2021 Q2

- Official Whitepaper Release
- Official Website Launch
- Development & Marketing Launch

2021 Q3

- Github Repos Release
- Open Substrate Modules
- Open Cryptographic Library

Q4 2021

- Release Micropayment Scheme

Q3 2022

- Launch Substrate Client (UI) Modules
- Testnet Launch

Q4 2022

- Mainnet V1.0 Launch
- Bug Bounty Program

Q1 2023

- Access control for NFT gated event

9 Development Milestones

Milestone 1 — Implement Cryptographic Modules

The main deliverable of this milestone includes:

- A cryptographic library that implements the inner product functional encryption and quadratic polynomial functional encryption.
- A substrate pallet that integrates the verification logic of the associated zero-knowledge proof for the legitimacy of the encrypted functional key.

Deliverable	**Specification**
License	Apache License 2.0
Documentation	We will provide both inline documentation of the code and a basic tutorial that explains how a user can (for example) spin up one of our Substrate nodes. Once the node is up, it will be possible to send test transactions that will show how the new functionality works.
Testing Guide	Core functions will be fully covered by unit tests to ensure functionality and robustness. In the guide, we will describe how to run these tests.

Deliverable	Specification
Article/Tutorial	We will publish a medium article that explains the functionality of the proposed cryptographic library and Substrate pallet delivered in this milestone.
Cryptographic modules	We will implement the cryptographic modules including inner product functional encryption and quadratic polynomial functional encryption [MSHBM2019] and the associated zero-knowledge proof. We will also implement the Substrate pallet that integrates the verification logic of the associated zero-knowledge proof for the legitimacy of the encrypted functional key.
Benchmark	Perform unit tests on the individual algorithms to ensure their safety benchmark on the gas cost and throughput of the proposed module.
Docker	We will provide a dockerfile to demonstrate the usage of our modules.

Milestone 2 — Client Implementation and Integration

The main deliverable of the milestone is the client that can trigger the aforementioned cryptographic modules and the micropayment scheme, and the necessary UI to enable the users to interact with all these algorithms.

Deliverable	Specification
License	Apache License 2.0
Documentation	We will provide both inline documentation of the code and a basic tutorial that explains how a user can (for example) spin up one of our Substrate nodes. Once the node is up, it will be possible to send test transactions that will show how the new functionality works.
Testing Guide	Core functions will be fully covered by unit tests to ensure functionality and robustness. In the guide, we will describe how to run these tests.
Article/Tutorial	We will publish a medium article that explains the functionality of the proposed client and UI delivered in this milestone.
Client modules	We will implement the client to support the key distribution and decryption of the functional encryption scheme [MSHBM2019]. The client will also generate the transaction that can trigger the aforementioned cryptographic modules and the micropayment scheme [MDJM2019], such as the encrypted functional key and zero-knowledge proof. We will provide a basic UI to take inputs from the users for all these algorithms and receive the outputs. More specifically, the UI will enable the data owner to input the raw data to generate the signed ciphertext and upload it to the cloud server. The UI will also allow the data purchaser to retrieve the functional key from the key authority and the ciphertext from the cloud and then perform the decryption. Finally, it will also allow these entities to interact with the Substrate module with the inputs and outputs defined in our architecture.
Benchmark	Perform unit tests on the individual algorithms to ensure their safety benchmark on the latency and usability of the proposed client functionalities.
Docker	We will provide a dockerfile to demonstrate the usage of our modules.

10 Future Plans

We will hire at least 8-10 more devs in the next three months. Meanwhile, we will apply for the Substrate Builder's Program. After that, Ruby Protocol wants to become a parachain for the Polkadot network. We have some preparations for auction and we may design a community-wide LPO.

In phase 1, we will complete the implementation of cryptographic modules as a Substrate pallet that integrates the verification logic of the associated zero-knowledge proof for the legitimacy of the encrypted functional key.

In phase 2, our goal is to deliver the micropayment scheme and enable the users to interact with all these algorithms in a working product.

Finally, our goal is to provide an essential open API and SDK from a high-level perspective with the above tools, fully powering the data monetization framework on Polkadot.

References

[GPSW06] Goyal, V., Pandey, O., Sahai, A., and Waters, B. (2006, October). Attribute-based encryption for fine-grained access control of encrypted data. In Proceedings of the 13th ACM conference on Computer and communications security (pp. 89-98).

[GGGJKLZ14] Goldwasser, S., Gordon, S. D., Goyal, V., Jain, A., Katz, J., Liu, F. H., ... and Zhou, H. S. (2014, May). Multi-input functional encryption. In Annual International Conference on the Theory and Applications of Cryptographic Techniques (pp. 578-602). Springer, Berlin, Heidelberg.

[GKPVZ13] Goldwasser, S., Kalai, Y., Popa, R. A., Vaikuntanathan, V., and Zeldovich, N. (2013, June). Reusable garbled circuits and succinct functional encryption. In Proceedings of the forty-fifth annual ACM symposium on Theory of computing (pp. 555-564).

[ALS2016] Agrawal, S., Libert, B., Stehle, D. (2016). Fully secure functional encryption for inner products, from standard assumptions. In Annual International Cryptology Conference. (pp. 333-362). Springer, Berlin, Heidelberg.

[B2017] Bourse, F. (2017). Functional encryption for inner-product evaluations (Doctoral dissertation).

[B2018] Buterin, V. (2018). On-chain scaling to potentially ~500 tx/sec through mass tx validation. Available at `https://ethresear.ch/t/on-chain-scaling-to-potentially-500-tx-sec-through-mass-tx-validation/3477`.

[BCTV2013] Ben-Sasson, E., Chiesa, A., Tromer E., and Virza M. (2014, August). Succinct non-interactive zero knowledge for a von Neumann architecture. In Proceedings of the

23rd USENIX Security Symposium, Security '14. Available at http://eprint.iacr.org/2013/879.

[BMEB2016] Bataineh, A. S., Mizouni, R., El Barachi, M., and Bentahar, J. (2016, June). Monetizing Personal Data: A Two-Sided Market Approach. *Procedia Computer Science*. **83** (pp. 472-479).

[CGW2015] Chen, J., Gay, R., and Wee, H. (2015, April). Improved dual system ABE in prime-order groups via predicate encodings. In Annual International Conference on the Theory and Applications of Cryptographic Techniques (pp. 595-624). Springer, Berlin, Heidelberg.

[FVBG17] Fisch, B., Vinayagamurthy, D., Boneh, D., and Gorbunov, S. (2017, October). Iron: functional encryption using Intel SGX. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (pp. 765-782).

[LCFS2017] Ligier, D., Carpov, S., Fontaine, C., and Sirdey, R. (2017, February). Privacy Preserving Data Classification using Inner-product Functional Encryption. In ICISSP (pp. 423-430).

[MDJM2019] Mehta, S., Dawande, M., Janakiraman, G., and Mookerjee, V. (2019, August). How to sell a dataset? pricing policies for data monetization. *SSRN Electronic Journal*.

[MSHBM2019] Marc, T., Stopar, M., Hartman, J., Bizjak, M., and Modic, J. (2019, September). Privacy-Enhanced Machine Learning with Functional Encryption. In European Symposium on Research in Computer Security (pp. 3-21). Springer, Cham.

[PHGR2013] Parno, B., Howell, J., Gentry, C., and Raykova, M. (2013, May). Pinocchio: Nearly practical verifiable computation. In 2013 IEEE Symposium on Security and Privacy (pp. 238-252). IEEE.

[RDGBP2019] Ryffel, T., Dufour-Sans, E., Gay, R., Bach, F., and Pointcheval, D. (2019). Partially encrypted machine learning using functional encryption. arXiv preprint arXiv:1905.10214.

[RRS2013] Reimsbach-Kounatze, C., Reynolds, T., and Stryszowski, P. (2013). Exploring the economics of personal data-a survey of methodologies for measuring monetary value.

[SC2017] Agrawal, S. and Chase, M. (2017). Simplifying design and analysis of complex predicate encryption schemes. In Annual International Conference on the Theory and Applications of Cryptographic Techniques. Springer, Cham.

[SGP2018] Sans, E.D., Gay, R. and Pointcheval, D. (2018). Reading in the dark: Classifying encrypted digits with functional encryption. IACR Cryptology ePrint Archive 2018, 206.

[TZLHJS2017] Tramer F., Zhang F., Lin H., Hubaux J.-P., Juels A., and Shi E. (2017) Sealed-glass proofs: Using transparent enclaves to prove and sell knowledge. In Security and Privacy (EuroSandP), 2017 IEEE European Symposium on, (pp. 19-34). IEEE.

[GPSW2006] Goyal, Vipul and Pandey, Omkant and Sahai, Amit and Waters, Brent. (2006) Attribute-based encryption for fine-grained access control of encrypted data Proceedings of the 13th ACM conference on Computer and communications security, (pp. 89-98).

[Mintlayer] Disrupting Asset Tokenization: an Introduction to Mintlayer's ACL https://www.mintlayer.org/news/2021-06-08-disrupting-asset-tokenization/

[litprotocol] Litprotocol https://litprotocol.com/

[nftGatedAccess] Another NFT Use Case: Access as Utility https://www.one37pm.com/nft/nft-access-passes-nftnyc

[pavemotors] pavemotors https://www.pavemotors.com/platform