Cozea: A Zero Knowledge blockchain oriented consent sharing protocol based on accumulators

Somnath Banerjee, Ras Dwivedi, and Ashutosh Shukla

Abstract. Data protection is a growing concern in a multitude of domains across the world. In most of the data protection mandates, the data owner is deemed the ultimate authority for controlling access to the data by sharing consent. Ideally, sharing such a consent should be privacy preserving and anonymous and a user couldnt be tracked even by the consent management application. In this paper, we present a zero-knowledge proof-based protocol for sharing consent for a dataset, keeping in mind a possible smart contract-based implementation.
In this protocol, a dataset or a file is identified by a unique identifier (e.g., hash of the file) and the consent is managed within a smart contract. A set of valid consents for a certain dataset is represented in the protocol as a single RSA-based accumulator of primes. Adding a consent simply means updating the accumulator and adding a prime, which is also shared with the intended party. The receiver of the consent must prove the knowledge of the prime in a zero-knowledge protocol with the file server. A corresponding Pedersen commitment of the prime is also added to a list maintained corresponding to every file. This commitment is tied to the public key of the consent receiver. This enables enforcing a condition of identification of the intended party, which must prove possession of the secret key, in addition to the knowledge of the prime.
Further, a new scheme is proposed for storing the product of primes, along with the initial accumulator raised to the inverse of the prime, allowing parties to independently regenerate witnesses upon successive updates of the accumulator.

1 Introduction

1.1 Data protection and data consent management

In order to limit the unauthorized access and use of data by third parties, it is necessary that a user shares consent for their data to be read, used or shared. In the digital space, historically, such a mechanism wasnt naturally present in most of the traditional client-server-architecture-based applications. As the data protection became a more actively debated topics, regulations like GDPR came into being. All such data regulation mandates recommend having a mechanism in place to ask for consent from the user before collecting or sharing their data. In a more siloed application, such a consent takes the form of a mere digital form with a checkbox, which the user is supposed to tick to agree to sharing data. Now, such an implementation naturally cannot be fully trusted given the opaque nature of what is happening behind the curtains. Also, such a siloed management of user consent information would lack certain standards and the users are at the mercy of the specific implementation of how the applications policies manage the stored data.

The next iteration of data protection and consent management systems came up with unified systems for managing data sharing. Such systems standardized collection of user consent with clearly detailed parameters like expiry, purpose, reference of data etc. When such a system is backed by a highly trusted entity like government, it provides better assurances of data protection to the end user. Nevertheless, a centralized implementation like this could still lack transparency in its operations. At the same time, the issue of tracking users activities remains. The backend database of this system could easily reveal who all the user is interacting with. Mechanisms like digital signatures on a consent artifact (a document containing consent information) do provide some degree of protection against alteration of data consent by any party. But this carries the problems like: such a consent artifact can easily be erased from the databases, there is no unified standard way of creating a digital signature, the centralized systems tend to manage the private keys on behalf of the user, the consent document may be maliciously modified before getting signed. At the same time, the standardization of the consent artifact might lack proper implementation and a lot of trust has to be (rightfully) assumed on the consent managing system.

1.2 A decentralized approach

A lot of the issues of a centralized system can be addressed with a decentralized application, supported by a smart contract based blockchain network. A smart contract gets executed in a verifiable manner across multiple nodes on a network and the result of the execution is maintained on the nodes as the state of the smart contract. The sanctity of the result can be maintained if the majority of the nodes on the network are not malicious, which is a safe practical assumption in most cases, evidently demonstrated by the largely successful blockchain platforms. Malicious nodes in the network are ignored or kicked out of the network. This decentralized nature of blockchain provides much better trust over traditional centralized systems. But a major drawback of almost all blockchain implementations is that all the data in smart contracts have to be public in order for their execution and further storage and propagation of the results. This is not a very privacy respecting approach in most cases, and centralized systems would mostly fare better against nave blockchain based decentralized applications.

1.3 Our Contribution

The main purpose of this paper is to build a private and anonymous scheme for generating and sharing consent for a file present on a data server. We propose a new scheme for managing consent through the use of RSA based accumulators [ref CL02]. These accumulators allow accumulating prime values into the exponent of an initial value of the accumulator. Further Camenisch and Lysyanskaya showed a zero knowledge proof protocol for proving the knowledge of a prime that is present in the accumulator. We use this feature to represent a consent shared between two parties. The data related to the file and the accumulator are stored in the smart contract in our protocol. We intended to keep the work simple and focussed primarily on the cryptographic schemes throughout the protocol, hence we represent files through simply generated unique identifiers like hashes. For our purposes, the smart contract would store consent data in a simple structure against every individual identifiers.

In our model, the protocol execution has two legs: one between the file owner and consent receiver, and the other between the receiver and the data provider. We assume that the data provider is an honest but curious entity who shares the data in question, only after verifying the proofs presented by the consent receiver. The state of the smart contract is updated by the file owner to represent the latest state of consents. The updated set of values do not reveal any information about the consent receiver. We propose schemes to add and revoke consent against a consent receiver. These are performed through overwrites to the value of the accumulator already present on the smart contract and the calculations happen offline in the end user application. The updated values of the accumulator, combined with a commitment to the prime number that represents the consent, is used by the data provider to verify the users consent.

We further address the issues of a. re-generating the witness to a prime in the accumulator, b. accessing the prime intended for a receiver by the receiver at a later point. For this, we propose a new scheme of storing values encrypted using ElGamal scheme right against the file identifier, but mixed within a list of various other, unidentifiable set of such values which may belong to other consent receivers.

2 Preliminaries

In this section, we will revisit the cryptographic primitives used in this work.

2.1 Cryptographic assumptions

RSA Assumption. Given an RSA modulus n, RSA exponent e and an integer $y \in \mathbb{Z}_n^*$ it is computationally infeasible to obtain integer x such that $y = x^e \pmod{n}$. The RSA modulus n is obtained as a product of two safe primes, that is, $n = pq$ where $p = 2p'+1$ and $q = 2q'+1$, when p' and q' are large primes.

Strong RSA Assumption. Given an RSA modulus n and an integer $y \in \mathbb{Z}_n^*$ it is computationally infeasible to obtain $x \in \mathbb{Z}_n^*$ and $e \neq \pm 1$ such that $y = x^e \pmod{n}$.

Discrete Logarithm Assumption. The DL assumption states that it is infeasible to find the discrete log of a group element for a group of prime order. Formally, let $y \in \mathbb{G}$ and g be a generator of the group $\mathbb{G}$ of prime order q, it is computationally infeasible to obtain $x \in \mathbb{Z}_q$ such that $y = g^x$.

2.2 Commitment Scheme

A commitment scheme consists of algorithms Commit and Open, over a setup CGen, defined as follows:
$c = Commit(m, r)$ The commit function outputs a commitment that hides a message m using a randomness r such that it is easy to compute, but hard to reverse.
$b = Open(c, m, r)$ Given a commitment c, a message m and the corresponding randomness r, the Open function returns true if and only if $c = Commit(m, r)$

Moreover, a commitment scheme is said to be hiding if it does not reveal any information about m, given c. And it is binding if it is infeasible to obtain another pair $(m', c') \neq (m, c)$ such that $Commit(m, r) = Commit(m', r')$, or in other words a commitment scheme is binding if it is hard to open a given commitment to a different message.

A well known commitment scheme called Pedersen Commitment which is a perfectly hiding and computationally binding scheme over the DL assumption. It is defined over the following algorithms

CGen(gk). The setup parameters are generated as follows:

- Choose primes $\mathbf{p},\mathbf{q} : \mathbf{q} | (\mathbf{p} - 1)$ and thus the group $\mathbb{G}_\mathbf{q}$, which is (unique) subgroup of $\mathbb{Z}_\mathbf{p}^*$ of order $\mathbf{q}$.
- Choose generators $\mathfrak{g}, \mathfrak{h}$ from $\mathbb{G}_\mathbf{q}$ such that $log_\mathfrak{g}(\mathfrak{h})$ is not known to prover.
- Output to public $(\mathfrak{g}, \mathfrak{h})$,

Commit(m, r). For a message $m \in \mathbb{Z}_\mathbf{q}$ and randomness $r \in \mathbb{Z}_\mathbf{q}$ the commitment function is given by $Commit(m, r) = \mathfrak{g}^m \mathfrak{h}^r$.

Remark 1. Since any element$\neq 1$ of $\mathbb{G}_\mathbf{q}$ is a generator $log_\mathfrak{g}(\mathfrak{h})$ is defined.

2.3 ElGamal Encryption scheme

Elgamal encryption scheme is a public encryption scheme, secure under the DDH assumption, defined by the following algorithms:

Setup. Given a security parameter k, run $Gen(k)$ to generate a generator g of a cyclic group $\mathbb{G}$ of prime order q, where $|q| = k$ Choose a random $x \in \mathbb{Z}_p^*$ as the private key. Compute the corresponding public key as $y = g^x$. Output the public params (G, g, q, h)

Encrypt. To encrypt an integer $m \in \mathbb{Z}_p$ the encryption algorithm hides m in the exponent of g. Choose a random $r \in \mathbb{Z}_p^*$. The ciphertext is given by $c = (c_1, c_2)$. where $c_1 = m \cdot y^r \mod q$ and $c_2 = g^r \mod q$

Decrypt. Given the private key x, the ciphertext $c = (c_1, c_2)$ is decrypted as $\frac{c_1}{c_2^x} \mod q = \frac{m \cdot y^r}{g^r x} \mod q = m$

2.4 Zero-Knowledge Proofs

A zero-knowledge proof(ZKP) demonstrates the accuracy of a statement about a secret without revealing any other information about it. Formally, given an element x of a language $\mathbb{L} \in NP$, a certain party called prover convinces another party, the verifier that $x \in \mathbb{L}$ using a witness w, over a protocol that does not reveal any other information about x. Further, the prover is said to show a proof of knowledge (PoK) if the prover convinces the verifier that he or she knows w.

ZKPs were first introduced by Goldwasser, Micali and Rackoff in 1989.

***Σ*-protocol.** A proof system between a Prover (P) and a verifier (V), execution of which generates conversations of the form (a,c,z); where a and z are generated by P and c represents a challenge sent to P by V. V accepts if $\phi(\alpha, a, c, z) = 1$, where ϕ is an efficiently computable predicate.

We use the following generic notation for a ZKP-statement:

$$st : \{(a, b, ...; x, y...) : f(a, b, ..., x, y, ..)\}$$

to mean that the prover shows the knowledge of $\{x, y, ...\}$ such that $f(a, b, ..., x, y, ...)$ is true and $\{a, b, ...\}$ are public parameters.

3 Cryptographic Accumulators

A lot of applications need to determine if a certain element is a part of a list, such as access control lists, which determine whether an entity has access to a resource. A cryptographic accumulator is a space-efficient data structure for maintaining set-membership information, with sublinear time complexity of update and verification of the set. As against a trusted third-party maintaining a list of elements, cryptographic accumulators aim to preserve the privacy of the user carrying out authentication or proving membership. The user can prove a statement about a secret value with regards to the universal accumulator value without knowing other values in the list, and without revealing the secret value itself.

Cryptographic accumulators were first introduced by Benaloh & deMare with applications to document time-stamping and membershp testing. There are various classifications for different types of accumulators.Formally, a secure accumulator over a family of inputs χ_k is a family of families of fucntions $\mathbb{G} = \mathbb{F}_k$ has the following properties:

- **Efficient Generation.** Given an input parameter k, there is an efficient probabilistic polynomial time algorithm that produces $f \in \mathbb{F}_k$ with some trapdoor information (such as factorization of RSA modulus)
- **Efficient Evaluation.** The generated accumulator function, f should be efficiently computable in the given domain.
- **Quasi-commutative.** The function f is said to be quasi-commutative if for u in the domain of the accumulator and for any $x_1, x_2 \in \chi_k$ the following is satisfied:

$$f(f(u, x_1), x_2) = f(f(u, x_2), x_1)$$

- **Security.** Given the accumulator value and the set of values accumulated in the accumulator, it is difficult for an adversary to come up with a witness w for an $x \notin \chi_k$ such that $f(w, x) = v$ where v is the value of the accumulator. Also, for a stronger notion of security, no information about the accumulated set should be leaked by the accumulator value itself or of any of the witnesses.

Further the secure accumulator is said to be dynamic if it has the property of **efficient deletion**, that is, there exists efficient p.p.t. algorithms to compute:

- $v' = f(u, \chi \backslash \{x_i\})$, where v' is the updated accumulator value after delething the $i-$the element from the list χ
- Witnesses w'_j for all $j \neq i$, such that, $f(w'_j, x_j) = v'$

3.1 RSA-Accumulator construction

Here we revisit the modular exponentiation based accumulator based on strong RSA assumption. We start with an RSA modulus $N = pq$, where p and q are safe primes. The initial value of the accumulator is set to $x \in \mathbb{Z}_N$. For every subsequent element e to be added to the accumulator, we simply raise the accumulator to the power of e, modulo N. In case of the RSA-based accumulator, all the values to be accumulated must be prime to avaoid correlation with other values. The set of input values to be accumulated is $\mathcal{X} = \{e \in \mathbb{Z}_N^* \mid e \ is \ prime\}$. The accumulator function is given by:

$$f(v, e) = v^e \mod N$$

Remark 2. The knowledge of factorization of N can be used to create a witness of a value not accumulated.

3.2 Dynamic accumulator based on RSA

This dynamic accumulator construction was given by Camenisch and Lysyanskaya. The RSA modulus N is chosen first as a product of two safe primes and the accumulator function is a map $f : QR_N \times \mathcal{X} \longrightarrow QR_N$, where QR_N is the set of quadratic residues modulo N and $\mathcal{X} = \{e \in \mathbb{Z}_N^* \mid e \text{ is prime }\}$, defined as:

$$f(u, e) = u^e \mod N$$

A witness is a value corresponding to an element in the set that is requred to verify its membership. In case of the RSA accumulator construction, let the set of primes accumulated be $\mathcal{E} = \{e_1, e_2, ..., e_k\}$ and the product of the primes, $t = \prod_{i=1}^{k} e_i$ then the accumulator value is $v = u^t \mod N$. The witness for e_i is $v^{\overline{e_i}}$ where $\overline{e_i} = e_i^{-1} \mod \phi(N)$

Definition 1. *Accumulator update: The accumulator update function is $f(u, x)$, where the inputs (u, x)*

- *are defined on domain $\mathcal{U}'_f \times \mathcal{X}'_f$ and,*
- *use the intended input domain $\mathcal{U}_f \times \mathcal{X}_f$,*

where u and x are the accumulator value and the update value respectively.

For RSA accumulator:

- the function $f(u, x) = u^x \mod n$, where RSA modulus $n = pq$ such that $p = 2p' + 1$ and $q = 2q' + 1$ where p, p', q, q' are all primes. The length of integer n is k, the security parameter.
- $\mathcal{U}'_f = \mathbb{Z}_n^*$ and $\mathcal{U}_f = QR_n - \{1\}$.
- $\mathcal{X}'_k = [2, A^2 - 1]$ and $\mathcal{X}_{A,B} = \{e \in \text{primes} \mid e \neq p', q' \text{ and } e \in [A, B]\}$, where A, B can have polynomial dependence on k.
- To delete value $\tilde{x}$ from accumulator u, update it as $u' = u^{\tilde{x}^{-1} \mod \phi(n)} \mod n$, where $\phi(n) = (p-1)(q-1)$.
- To add multiple values simply use their product as x, similarly for deletion and $\tilde{x}$.

4 Proposed System

We consider a system for storing data in a secure storage server, which is then retrieved by other entities who have been granted permission to access the data. The information about access should be easily accessible to the storage access control service, and the consent granted should be undeniable by and under complete control of the owner of the data. In this conceived system, the roles of the various entities are bound by their respective terms of service. Technically it should be infeasible for any outsider to break the system without a direct involvement of the parties within this data sharing protocol. All parties should have the least amount of information, as necessary for the execution of their roles. The trusted mechanism to store the consent related data in the proposed system will be smart contracts on blockchain, replacing a centralized trusted-third party. The smart contract(s) will be globally accessible and data related to various consent mappings are stored in encrypted format. The encrypted data, which is to be used to carry out the cryptographic protocols, should only be decipherable by the concerned parties. All parties have been assumed to keep all cryptographic secrets protected and secure. We first introduce the parties involved in a typcical flow for sharing data. Then we detail the algorithms run by these parties and outline the system architecture involving the smart contracts and flow of data.

4.1 Parties and components

Data owner The entity who owns the data, is the subject of the data or controls the data at its origin is referred to as the data owner, or simply owner. The owner should have full control of the access and permission on data. This control is excercised provably through cryptographic mechanisms, while the owner's secret set is solely generated and stored within owner's own premises.

The owner is equipped with software on their end, like a user application, which is responsible for generating the smart contract transactions and storing cryptographic material. Losing the cryptographic material renders the owner unable to update the state of consent referenced earlier. Under such circumstances, the owner must look for an alternative reset mechanism outside of the protocols specified in this work.

Data Requester The entity that wants to access a previously stored data for various purposes is being referred to in this work as the Data requester, or simply requester. The legal terms of data usage have not been taken into account for the consent related data in this work. Nevertheless, the requester requests access from the owner and must prove the same to the verifier before being granted access to the data stream.

Data Service Provider The data service provider, or simply provider, is responsible for storing and transporting the data securely as required. In many cases it could also be the service responsible for generating the data itself on behalf of the owner. The service provider must look up consent related information from the smart contract and engage in a consent verification protocol with the requester, every time before sharing the data. The owner and the provider enter into an agreement to follow the protocols of the system after the owner registers with the provider.

Smart contracts The smart contracts are responsible for storage and update of the consent accumulator and related values for every data reference, for every owner. We propose the use of a common smart contract on Ethereum for this purpose. However, this simple data model could be adapted to any other smart contract or similar mechanisms.

Communication Channels A most of the communication in our proposed system happens over private communication channels. This is very different from most of the smart contract based application designs. The primary reason for this is that we use smart contracts as a means of distributed trusted source of information alone and most of the protocols are run off-chain. The data requester sets up a communcation channel with the owner for requesting for access and to establish parameters for consent sharing. Similarly, the provider and owner establish a channel for service registration. Further, the data transmission and consent verification between the requester and the provider happens over yet another channel formed between themselves. The interaction with the blockchain happens independently of these private channels. The user's application, the requester and the provider could all use some third party node service to interact with the blockchain, or could run their own nodes. It is assumed this interaction happens through secure medium.

4.2 Consent Sharing Mechanism

Suppose there is a file that has been created or generated by the data owner and is now to be stored and distributed by the provider. The provider relies on the consent information whose reference in the smart contract has been given by the owner. Firstly, the data owner must register themselves with the provider and create the appropriate entry in the smart contract as described in the following protocol

Protocol 1: User-Initialization The user (owner) will perform the following:

1. Choose safe primes p and q, and hence the RSA modulus $N = pq$.
2. Create a key pair to be used with the smart contracts, according to the native curve and algorithm for the blockchain platform. In case of ethereum, the user generates the pair (SK, PK) on *secp256k1*.
3. Choose a random value for the accumulator $v_0 \in_R QR_N$.
4. Create the initial product of primes using a few randomly chosen big primes, $E_0 = e_0 e_1 e_2 ... e_s$ where s is a random integer around 10.

5. Hash the file and use the hash as its unique identifier, $file\_uid$.
6. Call the smart contract method $initConsent(file\_uid, PK, N, E_0, v_0)$.

The requester then communicates with the data owner requesting permission to access the file. The owner updates the permission on the smart contract using the following protocol.

Protocol 2: Add Consent The owner performs the following:

1. Select two new primes $e_i, e_i' \in_R \mathbb{Z}_N^*$
2. Create a Pedersen commitment
$$P_i = \mathfrak{h}^{e_i} y_i^{r_i} \tag{1}$$
where y_i is the requestor's public key and $r_i \in_R \mathbb{Z}_N$
3. Generate the ElGamal encrypted tuple of the values in the commitment,
$$EG_i = (C_e, C_r, C_d, C_{r'}) = (e_i y_i^{r_i'}, r_i \mathfrak{g}^{e_i r_i'}, d y_i^{r_i'}, \mathfrak{g}^{r_i'}) \tag{2}$$
where $d = v^{e_i^{-1} \mod \Phi(N)}$; and $r_i' \in_R \mathbb{Z}_N$ is the randomness used here for ElGamal encryption.
4. Recalculate the updated product of primes $E' = E \cdot e_i \cdot e_i' \mod \Phi(N)$
5. Call smart contract method $addConsent(E', P_i, EG_i)$
6. Store The mapping $y_i : (e_i, e_i', r_i)$ internally.
7. Communicate the status of contract method call with the requester.

The Pedersen commitment is a hiding and binding here using a randomly selected group member $\mathfrak{h}$. In order to later open this commitment to the verifier, the requester needs to know both the prime e_i and the randomness r_i. For the sake of convenience (with some added transaction cost) we store this information in the ElGamal tuple, EG_i is encrypted under the requester's public key y_i.

Using this, e_i can be decrypted with the pair $(e_i y_i^{r_i'}, \mathfrak{g}^{r_i'})$. Similarly, d can be decrypted from $(d y_i^{r_i'}, \mathfrak{g}^{r_i'})$, and r_i can be obtained as
$$r_i = \frac{C_r}{C_{r'}^{e_i}} = \frac{r_i \mathfrak{g}^{e_i r_i'}}{(\mathfrak{g}^{r_i'})^{e_i}} \tag{3}$$

Protocol 3: Revoke Consent The owner performs the following to revoke a previously added consent:

1. Fetch the prime product for y_i, $\lambda = e_i \cdot e_i'$
2. Generate the updated accumulator value $v' = v^{\lambda^{-}1 \mod \Phi(N)} \mod N$
3. Generate the updated product of primes $E' = E.\lambda^{-1} \mod \Phi(N)$
4. Call the contract method $updateConsent(E', v')$

Protocol 4: Prove consent This protocol is carried out between the requester and provider over a secure channel, off the chain. The requester proves in zero-knowledge the following:

– Requester knows a prime e_i accumulated in the accumulator of the $file_uid$ and the corresponding witness.
– Requester knows a secret x corresponding to the public key y in the Pedersen commitment P_i of the prime.

The requester performs the following:

1. Compute the prime e_i for the $file_uid$ from the corresponding entry in the smart contract, $EG_i = (C_e, C_r, C_d, C_{r'})$ as follows:
$$e_i = \frac{C_e}{C_{r'}^x};\ d = \frac{C_d}{C_{r'}^x};\ r_i = \frac{C_r}{C_r^{ex}} \tag{4}$$
2. Compute the value of witness for e_i,
$$\begin{aligned} w_i &= d^E \mod N \\ &= (v_0^{e_i^{-1} \mod \Phi(N)})^E \mod N \\ &= v^{e^{-1} \mod \Phi(N)} \mod N \end{aligned}$$
3. Create three auxiliary commitments for e_i, w_i and the corresponding randomness as:
$$P_e = g^{e_i} h^{r_1};\ P_w = w_i h^{r_2};\ P_r = g^{r_2} h^{r_3} \tag{5}$$
where $r_1, r_2 \in_R \mathbb{Z}_N$.
4. Share with the provider $P_e, P_w P_r$ and engage in the following zero-knowledge protocol
$$\begin{aligned} PK\Big\{&(\mathfrak{g}, \mathfrak{h}, N, g, h, v, P_i, P_e, P_u, P_r; e_i, w_i, r_i, r_1, r_2) : P_i = \mathfrak{h}^e y_i^{r_i} \wedge \\ &\mathfrak{h} = \left(\frac{P}{\mathfrak{h}}\right)^{\gamma} y_i^{r_i \gamma} \wedge \mathfrak{h} = (\mathfrak{h} P_i)^{\sigma} y_i^{r_i \sigma} \wedge v = P_u^e \left(\frac{1}{h}\right)_2^r \wedge \\ &1 = P_r^e \left(\frac{1}{h}\right)^{r_3 e} \left(\frac{1}{g}\right)^{r_2 e} \wedge e \in [2, A^2 - 1]\Big\} \end{aligned} \tag{6}$$
where $\gamma = (e-1)^{-1} \mod \mathbf{q}$ and, $\sigma = (e+1)^{-1} \mod \mathbf{q}$

This ZKP protocol is carried out as follows.
The prover and the verifier carry out a set of Σ- protocols involving a common challenge by the verifier. First, the prover choose the following random integers for the set of Σ - protocols.

$$\begin{aligned} r_\alpha &\in_R \left[-B2^{k'+k"}, B2^{k'+k"}\right] \\ r_\gamma, r_\phi, r_\psi, r_\sigma, r_\xi &\in_R \mathbb{Z}_q \\ r_\epsilon, r_\eta, r_\zeta &\in_R \left[-\lfloor n/4 \rfloor 2^{k'+k"}, \lfloor n/4 \rfloor 2^{k'+k"}\right] \textit{ and} \\ r_\beta, r_\delta &\in_R \left[-\lfloor n/4 \rfloor q 2^{k'+k"}, \lfloor n/4 \rfloor q 2^{k'+k"}\right] \end{aligned}$$

Next, the prover generates and sends the following commitments to the above integers as follows:

$$p_1 = \mathfrak{h}^{r_\alpha} y^{r_\phi} \qquad p_2 = \left(\frac{P_i}{\mathfrak{h}}\right)^{r_\gamma} y^{r_\psi} \qquad p_3 = (\mathfrak{h} P_i)^{r_\sigma} y^{r_\xi}$$

$$t_1 = h^{r_\zeta} g^{r_\epsilon} \qquad t_2 = h^{r_\alpha} g^{r_\eta} \qquad t_3 = P_w^{r_\alpha} \left(\frac{1}{h}\right)^{r_\beta}$$

$$t_4 = P_r^{r_\alpha} \left(\frac{1}{h}\right)^{r_\delta} \left(\frac{1}{g}\right)^{r_\beta}$$

The prover computes and sends to the verifier the following challenge response:

$$\begin{array}{lll}
s_\alpha := r_\alpha - ce_i & s_\beta := r_\beta - cr_1 & s_\phi := r_\phi - cr_i \mod q \\
s'_\beta := r_\beta - cr_2 e & s_\epsilon := r_\epsilon - cr_2 & s_\gamma := r_\gamma - c\gamma \mod q \\
s_\zeta := r_\zeta - cr_3 & s_\delta := r_\delta - cr_3 e & \\
s_\psi := r_\psi + cr\gamma \mod q & s_\sigma := r_\sigma - c\sigma \mod q & s_\xi = r_\xi + cr\sigma \mod q
\end{array}$$

The verifier checks if the following hold and accepts if they do:

$$\begin{array}{lll}
p_1 \overset{?}{=} P_i^e \mathfrak{h}^{s_\alpha} y_i^{s_\psi} & p_2 \overset{?}{=} \mathfrak{h}^c \left(\frac{P_i}{\mathfrak{h}}\right)^{s_\gamma} y_i^{s_\psi} & p_3 \overset{?}{=} \mathfrak{h}^c (\mathfrak{h}P)^{s_\sigma} y^{s_\xi} \\
t_1 \overset{?}{=} P_r^c h^{s_\zeta} g^{s_\epsilon} & t_2 \overset{?}{=} P_e^c h^{s_\alpha} g^{s_\eta} & t_3 \overset{?}{=} v^c P_u^{s_\alpha} \left(\frac{1}{h}\right)^{s}_{\beta} \\
t_4 \overset{?}{=} P_r^{s_\alpha} \left(\frac{1}{h}\right)^{s_\delta} \left(\frac{1}{g}\right)^{s_\beta} & s_\alpha \overset{?}{\in} [-B2^{k'+k''+2},\, B2^{k'+k''+2}]. &
\end{array}$$

4.3 Smart Contract Outline - ConsentManager

We present here an outline of the smart contract meant for the Ethereum EVM. This is not a comprehensive code for the actual usable smart contract for the protocol, but it gives a workable outline.

Structs/Types

1. **ElGamalTuple**:
   - *Ce*, binary representation of encryption of the prime e_i (bytes)
   - *Cr*, binary representation of encryption of thee randomness r_i (bytes)
   - *Cd*, binary representation of encryption of d e_i (bytes)
   - *Crp*, binary representation of blinded base g e_i (bytes)

2. **PedersenCommitment**:
   - value, the commitment value (bytes)

3. **ConsentData**:
   - *Owner*, Address of the owner (address - 32 bit Ethereum address)
   - *N*, the Modulus(bytes - 2048 bit binary representation in)
   - *E*, Value of product of primes, $E \mod \phi(N)$ (bytes)
   - *V0*, Initial value of the accumulator (bytes)
   - *EGi*, Array of ElGamal tuples (ElGamalTuple[ ])
   - *Pi*, Array of Pedersen Commitments (PedersenCommitment[])

State Variables

1. ***ConsentDataMap***, mapping (file_uid (bytes32) → ConsentData) - Map containing all of the consent related information for a given file_uid (i.e. SHA-256 hash of the file). This is meant to keep records of all registered files of all users in a given context/application globally.

2. ***registeredFiles*** ($bytes32[.]$) - An array containing all registered file UIDs.

Functions

1. *initConsent (file_uid, N, E0, V0)* - Function to initialize the consent record for a certain file, identified by the corresponding hash, $file\_uid$, in the following steps:
   - Check $ConsentDataMap[file\_uid]$ is null
   - Push $file\_uid$ to $registeredFiles$
   - Set Owner = msg.sender
   - Initialize $(N, E0, V0)$ for $ConsentDataMap[file\_uid]$ to their respective values in the parameter

2. $addConsent(file\_uid, EGi, Pi, E')$ - Function to add a new consent for the corresponding file. Execution as follows:
   - Check $(msg.sender == Owner)$ for $ConsentDataMap[file\_uid]$
   - Push EGi, Pi to the respective $EGi[.], Pi[.]$ arrays of $ConsentDataMap[file\_uid]$
   - Set $E \leftarrow E'$, updating the product of primes representation.

3. $updateConsent(E')$ - Function to update the accumulator value. This could be utilised to revoke consent stealthily.
   - Set $E \leftarrow E'$

5 Security Analysis

5.1 Security Requirements

We define the following security requirements:

Correctness: When the honest use shares consent to sharing a file with a requestor, this consent should be verifiable. This requries the following aspects to be in order:

- Identification of the file in question
- Identification of user's control on the consent
- Identification of the requestor for whom the consent is intended
- Access to the data of legitimate transaction containing the information to verify the proof of the consent
- Verification of consent using the shared information

Soundness: An adversary on the network cannot prove consent to a file, through its identifier, without a legitimate consent transaction on the blockchain. This requirement includes the following condition on the the adversary:

- Can't show incorrect linkage between hash and file permission
- Can't generate a proof to a consent that does not exist
- Can't produce an acceptable consent proof after it has been revoked

Zero Knowledge: We define the following conditions for its zero-knowledge property

- The protocol does not reveal any more data to any parties involved in it, than they need to know to verify a requested linked file's consent state

- No data stored publicly can enable an adversary to carry out the protocol to prove consent, without being the subject of a shared consent
- The information passed on by the owner to the requestor does not reveal any other information than what is needed by the requestor to prove the consent against that file
- The information sent to the verifier by the prover does not reveal any more information than what is strictly necessary for the verifier to check the membership proof proof

5.2 Correctness

File identification: In our protocol every file is uniquely identified with a single SHA-256 hash. It would be expected of honest file providers to hash every file before identifying or sharing them to check for hash mismatch. It is a part of the protocol to disallow duplicate entries, also covering the extremely rare occurrence of a hash collision.

User control: User control is exercised by controlling the data on the consent contract (s). Such control is readily verified within the ethereum blockchain, through enforcement of address checks (as addresses correspond to public keys). For every new file entry, either a new keypair is generated or the transaction sender is verified to be the corresponding owner of the file. Thus, unauthorized modification is only as feasible as compromising the underlying (ethereum) blockchain network.

Identifying the requestor: Each consent vector also includes the public key of he requestor which they want to use to carry out this protocol. This identification is established with the well known Schnorr Identification protocol and we will assume it's security properties there. The protocol requires that the authentication of the requestor is carried out in addition to zero-knowledge membership proof. Including the key with the consent vector is beneficial in two ways:

1. The requestor could not just pass on the membership secret to other parties without also sharing their secret to the corresponding public key. In most cases the stake on the public key would be high enough for the requestor to be not able to perform this.
2. Known parties can be directly added to and removed from the consent vector without interacting with them. For many use cases the service providers requesting data are going to be well-known parties who could publish their public keys in a website or a public API.

Transaction data access: Access to the blockchain data is unrestricted, since it is one of the core features of a public blockchain like Ethereum. It is thus assumed that in almost all cases access to the core consent related information is correctly available. The only exceptions being cases where very targeted network spoofing or blocking is being executed.

Verification of consent The correctness of the verification of consent follows from

Theorem 1. *The ZKP protocol in equation 7 is correct for an honest prover and honest verifier* $\iff$ *The prime* e_i *is indeed a part of the accumulator and* w_i *is a witness to it.*

This follows if P_e, P_w, P_r are correct commitments to e_i, w_i and the corresponding randomness, such that $w_i^{e_i} = v_i \bmod N$

$\iff$ The auxiliary commitments $p_1, p_2, p_3, t_1, t_3, t_4$ are correctly formed

$\iff$ The set of equations (10) for the validity checks are correct, which is indeed the case algebraically.

5.3 Soundness

Incorrect Linkage As discussed earlier, the file hashes cannot be duplicated on-chain. However it should be included in the implementation of the file provider to check if duplicate identifiers are linked to permissions of the file. Basically, the file provider is supposed to check this linkage on chain, after doing identity verification on its own.

Proof to a non-existent consent

Theorem 2. *The ZKP protocol in equation 7 is sound for an honest verifier.* $\iff$ *The verifier rejects the protocol with a high probability, when $e_i \notin Acc$, assuming the prover is malicious and has access to all the public information in the consent contract.*

Proof sketch Let's assume he's able to find such e and w, such that, $P_e = g^e h^{r_1}$; $P_w = wh^{r_2}$ and $P_r = g^{r_2} h^{r_3}$ that fulfils the protocol run.

TODO following from the proofs given by CL01, an adversary can't come up with such prime and witness.

Proof to a revoked consent. We reuse the notion of security by CL01, also given in section 3, and the formal definition for the secure accumulator. Since we have practically re-used the ZKP protocol, the security argument directly follows from the CL01. Under the strong RSA assumption, the construction provides a secure dynamic accumulator.

The proof presented by CL01 shows that an Adversary $\mathcal{A}$ that on input n and $u \in_R QR_n$, outputs m primes $x_1, ..., x_m \in \chi_{A,B}$ and $u' \in \mathbb{Z}_n^*$, $x' \in \chi'_{A,B}$ such that $(u')^{x'} = u^{\prod x_i}$ can be used to break the RSA assumptions.

However, in order to creae a forgery such that $e_i.e'_i$ is used used instead of e_i alone, $\frac{E'}{E} = e_i.e'_i \mod \phi(N)$ needs to be performed, which isn't possible as $\phi(N)$ is not known to the adversary.

5.4 Zero-Knowledge

Our adversary will have access to additional information, thanks to the transparency of blockchain transactions. Let's look at all tx related data to investigate any further advantage to the adversary in this respect. The most of the data is stored in the following state variables in the smart contract:

- EG_i is an encrypted tuple that reuses the randomness r'_i and following from Kurosawa02, it's secure. The individual values in the tuples are blinded by multiplications and no meaningful information can be extracted from them, if the factorization of n is not known.
- P_i is both hiding and blinding, it also doesn't hold any direct factors from the EG_i tuple so EG_i and P_i can't produce any advantage, even if used in conjunction
- E': The updated product of primes. The modular multiplication of the second random prime makes it difficult to extract the prime e_i due to difficult of the factorization problem. Thus, gettting the prime isn't possible for the adversary.

Privacy: Privacy in transaction is achieved by hiding the original file on which consent is being given. For this we use a file identifier, which would be the hash of the file, on all public records. Also, the file would be encrypted by file-owner before being sent to be stored with the provider and generation of identifier. *(Once the receiver has the encrypted file they use a re-encryption key to decrypt it).*

Anonymity: Anonymity is achieved by hiding the receiver of the consent transaction. For this, we first use a Pederson commitment which stores receiver's public key as well as their consent value. Then, to later access the file, the receiver shows a Zero-Knowledge proof involving the commitment to the provider, which verifies that the receiver has the consent to access the file. Anonymity is with respect to receiver but not sender since the information about sender (file-owner) is stored publicly along with the file identifier. Also, unlike other users, provider has the knowledge of receiver's identity which it needs to provide them with file access.

As discussed, each file is publicly identified with its identifier and its owner's public key is stored with it. Then, to keep track of users who have consent to the file we use an RSA accumulator for each file. This dictates consent values would be prime numbers, which would be accumulated. To a grant a consent sender would update the accumulator adding the new consent value and to revoke a consent sender would update the accumulator by deleting that consent value. Furthermore, consent value needs to be bound with the receiver, so that even if a receiver shares their prime with another user, this user cannot query the file. This binding is achieved by using receiver's public key to form a Pederson commitment of the prime.

6 Future Work

TODO

APPENDIX 1

For second ZKP, we make use of the ZKP protocol provided by Jan Camenisch and Anna Lysyanskaya in reference [?] which proves that a committed value is accumulated. Prover, i.e. receiver, proves in ZK to verifier, i.e. provider, that the prime e committed in $P = \mathfrak{h}^e y^r$ is in accumulator v through three auxiliary commitments $C_e = g^e h^{r_1}$, $C_u = uh^{r_2}$ and $C_r = g^{r_2} h^{r_3}$, where g and h are generators of QR_n such that prover does not know $log_g(h)$. We also require that $\mathcal{X}_{A,B}$ and $\mathbb{G}_{\mathbf{q}}$, along with k', bit length of challenge, and k'' *(what is this?)* satisfy $B2^{k'+k''+2} < A^2 - 1 < \mathbf{q}/2$. For PK following relations are used

$$P = \mathfrak{h}^e y^r, \qquad \mathfrak{h} = \left(\frac{P}{\mathfrak{h}}\right)^{\gamma} y^{r\gamma}, \qquad \mathfrak{h} = (\mathfrak{h}P)^{\sigma} y^{r\sigma}, \tag{7}$$

$$C_r = g^{r_2} h^{r_3}, \qquad C_e = g^e h^{r_1}, \tag{8}$$

$$v = C_u^e \left(\frac{1}{h}\right)^{r_2}, \qquad 1 = C_r^e \left(\frac{1}{h}\right)^{r_3 e} \left(\frac{1}{g}\right)^{r_2 e}, \tag{9}$$

$$e \in [-B2^{k'+k''+2}, B2^{k'+k''+2}], \tag{10}$$

where $\gamma = (e-1)^{-1} \bmod \mathbf{q}$ and $\sigma = (e+1)^{-1} \bmod \mathbf{q}$.
ZKP protocol: Common knowledge: P, v, $\mathfrak{h}$, y, n, g and h.
Prover's additional knowledge: e, u and r.
Prover chooses randomnesses

$$\mathsf{r}_\alpha \in_R (-B2^{k'+k''}, \ldots, B2^{k'+k''}) ,$$
$$\mathsf{r}_\gamma, \mathsf{r}_\varphi, \mathsf{r}_\psi, \mathsf{r}_\sigma, \mathsf{r}_\xi \in_R \mathbb{Z}_\mathfrak{q} ,$$
$$\mathsf{r}_\varepsilon, \mathsf{r}_\eta, \mathsf{r}_\zeta \in_R (-\lfloor n/4 \rfloor 2^{k'+k''}, \ldots, \lfloor n/4 \rfloor 2^{k'+k''}) \text{ , and}$$
$$\mathsf{r}_\beta, \mathsf{r}_\delta \in_R (-\lfloor n/4 \rfloor \mathfrak{q} 2^{k'+k''}, \ldots, \lfloor n/4 \rfloor \mathfrak{q} 2^{k'+k''}) ,$$

and commits them as

$$\mathfrak{t}_1 = \mathfrak{h}^{r_\alpha} y^{r_\varphi} \qquad \mathfrak{t}_2 = \left(\frac{P}{\mathfrak{h}}\right)^{r_\gamma} y^{r_\psi} \qquad \mathfrak{t}_3 = (\mathfrak{h}P)^{r_\sigma} y^{r_\xi}$$
$$t_1 = h^{r_\epsilon} g^{r_\zeta} \qquad t_2 = h^{r_\alpha} g^{r_\eta}$$
$$t_3 = C_u^{r_\alpha} \left(\frac{1}{h}\right)^{r_\beta} \qquad t_4 = C_r^{r_\alpha} \left(\frac{1}{h}\right)^{r_\delta} \left(\frac{1}{g}\right)^{r_\beta} .$$

Then verifier sends the challenge $c \in \{0,1\}^{k'}$ to prover.
Prover computes and sends

$$\mathsf{s}_\alpha := \mathsf{r}_\alpha - \mathsf{c}\mathsf{e} , \qquad \mathsf{s}_\eta := \mathsf{r}_\beta - \mathsf{c}\mathsf{r}_1 , \qquad \mathsf{s}_\varphi := \mathsf{r}_\varphi - \mathsf{c}\mathsf{r} \bmod \mathfrak{q} ,$$
$$\mathsf{s}_\beta := \mathsf{r}_\beta - \mathsf{c}\mathsf{r}_2\mathsf{e} , \qquad \mathsf{s}_\varepsilon := \mathsf{r}_\varepsilon - \mathsf{c}\mathsf{r}_2 , \qquad \mathsf{s}_\gamma := \mathsf{r}_\gamma - \mathsf{c}(\mathsf{e}-1)^{-1} \bmod \mathfrak{q} ,$$
$$\mathsf{s}_\zeta := \mathsf{r}_\zeta - \mathsf{c}\mathsf{r}_3 , \qquad \mathsf{s}_\delta := \mathsf{r}_\delta - \mathsf{c}\mathsf{r}_3\mathsf{e} , \qquad \mathsf{s}_\psi := \mathsf{r}_\psi - \mathsf{c}\mathsf{r}(\mathsf{e}-1)^{-1} \bmod \mathfrak{q} ,$$
$$\mathsf{s}_\sigma := \mathsf{r}_\sigma - \mathsf{c}(\mathsf{e}+1)^{-1} \bmod \mathfrak{q} \text{ , and } \quad \mathsf{s}_\xi := \mathsf{r}_\xi - \mathsf{c}\mathsf{r}(\mathsf{e}+1)^{-1} \bmod \mathfrak{q}$$

Finally verifier accepts if

$$\mathfrak{t}_1 \overset{?}{=} P^c \mathfrak{h}^{s_\alpha} y^{s_\varphi} \qquad \mathfrak{t}_2 \overset{?}{=} \mathfrak{h}^c \left(\frac{P}{\mathfrak{h}}\right)^{s_\gamma} y^{s_\psi} \qquad \mathfrak{t}_3 \overset{?}{=} \mathfrak{h}^c (\mathfrak{h}P)^{s_\sigma} y^{s_\xi}$$
$$t_1 \overset{?}{=} C_r^c h^{s_\epsilon} g^{s_\zeta} \qquad t_2 \overset{?}{=} C_e^c h^{s_\alpha} g^{s_\eta}$$
$$t_3 \overset{?}{=} v^c C_u^{s_\alpha} \left(\frac{1}{h}\right)^{s_\beta} \qquad t_4 \overset{?}{=} C_r^{s_\alpha} \left(\frac{1}{h}\right)^{s_\delta} \left(\frac{1}{g}\right)^{s_\beta}$$
$$s_\alpha \overset{?}{\in} [-B2^{k'+k''+2}, B2^{k'+k''+2}].$$

Receivers query the file by using a zero-knowledge proof to show knowledge of their prime e in the accumulator value on the SC. Since the accumulator could have been updated since when this user received the consent, to generate this ZKP receivers need to calculate updated witness to their prime i.e. generate $u : u^e = v \bmod n$. This calculation is done efficiently through a new scheme we propose by using a product value, which is product of primes currently in accumulator in mod $\phi(n)$. Thus, for each file we have the following information publicly stored:

1. its file identifier id, which would be the hash of the encrypted file,
2. its owner's public key y_o
3. an accumulator $v \in QR_n$, which accumulates primes e_i,
4. a product of primes currently in accumulator,

$$E = \prod_{i:e_i \text{ in } v} e_i \bmod \phi(n), \tag{11}$$

which receivers use to generate ZK proof of their prime in accumulator, and,

5. a list of Pederson commitments, $(P_1, P_2, \ldots)$, committing the primes along with the receiver, as $P_i = \mathfrak{h}^{e_i} y_i^{r_i}$ where $y_i = \mathfrak{g}^{x_i}$ is receiver's public key, x_i their secret key and r_i is randomness.

All this information is maintained by the smart contract(SC). Any changes to these occur through consent grant/revoke transactions which file-owner would publish.

The zero-knowledge proof generated by receiver shows the provider, in zero-knowledge, that the receiver knows the witness to a prime in accumulator. For the receivers to generate zero-knowledge proof about their prime in the accumulator, they need to know

– their prime, e_i,
– randomness in P_i, i.e. r_i and,
– the value $d_i = v_0^{e_i^{-1} \bmod \phi(n)}$.

These three are also put on the SC as ElGamal encryption for the receiver. Also, for facilitating revocation the prime and the receiver's public key is encrypted and stored for the sender. All this is encrypted as an ElGamal tuple $EG_i = (e_i y^{r'_i}, r\mathfrak{g}^{e_i r'_i}, d_i y^{r'_i}, y y_o^{r'}, \lambda y_o^{r'}, \mathfrak{g}^{r'_i})$ where $r'_i \in Z^*_{\mathbf{q}}$. So instead of just list of Pederson commitments (in point 5), we have a list of vectors, where each vector is (P_i, EG_i).

References