How to Recover a Cryptographic Secret From the Cloud

David Adei, Chris Orsini, Alessandra Scafuro*, and Tanner Verber†

North Carolina State University

May 21, 2024

Abstract

Clouds have replaced most local backup systems as they offer strong availability and reliability guarantees. Clouds, however, are not (and should not be) used as backup for cryptographic secrets. Cryptographic secrets might control financial assets (e.g., crypto wallets), hence, storing such secrets on the cloud corresponds to sharing ownership of the financial assets with the cloud, and makes the cloud a more attractive target for insider attacks.

Can we have the best of the two worlds, where a user, Alice, can conveniently store a copy of her cryptographic secrets on the cloud and she is the only one who can recover them (without trusting any entity)? Can she do so even when she loses her devices and forgets *all credentials*, while at the same time retaining full ownership of her secrets?

In this paper, we provide a cloud-based secret-recovery mechanism where confidentiality is always guaranteed when Alice has not lost her credentials, even in the presence of a malicious cloud. If Alice loses all her credentials, she can still recover her secrets (in most circumstances). This is in contrast with all previous work that relies on the assumption that Alice remembers some authentication secret. We prove our system secure in the Universally Composable framework. Further, we implement our protocols and evaluate their performance.

1 Introduction

Some digital assets, such as cryptocurrencies [44] or encrypted databases [49], require the knowledge of a cryptographic secret (e.g., a signing key, a decryption key) to be accessed and used by its owner. The security of such digital assets hinges on the cryptographic secret being known *only* by its owner, and, if the owner loses the secret, she loses access to this asset[1].

To avoid this loss, could a user, Alice, store a backup copy of her cryptographic secrets on a cloud, while retaining full ownership of her secrets? It seems that the answer to this question should be a definitive no. Firstly, by sending the cryptographic secrets s to the cloud, the user is effectively giving the cloud the ability to access the asset associated with s, defeating the purpose of using a cryptographically secured digital asset to begin with. However, even if we trust the cloud to not illegitimately access users' assets, another potential point of failure is the authentication method that Alice uses to connect to the cloud. The security of the cryptographic systems for which Alice is using her cryptographic secrets (e.g., her wallet) is downgraded to the security of the authentication system Alice uses with the cloud. Finally, if Alice uses a strong two-factor authentication (2FA) involving her physical devices as second factors, there is still a chance that Alice will lose access if she loses all her physical devices.

In this paper, we revisit the concept of break-glass encryption, introduced by Scafuro [50], to provide an efficient and concrete mechanism that allows Alice to use the convenience of a cloud to store a cryptographic secret, in such a way that (1) the cloud does not learn the secret (2) Alice can recover her secret even when she loses all credentials and devices, while no one else can recover the secret.

*Alessandra Scafuro and Tanner Verber are supported by a research grant from Horizen Labs.

†Author contact tverber@ncsu.edu.

[1]This is a severe problem in cryptocurrencies where losing access to the signing key results in losing the ability to create transactions and hence use the money.

Such a mechanism seems to have two contradicting security requirements. On the one hand, we must provide provable cryptographic guarantees that no one, not even the cloud, can learn the cryptographic secret that Alice is storing. On the other hand, we must guarantee that if Alice loses all her devices and secrets – which puts Alice in a position of being like anyone else – she can connect to the cloud and recover her secret. While seemingly irreconcilable, we show that this can be achieved.

To better frame the ideas behind our scheme, we present the overarching approach we take in this paper to achieve the security requirements outlined above. The first requirement asks that Alice stores her secret s on the cloud, in such a way that the cloud will never learn the secret. This would be easily achieved by having Alice store an encryption of s under some key retK that Alice knows which we call the "retrieval key". As long as Alice remembers retK, she will be able to retrieve and decrypt her cryptographic secret. The second requirement is that if Alice loses *everything*, she can recover and decrypt s. This is impossible unless Alice gave retK to someone else, but we want that no one can access this secret besides Alice. Note that this requirement rules out any solution based on escrowing the key to trusted parties [43, 52, 15, 29, 16]. In particular, we want that no one can ever access Alice's secret without her detecting it (which would happen if the key is escrowed to others [2]).

To overcome this challenge we use the aid of a trusted execution environment (TEE) [41, 47]. TEE is a technology that allows a client to create her own private space on an untrusted cloud machine (also called host). The assumption is that every computation and piece of data stored in the enclave is opaque to the host.[3] Hence, we require that the cloud is equipped with a TEE and Alice can create her own secure enclave to privately store retK. The TEE can be seen as the trusted party that knows about Alice's key and can use it in case of emergency to decrypt Alice's ciphertext, recover her secret s, and *re-encrypt* it under a new key, say pk, that Alice freshly created. This is done obliviously to the cloud who just observes the ciphertext c as input to the enclave, and then the fresh ciphertext c' in output, which is a re-encryption of c under the new pk.

However, the most challenging question remains unanswered. How does the TEE know that pk is a key chosen by Alice, and not by the cloud or another party who is pretending to be Alice?

This brings us to the second, and most challenging, tool that we build in this paper: a mechanism to obtain credential-less cryptographically secured permission. This mechanism, inspired by a similar definition presented, but not realized, by Scafuro [50] allows *only* Alice to legitimately authenticate, even when she loses all cryptographic secrets. Permission should be publicly verifiable and is given to the TEE to authenticate a recovery request from Alice. This again seems improbable, since any mechanism that allows Alice to obtain a certificate without any particular secret, should also allow any party to do the same. However, Scafuro observed that there is a difference between Alice and the others. Alice would ask for a certificate only in the rare case that she lost her credentials. Most of the time, Alice does have such credentials and can use them to authenticate. In contrast, other parties are never able to authenticate. This asymmetry can be leveraged to create a permission mechanism that guarantees that Alice can use her keys to stop any malicious requests for permission and that, in the case where Alice lost everything, her attempt to create permission cannot be (cryptographically) stopped by anyone.

Our solution requires that Alice uses her credentials (as long as she has them) to actively participate in stopping illegitimate attempts at recovery, as well as initiate a recovery request as soon as she realizes that she lost them. The latter is necessary as without her credentials, Alice does not have the power to stop any future illegitimate requests.

There is a possibility of a non-cryptographic attack that blocks Alice, whereby an adversary *guesses* that Alice will make a recovery request at a specific time, and will initiate a recovery request in parallel. As we discuss later, this attack would prevent Alice from recovering, but will never compromise the security of Alice's secrets since two competing requests result in aborting permission generation. Likewise, if Alice fails to reject a malicious request, she will lose her confidentiality. Therefore, Alice needs to ensure that she watches for any requests made on her behalf.[4]

[2] We note that even using trusted escrows does not completely solve the problem. Indeed, such parties must make sure that the *real* Alice is asking to retrieve the secret. This is the problem of credentials-less authentication that we discuss later.

[3] In practice, TEEs are still not bulletproof since hardware side-channel attacks exist. Nevertheless, this technology is still developing and it is used in several works [28, 36, 57, 34, 25]

[4] As we will see, this watching can be outsourced.

Our Contributions. We build upon the ideas presented in [50] to provide a full protocol that allows clients to securely recover cryptographic secrets stored on a cloud without requiring credentials while guaranteeing that no one else, not even the cloud, learns the secret. Our protocol requires the construction of two tools: a tool to cryptographically authenticate a client when she loses all her credentials (*Credential-less* Permission Mechanism) and a tool to allow the cloud to obliviously recover a secret for the client, without learning the secret. We define the two tools in a modular way and we realize them independently. We provide the following contributions:

1. **First Realization of Credential-less Permission Mechanism $\mathcal{G}_{Perm}$ Using Blockchains.** We provide the first concrete protocol that securely realizes the $\mathcal{G}_{Perm}$ functionality first defined by Scafuro [50]. In proving UC security of such a protocol many subtleties arise, highlighting issues with previous definitions and proposed instantiations. We therefore revisited and improved the definition of $\mathcal{G}_{Perm}$. Our realization leverages blockchain technology. To highlight the importance of this contribution, note that any protocol for recovering data requires authentication, else the data can be stolen. Since our protocol, to our knowledge, is the first to realize credential-less permission, it is the first to allow for truly credential-less data recovery.
2. **UC-Definition of Credential-less Secret Recovery $\mathcal{F}_{SecRec}$.** We provide a formal definition of Credential-less Secret-Recovery, in the UC-framework, by introducing the $\mathcal{F}_{SecRec}$ ideal functionality.
3. **UC-Protocol Realizing $\mathcal{F}_{SecRec}$ in the $\mathcal{G}_{Perm}$ Hybrid World Using TEE.** We instantiate $\mathcal{F}_{SecRec}$ with a very efficient protocol that leverages the security of TEE, and requires minimal overhead for the cloud. We provide a formal UC-security proof of our protocol, and hence we use the UC-formalization of TEE [47], $\mathcal{G}_{att}$ (Fig. 8). Our protocol is modular and uses $\mathcal{G}_{Perm}$ as a module that can be instantiated with other realizations besides the one we provide in this paper.
4. **Software Implementation and Evaluation of Permission Mechanism and Secret Recovery.** We provide a concrete software implementation of our system. We instantiate the TEE with AWS Nitro Enclave [12] and a simulation of Hyperledger Fabric [1]. More details about the implementation can be found in Sec. 7.

2 Our Techniques

Here we give a high-level description of the techniques used to provide our contributions.

2.1 First Realization of Credential-less Permission Mechanism $\mathcal{G}_{Perm}$ Using Blockchains

We revisit the definition of credential-less permission provided by Scafuro [50] in our own ideal functionality $\mathcal{G}_{Perm}$ (Fig. 2) making technical changes to improve modularity. Among these changes, our formulation defines the role of clients and servers, and their connection, and better defines the role of external parties in requesting permission. The verification of the permission is defined as an external predicate. These changes make $\mathcal{G}_{Perm}$ usable as a building block in any application that requires authenticated credential-less permission. Our main contribution on this front, however, is in our *UC-realization of* $\mathcal{G}_{Perm}$. While Scafuro provided a definition and a discussion of potential realizations, this work is the first to concretely realize $\mathcal{G}_{Perm}$ with our protocol Π_{Perm} and prove security.

At a high level, our blockchain-based realization is as follows. To register to the permission system, Alice posts a transaction indicating the identifier she wants to use perm-info, a public key $\mathsf{vk_C}$ that she wants to use to block/accept permission requests, and the server(s) that are allowed to use her permissions. Only servers that have published their verification key to the blockchain can be chosen by Alice. In this registration transaction, Alice also establishes timing parameters related to the creation of the permission: t_{open}, t_{wait}, and t_{chal}, which will become clear later in the description.

To have a transaction posted on the blockchain, Alice may need to create a blockchain account (e.g., wallet). However, in Π_{Perm}, Alice need not remember the secret key associated with this wallet and can create accounts on the fly. Π_{Perm} uses the blockchain as a public bulletin board and does not use any account information inherent to the chain. We do not consider the expenses of posting on the blockchain,

but they could be leveraged to discourage malicious parties from posting illegitimate requests and to reward Alice's attempt to stop them (see *Competing Requests* paragraph Sec. 2).

Now, say Alice loses the keys she used to access the server S. She must now create permission for S. First, she posts a transaction tx on the blockchain containing $\mathsf{perm\text{-}info}$, the public key of her server S, and a request field that we call req, which contains additional information for S.[5].

Once Alice's transaction appears on the blockchain, she must wait for t_{wait} blocks to pass, then there are two cases. If Alice does remember the signing key $\mathsf{sk_C}$ associated with $\mathsf{vk_C}$ that was registered in $\mathsf{perm\text{-}info}$, Alice can endorse by signing this transaction.[6] If Alice does not remember any key, then Alice waits for t_{chal} blocks to be added to the ledger after the waiting period of t_{wait} blocks has passed. These t_{chal} blocks represent valid permission that we call a *silent proof*. Note that constructing this sequence required no secrets from Alice.

Since this required no secret, a malicious party, Jeff, could attempt to create permission by following the procedure above. Jeff can create a transaction tx^* containing $\mathsf{perm\text{-}info}$ and attempt to construct a silent proof. The key observation here is that, if Alice did not lose her secrets, then she still has $\mathsf{sk_C}$ and can post a signature to *deny* the request and stop Jeff from obtaining a silent proof. In this case, when Alice sees the transaction tx^*, she will wait the t_{wait} blocks and follow up with a transaction where she denies tx^* with a signature that verifies under $\mathsf{vk_C}$. The ability to deny is why we consider silence to be proof of accepting permission.

Note that this requires that Alice monitors the blockchain and is on the lookout for illegitimate permission requests. While this might seem taxing for Alice, monitoring the ledger could be offloaded to a server who notifies Alice in the case of a request. Further, the following observations show that this work can move from taxing to rewarding. The parameter t_{wait} plays an important role in the frequency of the monitoring activity. Alice could choose t_{wait} to be long enough (e.g., one week) so that she does not have to monitor the blockchain constantly, but has a monitoring procedure going off once a week. Likewise, t_{chal} should be set to be long enough that Alice can reliably post a denial within that period. Second, while we do not explicitly implement this in the paper, every permission request can have a required request fee that is automatically paid to the wallet associated with $\mathsf{perm\text{-}info}$ when denied, as is done in KELP [20]. In this way, every denial yields Alice a reward.

The last, most challenging, case is where Alice loses her keys and posts a transaction tx requesting permission for $\mathsf{perm\text{-}info}$, S, req, and at the same time Jeff posts a transaction tx^* for the same $\mathsf{perm\text{-}info}$, S, req'. Since Alice lost her keys, she cannot stop tx^*. This attack is similar to "front-running attacks" that plague blockchain applications.

Luckily the front-running problem can be greatly mitigated using a commit-and-reveal approach. In this approach, clients first post a commitment to their transaction. Once the transaction containing the commitment is posted, the party posts a transaction containing the opening of the commitment. Thanks to the hiding of the commitment, when Alice publishes the commitment to $\mathsf{perm\text{-}info}, \mathsf{S}$, and req, Jeff will not know what $\mathsf{perm\text{-}info}$ is committed and will not be able to front-run Alice. The commit-and-reveal approach increases the overhead on the blockchain and the users, however, for our setting, this is not problematic since permission transactions are expected to be infrequent. Hence, the protocol discussed above is slightly modified to follow commit-and-reveal. A parameter t_{open} is then used to establish how many blocks may pass before the opening must be posted[7]. If Alice is concerned with Jeff using network traffic to determine the source of her commitment, she could use an anonymous network such as Tor [9] or, if the ledger is a bitcoin-like ledger, Dandelion [54].

A Non-Cryptographic Attack. Note that although hiding guarantees that Jeff cannot detect which $\mathsf{perm\text{-}info}$ is committed in the permission request, there is still a possibility that Jeff tries to mount a denial of service attack to the system by publishing commitments to all $\mathsf{perm\text{-}info}$s, with the hope of guessing the one that is committed in an honest transaction[8]. Furthermore, Jeff might be someone that knows that Alice was robbed of their phone and could be the one posting the request.

[5]This field is specific to the application for which the permission is used. Looking ahead to Secret Recovery, req will be a fresh public key pk that the TEE will use to re-encrypt the secret it holds for Alice

[6]This step might seem redundant but it will become clear later why we break it down into two transactions.

[7]This prevents Jeff from posting a commitment and waiting to front-run Alice's opening, even if it occurs long after Jeff's commitment.

[8]Note that request fees would financially deter Jeff from mounting this type of attack.

All such attacks are not cryptographic in nature, as they concern the ability of Jeff to predict which perm-info will be lost. The consequence of this is that two transactions with the same perm-info will appear on the blockchain. In this case, our protocol will ignore the request and no permission will be created. Note that this approach guarantees that in case of doubts, no one gets permission, which means that Alice's secrets remain protected (although they are not recoverable by Alice).

2.2 UC-Definition of Credential-less Secret Recovery $\mathcal{F}_{SecRec}$

We model secret recovery via the ideal functionality $\mathcal{F}_{SecRec}$ (Fig. 25), where a client Client can store a secret s with $\mathcal{F}_{SecRec}$ and a cloud Cloud is informed that Client has stored a secret and has a public identity perm-info. Client can ask $\mathcal{F}_{SecRec}$ to retrieve s at anytime. Upon this request, $\mathcal{F}_{SecRec}$, before sending s to Client, will first need the approval from Cloud. This step models the fact that in the real world, a cloud can refuse to provide service.

Any party P can request **Recovery**. If this request is associated with a valid permission perm, $\mathcal{F}_{SecRec}$ accepts and sends the secret s to this credential-less party P – again assuming that Cloud has agreed to provide this service. Thanks to our modular UC-definition, perm can be checked in $\mathcal{F}_{SecRec}$ by accessing $\mathcal{G}_{Perm}$. Finally, we also allow a client to remove her secrets from the cloud.

2.3 UC-Protocol Realizing $\mathcal{F}_{SecRec}$ in the $\mathcal{G}_{Perm}$ Hybrid World Using TEE

We provide a realization Π_{SecRec} of $\mathcal{F}_{SecRec}$ in the $\mathcal{G}_{Perm}$ hybrid world and using a TEE, modeled as an ideal functionality $\mathcal{G}_{att}$ [47] (Fig. 8). In the realization, the first step for Alice is to register with $\mathcal{G}_{Perm}$, choosing a public id perm-info and communicating the identity of the cloud Cloud she wants to associate the permission to.

Then, to store a secret s on Cloud's machine, Alice interacts with the TEE hosted by Cloud. $\mathcal{G}_{att}$ executes a simple program. (1) perform key agreement with Client with id perm-info, (2) process recovery requests for Client if they are authenticated with a valid permission perm.

After engaging in the key agreement[9], Alice and $\mathcal{G}_{att}$ have a shared "retrieval key" retK. This key is used by Alice to encrypt s, via a INT-CTX-secure encryption scheme, and obtain the ciphertext c, which is then stored by Cloud. As long as Alice remembers retK, she can retrieve s by simply downloading c and decrypting it with retK.

If Alice loses her key(s), she will use $\mathcal{G}_{Perm}$ to obtain perm for request req. In our protocol, perm represents permission for $\mathcal{G}_{att}$ to recover the secret and re-encrypt it under a new public key pk that Alice picks. Hence, in our protocol req = ("*recover*"||pk).

After obtaining perm from $\mathcal{G}_{Perm}$, Alice sends the pair (req, perm) to the cloud which will be used to operate $\mathcal{G}_{att}$. $\mathcal{G}_{att}$ is queried with input (req, perm, perm-info, c) and will first check that perm verifies for perm-info. If the permission is valid, $\mathcal{G}_{att}$ will decrypt c with retK and re-encrypt under pk.

Finally, to remove the stored secret from the system, Alice simply sends a signed removal request to $\mathcal{G}_{att}$ to remove her secrets from the enclave.

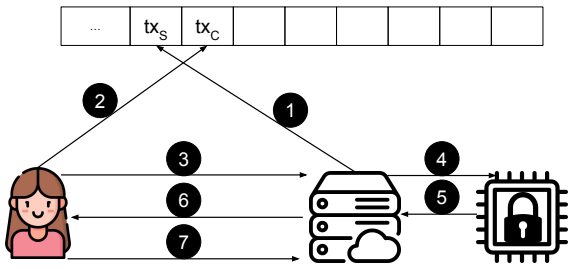
In Fig. 1, we illustrate the high-level flow of our secret recovery system implemented with Π_{Perm} and Π_{SecRec}

2.4 Software Implementation and Evaluation of Permission Mechanism and Secret Recovery

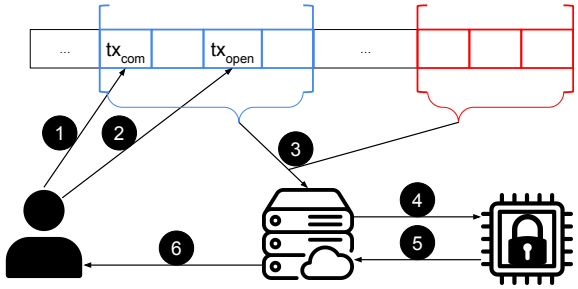
We implemented our Credential-less Secret Recovery system using the AWS NITRO enclave [12] and a simulation of Hyperledger Fabric [1] by following the BFT consensus algorithm [14], block validation, and default configuration provided in their documentation. Our implementation is written in the Python programming language using standard packages and demonstrates that all procedures of secret recovery, aside from recovery which is rare, are very fast (tens of milliseconds). Details of our implementation can be found in Sec. 7.

We chose to use the AWS Nitro Enclave primarily due to the ease of configuration. With Nitro, we are able to quickly adjust the memory and CPU allocation provided to the enclave. We chose to simulate

[9]Note that any secure key exchange would work, we chose DHKA for simplicity.



(a) Registration and Storage Procedures: 1 Cloud registers via ledger 2 Client registers with cloud via ledger 3 Client begins key exchange with TEE 4 Cloud forwards client message 5 TEE responds to key exchange 6 Cloud forwards TEE message 7 Client sends encryption of secret under shared key with TEE



(b) Recovery Procedure: 1 Party posts commitment to request to ledger 2 Party opens request 3 After waiting period, cloud takes ledger blocks as permission (commitment window in blue, challenge window in red) 4 Cloud inputs permission to TEE 5 After verifying permission, TEE outputs a fresh encryption of the stored secret 6 Cloud forwards fresh encryption to requesting party

Figure 1: Secret Recovery Protocol Π_{SecRec} Implemented with Permission Protocol Π_{Perm}

Hyperledger, as opposed to launching a Hyperledger testbed, due to the fact that our protocols use the blockchain as a simple bulletin board. Further, the simulation allows us to jump ahead of the wait period and quickly run our experiments.

We ran our protocols 100 times and provide the average runtime of the client, cloud, and enclave for each of the four procedures of Secret Recovery in Table 1. Our results show that the client, cloud, and enclave require less than one second to **Store** a secret, **Retrieve** the secret, and **Remove** the secret. **Recover** is the most time-intensive task in our protocol, and we found that even this procedure requires just over one minute of compute time (excluding the waiting period).

This implementation demonstrated the importance of keeping permission small. The default preferred block size in Hyperledger is 2MB, with blocks being posted at most every two seconds. This is at odds with the need to give clients enough time to respond to malicious requests. If we give a client one week to deny a request, and allow them to deny at any time, then permission includes over 300,000 blocks, totaling about 600GB. Clearly, we cannot expect the enclave to verify this amount of data in a timely fashion. Hence our decision to include the waiting period, giving the client enough time to respond to a malicious request, but providing a much shorter window to post the response keeping permission to a reasonable size. It is important to note that this decision means that the compute time of **Recover** is *independent of the size of* t_{wait}. We discuss further optimizations of our implementation in Sec. 7.3.

2.5 Areas for Improvement.

Lastly, we discuss the weak points of our protocols and suggest future improvements to strengthen the results.

Monitoring the Ledger. The client must detect requests made on their behalf, else they could lose their secret. The monitoring, and *only* the monitoring could be offloaded to a hired server, who notifies the client that they need to post a denial.

Proof of Silence. In a proof of silence, it could be the case that the client simply did not see the request. Our solution has the goal of providing a route for recovery when the client has forgotten *everything*, however in practice it may be worthwhile to introduce a measure that allows a client to recover with a short secret (e.g. PIN, password) before defaulting to a proof of silence.

Competing Requests. If two users claim the same identity, neither will have a way to prove themselves. One possible approach is to accept the first request, as is done in KELP [20]. However, if Jeff learned of credential-loss in real time (e.g., by stealing Alice's devices), he could request immediately and potentially steal Alice's secret. Therefore we opt to protect Alice's privacy and accept neither request. Note that

Functionality: $\mathcal{G}_{Perm}$

Participants: A set of servers $\mathcal{S}$, a set of clients $\mathcal{C}$, a party P, the adversary $\mathcal{A}$
Variables: L, the set of registered servers, L_{S} the list of clients registered to server S
External Functionalities: $\mathcal{G}_{refClock}$ the global clock functionality (Fig. 7)
Algorithms: VerifyPerm checks the validity of permissions
Procedures:

- **Registration - Server** Upon receipt of $(\texttt{register server}, \mathsf{S})$ from $\mathsf{S} \in \mathcal{S}$ for the first time, add S to L, set $L_{\mathsf{S}} = \varnothing$, and send $(\texttt{registered}, \mathsf{S})$ to $\mathcal{C}$ and $\mathcal{A}$
- **Registration - Client** Upon receipt of $(\texttt{register client}, \mathsf{C}, \mathsf{S})$ from $\mathsf{C} \in \mathcal{C}$ for the first time, send $(\texttt{registration request}, \mathsf{C}, \mathsf{S})$ to $\mathcal{A}$
  1. Upon receipt of $(\texttt{client perm-info}, \mathsf{C}, \mathsf{perm\text{-}info})$ from $\mathcal{A}$ add $(\mathsf{perm\text{-}info}, \mathsf{S}, \perp)$ to L_{S} and send $(\mathsf{perm\text{-}info}, \mathsf{S})$ to C, S, and $\mathcal{A}$
- **Generate Permission** Upon receipt of $(\texttt{generate permission}, \mathsf{perm\text{-}info}, \mathsf{S}, \mathsf{req})$ from party P, send $(\texttt{permission request}, \mathsf{perm\text{-}info})$ to S and $\mathcal{A}$
  1. If $(\mathsf{perm\text{-}info}, \mathsf{S}, \mathsf{req}\|\mathsf{res}\|\mathsf{perm}) \notin L_{\mathsf{S}}$ then output $(\texttt{nonexistent client}, \mathsf{perm\text{-}info})$ to P, S, and $\mathcal{A}$
  2. Else send $(\texttt{permission requested}, \mathsf{perm\text{-}info}, \mathsf{S}, \mathsf{req})$ to C and receive a response res, where res can be $\texttt{accepted}, \texttt{denied}$, or $\texttt{silent}$ and let t_{elapse} be the time between sending and receiving a response according to $\mathcal{G}_{refClock}$
     (a) If $\mathsf{res} = (\texttt{accepted})$ send $(\texttt{acceptance proof}, \mathsf{perm\text{-}info}, \mathsf{S}, \mathsf{req}, t_{elapse})$ to $\mathcal{A}$ and receive perm
     (b) Else if $\mathsf{res} = (\texttt{denied})$ send $(\texttt{denial proof}, \mathsf{perm\text{-}info}, \mathsf{S}, \mathsf{req}, t_{elapse})$ to $\mathcal{A}$ and receive perm
     (c) Else send $(\texttt{silent}, \mathsf{perm\text{-}info}, \mathsf{S}, \mathsf{req})$ to $\mathcal{A}$ and receive perm
     (d) Add $(\mathsf{perm\text{-}info}, \mathsf{S}, \mathsf{req}\|\mathsf{res}\|\mathsf{perm})$ to L_{S} and send $(\texttt{permission}, \mathsf{perm\text{-}info}, \mathsf{S}, \mathsf{req}\|\mathsf{perm})$ to P, S, and $\mathcal{A}$
- **Verify Permission** Upon receipt of $(\texttt{verify permission}, \mathsf{perm\text{-}info}, \mathsf{S}, \mathsf{req}, \mathsf{perm})$ from party P, if there exists $(\mathsf{perm\text{-}info}, \mathsf{S}, \mathsf{req}\|\mathsf{res}\|\mathsf{perm}) \in L_{\mathsf{S}}$
  1. Let $\mathsf{VerifyPerm}(\mathsf{perm\text{-}info}, \mathsf{S}, \mathsf{req}, \mathsf{perm}) = b_{ver}$
  2. If $\mathsf{res} = \texttt{accepted}$ output $(\texttt{accepted}, \mathsf{perm\text{-}info}, \mathsf{S}, \mathsf{req}\|\mathsf{perm})$ to P and $\mathcal{A}$
  3. Else if $\mathsf{res} = \texttt{denied}$ output $(\texttt{denied}, \mathsf{perm\text{-}info}, \mathsf{S}, \mathsf{req}\|\mathsf{perm})$ to P and $\mathcal{A}$
  4. Else if $\mathsf{res} = \texttt{silence}$, if $b_{ver} = 1$ output $(\texttt{accepted}, \mathsf{perm\text{-}info}, \mathsf{S}, \mathsf{req}\|\mathsf{perm})$ to P and $\mathcal{A}$. Else output $(\texttt{denied}, \mathsf{perm\text{-}info}, \mathsf{S}, \mathsf{req}\|\mathsf{perm})$ to P and $\mathcal{A}$
  5. Else output $(\texttt{denied}, \mathsf{perm\text{-}info}, \mathsf{S}, \mathsf{req}\|\mathsf{perm})$ to P and $\mathcal{A}$

Figure 2: $\mathcal{G}_{Perm}$ The Global Functionality for Verifiable Permission

competing requests are unlikely: two parties must make a request within the same short window. KELP also includes "request fees" to dissuade an adversary from sending out requests for all, or most, clients in the hopes of a client being unable to deny the request. The same could be implemented here to reduce the likelihood of competing requests. The fees would be paid to either Alice, if she denies, or whichever party successfully earns permission. This would make monitoring the ledger and denying transactions rewarding for Alice, and also makes the above guessing attack costly to an adversary.

Avoiding Trusted Execution Environment. The work performed by the TEE could be replaced with a secure-two-party protocol between two non-colluding servers. This can be performed using a two-out-of-two secret sharing scheme. Note that, while this resembles a simple threshold escrow approach, the main difference is that in escrow approaches, there is still an implicit assumption that the client who asks the server to reconstruct her secret *must still be authenticated* in some manner. Indeed, if this was not the case, then anyone can ask the server to perform reconstruction.

Instead, in our approach a client would authenticate with the server using the credential-less permission system. We discuss the fundamental difference between our setting and the escrow setting in Sec. 3.

3 Related Work

Break-glass Encryption. Our work is inspired by the concept of "Break-glass Encryption" introduced by Scafuro [50], which we compare to here.

Break-glass encryption allows Alice to decrypt her ciphertexts stored on the cloud, even when she loses her decryption key. Break-glass encryption does not provide confidentiality in the presence of a malicious cloud, even if Alice possesses her credentials, but only detectability. In other words, in [50] at any point, the cloud can ask the TEE to "decrypt" Alice's ciphertext without having any valid permission. The only guarantee is that such behaviour will be detectable because the TEE will "leave a mark" on Alice's ciphertext, and Alice will recognize this mark when she tries to download her ciphertext. To leave such a mark, the cloud must *continuously* update Alice's ciphertext. Instead, in our work we demand that confidentiality is guaranteed even in the presence of a malicious cloud (when Alice possesses her credentials).

[50] uses the credential-less permission in a very different way. In [50] the permission is used to protect an honest cloud against a malicious client that attempts to accuse them of malicious behavior. In [50], the TEE uses the permission as a string that needs to be encrypted in the ciphertext provided in the output. The permission is communicated to the user as a proof that the break-glass procedure was performed in response to a permission. In contrast, in our work, the TEE must verify the permission before accessing the secret. This protects Alice's confidentiality even when the cloud is malicious (when Alice has her credentials). On the downside, for the TEE to be able to verify the permission, the latter must be fully verifiable by a constrained machine such as TEE that is not connected to the internet.

Besides these major differences in how credential-less permission is used, our work improves on [50] in several ways. We provide a concrete realization of the permission functionality $\mathcal{G}_{Perm}$ (which we revised and improved its modularity) and we prove its security in the UC-framework, while [50] provides only an informal description. Since we provide a concrete instantiation, we also unveil issues that were not foreseen in [50], such as the front-running attacks. Compared to [50], our recovery protocol is very practical, requiring our cloud to only store one ciphertext and query the TEE only four times: twice upon storage, once in the case of removal, and once in case of emergency recovery. In contrast, [50] requires the cloud to use the TEE continuously to update the client's ciphertexts with bookkeeping information, necessary to achieve detectability.

Recovery of Cryptowallets. The problem of recovery is most popular in blockchain settings, as in our previous example of a user losing the key to their cryptocurrency wallet and subsequently their funds. Some approaches aim to solve this specific example, by providing a method for recovering funds in a wallet without regaining access to that account [51, 21]. One approach to this is *pre-signing* a transaction that transfers all funds to a secondary account [53]. However, this requires management of the second account, and it is not clear how this approach for UTXO wallets. The work by Blackshear et al. [20] also focuses on this specific problem, and provides a mechanism that allows the owner of a wallet, who forgot the key, to replace her old wallet with a fresh wallet for which she does know the key. Their approach uses time-lock commitments and

smart contracts [20]. These solutions, however, only apply to cryptocurrency wallets and do not let the user regain access to their account, only transfer the funds.

For users who want to regain access to their account, some wallets come equipped with their own recovery mechanism [3, 4, 7, 8, 11, 39]. We refer the reader to [24] for more detail on these solutions, as our work solves the more general problem of recovering any kind of secret stored on a cloud.

Key-Escrow. Other approaches for recovering a forgotten key are based on key-escrow. Key-escrow is an approach to key recovery where a trusted party stores the key on behalf of the key-owner [15, 29]. The key can also be split into parts and shared among trustees of the authority so that cooperation from a threshold amount of the trustees is required [43, 52]. Similarly, a user might escrow only part of their key, say half of the bits of the key, so the user need only remember half of the key, and the authority is not given the full key [16]. Due to the trusted nature of the authority, the key can be used by the authority at any time. In fact, many applications of key escrow involve law enforcement playing the role of the authority and using the key for "authorized wiretaps". Key escrow might be the best approach for settings where the users and the escrow entities are well-known to each other or have real-world/physical addresses that can be reached (e.g., they have established businesses with established trust). However, when the user has only a digital identity, there is a problem in identifying whether a recovery request is coming from the legitimate user. Hence, even with escrow one needs to use a credential-less permission mechanics to protect the users.

Acsesor [23] provides a solution where Alice encrypts her secret and shares the key among a set of guardians. In Acsesor, Alice chooses her own policy, which defines the circumstances under which recovery can occur. This solution, however, requires that Alice authenticates with a server, who acts as a middleman between Alice and the guardians, to initiate recovery. In our solution, we provide a way for Alice to recover her secret even when she has no way of authenticating

Multi-Cloud Storage. In a recent work, KAPRE and KAME [37] provide the latest in a string of work [55, 19, 48, 42, 45, 56] on multi-cloud storage systems. Both protocols involve a user storing data privately on a set of cloud service providers (CSPs) through an untrusted proxy. This proxy handles all communication and most of the computation on behalf of the client. In KAPRE, the client samples a symmetric key and encrypts their data. Next, the client prepares and encrypts the coefficients for Shamir secret sharing under an additively homomorphic re-encryption scheme, and prepares a share in plaintext for the proxy. The proxy then uses these encryptions of coefficients to compute and distribute shares of the key to the CSPs and shares the ciphertext of the data via an Information Dispersal Algorithm. Re-encryption is used to convert the homomorphically computed shares to be under the public key of the respective CSP, so they may decrypt and hold a share in the clear. This prevents the proxy from violating privacy from seeing any share other than their own in the clear. KAPRE follows a similar approach, however it uses multikey additively homomorphic encryption in place of re-encryption.

The two protocols have the same download phase. To download, the client sends nonces, encrypted under CSP public keys respectively, to the proxy for distribution. Upon decrypting their nonce, each CSP uses the nonce to mask their share. Then, each CSP can send their masked share to the proxy. The proxy is then able to use these masked shares to compute a masked version of the client's symmetric key. Finally, the proxy returns the masked key and the reconstructed ciphertext to the client, who unmasks the key and decrypts the data.

KAPRE achieves privacy against an adversary corrupting all of the CSPs assuming an honest proxy and KAME achieves privacy against the proxy colluding with all but one of the CSPs. The main contribution of these protocols is that much of the workload is offloaded to a proxy *and* the client need not remember a key at any point after storage *assuming the client has some method of authenticating*. We are still the first to provide a method of recovery *including authentication* that requires no key, although our protocol does require the client remember a signing key up to the point of recovery. Further, KAPRE and KAME assume that at least the proxy or at least one CSP are honest, whereas we do not assume our cloud is honest, only that the TEE is created correctly.

Authentication from Alternative Credentials. Another popular approach is to lock secrets behind a password, passphrase, or passcode, as these are often easier to remember than a key. For example, in Torus [10], users provide an email for them to receive a backup passphrase to recover their wallet. Similarly, users of the Trezor Hardware Wallets are able to recover their wallets using a seed phrase of 12, 18, 20, 24, or 33 words [6]. While seed phrases are meant to be made up of easily remembered words, it can still be a challenge to remember that many words. Rather than longer seed phrases, in SafetyPin [27], users are able

to recover their cloud-based mobile backups using a short PIN. This paper also provides a safeguard against brute-force guessing attacks against these PINs. Lastly, there are approaches based on password-protected secret sharing [32, 33]. These protocols allow a client to prove knowledge of a password to a server, without revealing said password, to obtain a previously stored secret. Furthermore, we aim to provide a route for users who have forgotten *everything*, including any password, passphrase, or passcode.

To provide support for these users who have forgotten everything, there is also the method of biometric-based recovery. One approach to using biometrics in this space is to use biometric-based encryption to encrypt the key, that way it can always be decrypted using a fingerprint upon recovery [13]. The user thus need not remember anything, however, in this solution, the encrypted key is split among a set of "stewards" who are trusted and must be online for recovery. Our solution has a single cloud, who is not trusted but does run a TEE, and we do not require costly biometric encryption, rather standard encryption.

The work by Maram, Kelkar, and Eyal [40] considers a related problem that they call the authentication problem. The authors consider a scenario of two parties, an honest party and an adversary, both interacting with a mechanism to prove identity. This mechanism will use some set of credentials, or other facts that the honest party should know, to verify the identity. We instead consider the case where the user has no credentials left, rather the user has lost everything, and can still recover a stored secret.

4 Background

In this section we present the tools that our constructions are built on.

4.1 Symmetric key encryption

We revisit the definition from Chapter 3 of [35]. Let $\Pi := (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be a symmetric key encryption scheme where:

- Gen is the key generation algorithm that takes as input 1^λ and outputs a key k. Concretely, $k \leftarrow\!\!\$\ \mathsf{Gen}(1^\lambda)$.
- Enc is the encryption algorithm that takes as input a key k and a message $m \in \{0,1\}^*$ and outputs a ciphertext c. That is, $c \leftarrow\!\!\$\ \mathsf{Enc}(k, m)$.
- Dec is the decryption algorithm that takes as input a key k and a ciphertext c and outputs a message m or an error ($\perp$). That is $m = \mathsf{Dec}(k, c)$.

For completeness, it is required that for every $\lambda \in \mathbb{N}$, every $k \leftarrow\!\!\$\ \mathsf{Gen}(1^\lambda)$, and every $m \in \{0,1\}^*$, it holds that $\mathsf{Dec}_k(\mathsf{Enc}_k(m)) = m$.

In Fig. 3, we show the IND-CPA game presented in [35] for a symmetric encryption scheme as described before. This game is between a challenger and a PPT adversary $\mathcal{A}$.

$\mathsf{PrivK}^{cpa}_{\mathcal{A},\Pi}(\lambda)$

1. $k \leftarrow\!\!\$\ \mathsf{Gen}(1^\lambda)$.
2. $m_0, m_1 \leftarrow \mathcal{A}^{\mathsf{Enc}_k(\cdot)}(1^\lambda)$ s.t. $|m_0| = |m_1|$.
3. $b \leftarrow\!\!\$\ \{0,1\}$, $c^* \leftarrow\!\!\$\ \mathsf{Enc}_k(m_b)$.
4. $b' \leftarrow \mathcal{A}^{\mathsf{Enc}_k(\cdot)}(c^*)$.
5. If $b' = b$ output 1, else output 0.

Figure 3: IND-CPA for Π

We say an encryption scheme Π is IND-CPA secure if:

$$\Pr[\mathsf{PrivK}^{cpa}_{\mathcal{A},\Pi}(\lambda) = 1] \leq \frac{1}{2} + \mathsf{negl}(\lambda).$$

Additionally, we provide the INT-CTX experiment [17, 50] in Fig. 4 for a symmetric encryption scheme $\Pi := (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ for reference.

$Exp^{INT-CTX}_{\mathcal{A},\Pi}(1^\lambda)$

- $K \leftarrow\!\$\ \Pi.\mathsf{Gen}(1^\lambda)$. Initialize $S = \emptyset$, $\mathsf{win} = \mathsf{false}$ and provide oracle access to $\Pi.\mathsf{Enc}_K(\cdot)$ to $\mathcal{A}$.
- For any query M_i, $C_i \leftarrow\!\$\ \Pi.\mathsf{Enc}_K(M_i)$, $S = S \cup \{C_i\}$.
- For any query $VF(C)$, $M \leftarrow\!\$\ \Pi.\mathsf{Dec}_K(C)$. If $M \neq \perp$ and $C \notin S$ return 1 and set $\mathsf{win} = \mathsf{true}$.
- After receiving "Finalize", output win.

Figure 4: INT-CTX for Π

We say an encryption scheme Π is INT-CTX secure if:

$$\Pr[Exp^{INT-CTX}_{\mathcal{A},\Pi}(1^\lambda) = \mathsf{win}] \leq \mathsf{negl}(\lambda).$$

4.2 Public key encryption.

Next we discuss the definition from Chapter 11 of [35]. Let $\Pi_{pub} := (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be a public key encryption scheme and M be a message space where:

- Gen is the key generation algorithm that takes as input 1^λ and outputs a key pair (pk, sk). Concretely, $(pk, sk) \leftarrow\!\$\ \mathsf{Gen}(1^\lambda)$.
- Enc is the encryption algorithm that takes as input a public key pk and a message m from M and outputs a ciphertext c. That is, $c \leftarrow\!\$\ \mathsf{Enc}(pk, m)$.
- Dec is the decryption algorithm that takes as input a secret key sk and a ciphertext c and outputs a message m or an error ($\perp$). That is $m = \mathsf{Dec}(sk, c)$.

For completeness, it is required that for every $\lambda \in \mathbb{N}$, every $(pk, sk) \leftarrow\!\$\ \mathsf{Gen}(1^\lambda)$, and every $m \in M$, it holds that $\mathsf{Dec}_{sk}(\mathsf{Enc}_{pk}(m)) = m$ except with negligible probability.

$\mathsf{PubK}^{cpa}_{\mathcal{A},\Pi_{pub}}(\lambda)$

1. $(pk, sk) \leftarrow\!\$\ \mathsf{Gen}(1^\lambda)$.
2. $m_0, m_1 \leftarrow \mathcal{A}(pk)$ s.t. $|m_0| = |m_1|$ and $m_0, m_1 \in M$.
3. $b \leftarrow\!\$\ \{0,1\}$, $c^* \leftarrow\!\$\ \mathsf{Enc}_{pk}(m_b)$.
4. $b' \leftarrow \mathcal{A}(c^*)$.
5. If $b' = b$ output 1, else output 0.

Figure 5: IND-CPA for Π_{pub}

We capture the IND-CPA experiment for public key encryption in Fig. 5. We say an encryption scheme Π_{pub} is IND-CPA secure if:

$$\Pr[\mathsf{PubK}^{cpa}_{\mathcal{A},\Pi_{pub}}(\lambda) = 1] \leq \frac{1}{2} + \mathsf{negl}(\lambda).$$

4.3 Digital signature schemes.

Now we revisit the definition from chapter 12 of [35]. A digital signature scheme $\Sigma := (\mathsf{Gen}, \mathsf{Sig}, \mathsf{Vf})$ for a message space M is a tuple of three PPT algorithms such that:

- Gen is the key generation algorithm that takes 1^λ as input and outputs a pair of public and private keys (pk, sk). Concretely $(pk, sk) \leftarrow\!\$\ \mathsf{Gen}(1^\lambda)$.
- Sig is the signing algorithm that takes a private key sk and a message m from the message space M as input and outputs a signature σ. We write this as $\sigma \leftarrow\!\$\ \mathsf{Sig}(sk, m)$.
- Vf is the verification algorithm that takes a public key pk, a message m, and a signature σ as input and outputs 1 if a valid signature and 0 if an invalid signature. We write this as $b = \mathsf{Vf}(pk, m, \sigma)$.

For completeness, we require that for $(pk, sk) \leftarrow\$ \mathsf{Gen}(1^\lambda)$, $m \in M$, it holds that $1 = \mathsf{Vf}(pk, m, \mathsf{Sig}(sk, m))$ except with a negligible probability.

We present the game in Fig. 6 to capture unforgeability. We say a signature scheme Σ is existentially unforgeable against chosen-message attacks (i.e. secure) if for all PPT $\mathcal{A}$ we have that $\Pr[\mathsf{Sig\text{-}Forge}_{\mathcal{A},\Sigma}(\lambda) = 1] \leq \mathsf{negl}(\lambda)$.

$\mathsf{Sig\text{-}Forge}_{\mathcal{A},\Sigma}(\lambda)$

1. $(pk, sk) \leftarrow\$ \mathsf{Gen}(1^\lambda)$.
2. $m, \sigma \leftarrow \mathcal{A}^{\mathsf{Sig}_{sk}(\cdot)}(pk)$ let $\mathcal{Q}$ be the set of queries $\mathcal{A}$ asked to the oracle.
3. If $1 = \mathsf{Vf}(pk, m, \sigma)$ and $m \notin \mathcal{Q}$ output 1; otherwise output 0.

Figure 6: EUF-CMA forΣ

4.4 Commitment Schemes

A commitment scheme $\mathtt{COM}$ is a tuple of algorithms $(\mathsf{Commit}, \mathsf{Open})$ that allows a party to produce a commitment $\mathtt{com}$ to a value x. We make use of a statistically hiding and computationally binding commitment scheme [26], defined as:

- A commitment scheme is considered *statistically hiding* if for any x, x' $\mathsf{Commit}(x)$ and $\mathsf{Commit}(x')$ are statistically indistinguishable (as defined by [30]) [26]
- A commitment scheme is considered *computationally binding* if the probability that any PPT adversary can produce $\mathtt{com}, \mathtt{open}, \mathtt{open}'$, such that $\mathtt{open} \neq \mathtt{open}'$ but both open $\mathtt{com}$ is less than $\mathsf{negl}(\lambda)$ [35]

4.5 Global Clock Functionality

Next, we present the global reference clock functionality $\mathcal{G}_{refClock}$ [22] (Fig. 7). This functionality is used by our ideal functionality $\mathcal{G}_{Perm}$ to determine the amount of time that has passed between notifying a client of a request for permission on their behalf, and the client responding with an acceptance, denial, or silence.

Functionality: $\mathcal{G}_{refClock}$

Participants: The environment $\mathcal{Z}$, some party P
Variables: An integer G representing the time, initially 0

Procedures:

- **Increment Time**
  - Upon receipt of (`increment time`) from $\mathcal{Z}$, set $G = G + 1$ and send (`incremented`) to $\mathcal{Z}$. Ignore any (`increment time`) from any other party
- **Get Time**
  - Upon receipt of (`get time`) from a party P, send $(\mathtt{time}, G)$ to P

Figure 7: $\mathcal{G}_{refClock}$ The Global Functionality for a Reference Clock

4.6 Proof-of-Publication Ledger $\mathcal{L}$.

We assume that the parties have access to an unforgeable, verifiable ledger $\mathcal{L}$ as modeled in [34]. The concept of unforgeability here is the same as the unforgeability of signatures. Upon posting a transaction tx to a ledger with chain ID cid, the posting party receives an authentication tag σ such that it is hard to compute

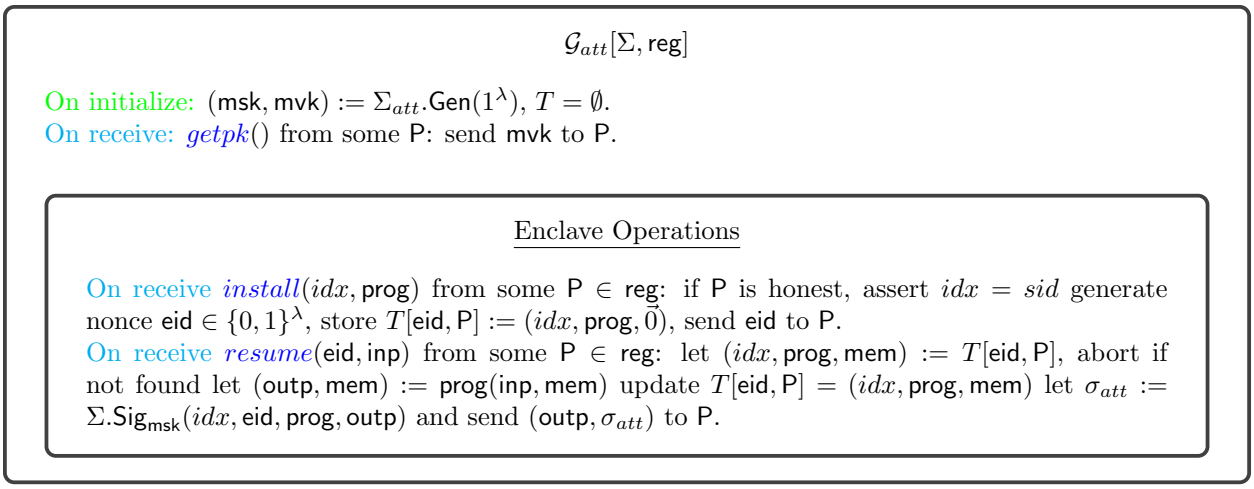


Figure 8: $\mathcal{G}_{att}$ The Ideal Functionality of a TEE

σ without posting tx to the ledger. cid allows us to identify a specific chain of posts, and for the purpose of our permission protocol we will assume that only posts pertaining to the permission protocol and server S will be made to the chain with ID cid. The structure of cid is dependent on the instantiation of $\mathcal{L}$. At a high level, $\mathcal{L}$ provides the following:

- $(\mathsf{tx}, \sigma) \leftarrow \mathcal{L}.\mathsf{Post}(z, \mathsf{cid})$: This allows a user to post $\mathsf{data} = z$ on the append only ledger for the chain identifier, cid. The output of this interface is the transaction, tx, and an authentication tag σ for verifying that the data was posted on the ledger. We will often use tx.data to refer to the contents posted, in this case z. Further, we use tx.idx to refer to the integer index identifying the block that tx appears in on the ledger
- $\{0,1\} \leftarrow \mathcal{L}.\mathsf{Verify}(\mathsf{tx}, \sigma)$: This allows any user to verify that the pair (tx, σ) were the result of the $\mathcal{L}.\mathsf{Post}$ procedure above

We use this ledger due to the public verifiability, which allows our TEE to verify transactions *without having access to the ledger*. It is vital that the TEE be able to verify transactions so that a malicious server is not able to produce forged permission.

4.7 Trusted Execution Environment.

The Cloud has access to a trusted execution environment (TEE) represented as the ideal functionality $\mathcal{G}_{att}$ (Fig. 8) presented in [47]. Recall that a TEE is an enclave on a computer that can securely perform computation and store data. In Fig. 8 all Blue activation points are activation points that can be executed more than once. However, Green activation points can be only executed once. Let prog be the program run by the enclave. Upon performing this computation, the TEE returns an attestation, which is a signature based on keys provided by the manufacturer.

At a high level, $\mathcal{G}_{att}$ is defined for a signature scheme Σ_{att} and registry reg that lists the parties equipped with a processor containing the TEE, these parties are known as the host. Upon initialization (at the factory) the TEE enabled processor is initialized with a signature key pair msk, mvk. This initialization is only performed once, and the signature pair is used for attesting the execution of a program in the TEE. $\mathcal{G}_{att}$ allows parties to query the public verification key, mvk. In practice, this is often implemented as an online trusted resource where users can verify an attestation from the TEE.

To use the TEE, the host P first calls $install(idx, \mathsf{prog})$ to install the defined program, prog. $\mathcal{G}_{att}$ checks that $\mathsf{P} \in \mathsf{reg}$ and $idx = sid$, where idx is provided by the host and sid is the session ID stored by the TEE. Then it generates a random identifier, eid, to identify the installed enclave and returns eid to party, P.

Next to run the program on any input, inp, the host calls $resume(\mathsf{eid}, \mathsf{inp})$, and the TEE runs the program defined for eid on the input inp. Finally, it returns the output along with a signature under msk as its attestation, which can be verified using the public mvk.

5 Credential-less Permission Mechanism via Blockchain

Protocol: Π_{Perm} - Register and Manage Permissions

- **Register - Server**
  - S: Upon receiving $(\texttt{register server}, \mathsf{S})$ from $\mathcal{Z}$
    1. Set $\mathbb{T}_{\mathsf{S}} = \varnothing$
    2. Generate signature keys: $(\mathsf{vk}_{\mathsf{S}}, \mathsf{sk}_{\mathsf{S}}) \leftarrow \Sigma.\mathsf{Gen}(1^{\lambda})$ and choose a chain id cid.
    3. Post verification key on the ledger: $\mathsf{tx}_{\mathsf{S}}, \sigma_{\mathsf{S}} \leftarrow \mathcal{L}.\mathsf{Post}(\text{“}register\text{”} \| \mathsf{vk}_{\mathsf{S}}, \mathsf{cid})$
- **Register - Client**
  - C: Upon receiving $(\texttt{register client}, \mathsf{S})$ from $\mathcal{Z}$, find $\mathsf{tx}_{\mathsf{S}}, \sigma_{\mathsf{S}}$ on $\mathcal{L}$ with $\mathsf{tx.data} = \text{“}register\text{”} \| \mathsf{vk}_{\mathsf{S}}$
    1. Generate keys: Set $(\mathsf{vk}_{\mathsf{C}}, \mathsf{sk}_{\mathsf{C}}) \leftarrow \Sigma.\mathsf{Gen}(1^{\lambda})$
    2. Choose $t_{open}, t_{wait}, t_{chal}$ as the number of blocks allotted to open a commitment, the number of blocks to wait, and the number of blocks allotted to challenge a permission respectively
    3. Set permission identity: $\mathsf{perm\text{-}info} = (\mathsf{vk}_{\mathsf{C}}, t_{open}, t_{wait}, t_{chal}, \mathsf{vk}_{\mathsf{S}})$
    4. Post registration on the ledger: $\mathsf{tx}_{\mathsf{C}}, \sigma_{\mathsf{C}} \leftarrow \mathcal{L}.\mathsf{Post}(\mathsf{perm\text{-}info}, \mathsf{cid})$ and send $\mathsf{tx}_{\mathsf{C}}, \sigma_{\mathsf{C}}$ to S
  - S : Upon receiving $(\mathsf{tx}_{\mathsf{C}}, \sigma_{\mathsf{C}})$ from C
    1. Check for prior registration: If $\mathsf{perm\text{-}info} \in L_{\mathsf{S}}$ abort
    2. Authorize registration: Compute $\sigma_{sig} = \Sigma.\mathsf{Sign}(\mathsf{sk}_{\mathsf{S}}, \mathsf{tx}_{\mathsf{C}}.\mathsf{data})$, post $(\mathsf{tx}_{reg}, \sigma_{reg}) = \mathcal{L}.\mathsf{Post}(\mathsf{tx}_{\mathsf{C}}.\mathsf{data} \| \sigma_{sig}, \mathsf{cid})$, and send $(\mathsf{tx}_{reg}, \sigma_{reg})$ to C
  - C : Upon receipt of $(\mathsf{tx}_{reg}, \sigma_{reg})$ from S
    1. If $\Sigma.\mathsf{Vf}(\mathsf{vk}_{\mathsf{S}}, \sigma_{sig}, \mathsf{tx}_{\mathsf{C}}.\mathsf{data}) \neq 1$ abort
- **Manage Permissions:** (Run continuously by C upon registration)
  - C : Upon seeing any request tx referencing C posted to the ledger wait t_{wait} blocks
    1. If $\mathcal{Z}$ sends $(\texttt{accepted}, \mathsf{tx})$
       (a) Sign the transaction to accept: $\sigma_{granted} = \Sigma.\mathsf{Sign}(\mathsf{sk}_{\mathsf{C}}, \text{“}accepted\text{”} \| \mathsf{tx.data})$
       (b) Post acceptance: $(\mathsf{tx}_{accept}, \sigma_{accept}) = \mathcal{L}.\mathsf{Post}(\text{“}accepted\text{”} \| \mathsf{tx.data} \| \sigma_{granted})$
    2. Else if $\mathcal{Z}$ sends $(\texttt{denied}, \mathsf{tx})$
       (a) Sign the transaction to deny: $\sigma_{refuse} = \Sigma.\mathsf{Sign}(\mathsf{sk}_{\mathsf{C}}, \text{“}denied\text{”} \| \mathsf{tx.data})$
       (b) Post denied: $(\mathsf{tx}_{denied}, \sigma_{denied}) = \mathcal{L}.\mathsf{Post}(\text{“}denied\text{”} \| \mathsf{tx.data} \| \sigma_{refuse})$
    3. Else do nothing

Figure 9: Registration and Manage Permissions Procedures of Verifiable Permission Protocol Π_{Perm}

Revisiting Definition of Credential-less Permissions in the UC-framework. In Fig. 2 we present $\mathcal{G}_{Perm}$ the ideal functionality for authenticated credential-less permission generation. In this functionality, a client C registers to a server S with public information $\mathsf{perm\text{-}info}$. This information is not secret and should be accessible to the client even after losing all secrets. Any party can request publicly-verifiable, unforgeable permissions on behalf of C. C has the opportunity to accept or deny the permission, or to request that it be generated by silence.

Credential-less Permission Protocol. Here we present our protocol Π_{Perm} to realize $\mathcal{G}_{Perm}$. Π_{Perm} is shown in Fig. 9 and 10, and we give the permission verification procedure in Fig. 11. We consider a setting in which we have many clients $\mathcal{C}$ where each $\mathsf{C} \in \mathcal{C}$ contracts some server $\mathsf{S} \in \mathcal{S}$, such that S needs permission to perform some task req. Because permission requires no secret, they can be requested by any party. For all requests, there is a brief buffer ζ added to the commitment window to ensure that there are not two valid requests in the same window.

Our protocol makes use of a proof-of-publication ledger $\mathcal{L}$ [34] (Sec. 4.6). $\mathcal{L}$ is a ledger such that upon posting a transaction tx to the ledger with id cid, an unforgeable tag σ[10] is output that verifies that the transaction is on the ledger. Further, σ is a *publicly verifiable* tag, that is, any party can verify the pair (tx, σ), even without access to the ledger.

[10] Unforgeability here is the same concept as the unforgeability of signatures

Protocol: Π_{Perm} - Generate and Verify Permission

- **Generate Permission**
  - A party P: Upon receiving $(\texttt{generate permissions}, \mathsf{perm\text{-}info}, \mathsf{S}, \texttt{req})$ from $\mathcal{Z}$
    1. Compute a commitment to the request: $(\texttt{com}, \texttt{open}) = \mathsf{Commit}(\mathsf{perm\text{-}info} \| \texttt{req} \| \mathsf{tx_C.idx})$
    2. Post commitment to the ledger: $\mathsf{tx}_{com}, \sigma_{com} \leftarrow \mathcal{L}.\mathsf{Post}(\texttt{com}, \mathsf{cid})$
    3. Open the commitment and post the opening: Upon seeing tx_{com} posted to the ledger, post $\mathsf{tx}_{open}, \sigma_{open} \leftarrow \mathcal{L}.\mathsf{Post}(\texttt{open}, \mathsf{cid})$
  - C: Upon seeing tx_{open} on $\mathcal{L}$ where $\mathsf{perm\text{-}info} \in \mathsf{tx}_{open}.\mathsf{data}$, wait t_{wait} blocks, then
    1. If $\mathcal{Z}$ sends $(\texttt{accepted})$
       (a) Sign the opening to accept the request: $\sigma_{granted} = \Sigma.\mathsf{Sign}(\mathsf{sk_C}, \text{“}accepted\text{”} \| \mathsf{tx}_{open}.\mathsf{data})$
       (b) Post the acceptance: $\mathsf{tx}_{accept}, \sigma_{accept} = \mathcal{L}.\mathsf{Post}(\text{“}accepted\text{”} \| \mathsf{tx}_{open}.\mathsf{data} \| \sigma_{granted}, \mathsf{cid})$
    2. Else if $\mathcal{Z}$ sends $(\texttt{denied})$
       (a) Sign the opening to deny the request: $\sigma_{refuse} = \Sigma.\mathsf{Sign}(\mathsf{sk_C}, \text{“}denied\text{”} \| \mathsf{tx}_{open}.\mathsf{data})$
       (b) Post the denial: $\mathsf{tx}_{denied}, \sigma_{denied} = \mathcal{L}.\mathsf{Post}(\text{“}denied\text{”} \| \mathsf{tx}_{open}.\mathsf{data} \| \sigma_{refuse}, \mathsf{cid})$
    3. Else if $\mathcal{Z}$ sends $(\texttt{silence})$, do nothing
  - S: Upon seeing tx_{open} on $\mathcal{L}$ where $\mathsf{perm\text{-}info} \in \mathsf{tx}_{open}.\mathsf{data}$, if $\mathsf{perm\text{-}info} \notin L_{\mathsf{S}}$ abort. Else,
    1. Set ledger blocks as challenge windows:
       (a) Let $\mathsf{comwindow_{req}}$ be the t_{open} blocks before and the $t_{open} + \zeta$ blocks after tx_{open}, and $\mathsf{chalwindow_{req}}$ be the t_{chal} blocks occurring t_{wait} blocks after tx_{open}
    2. Check for valid commitment: If there does not exist $\mathsf{tx}_{com} \in \mathsf{comwindow_{req}}$ such that $\mathsf{tx}_{open}.\mathsf{data}$ is a valid opening of $\mathsf{tx}_{com}.\mathsf{data}$, output $\perp$ and abort
    3. Verify permission request:
       (a) Check for acceptance: If there exists $\mathsf{tx}_{accept} \in \mathsf{chalwindow_{req}}$ such that $\Sigma.\mathsf{Vf}(\mathsf{vk_C}, \sigma_{granted}, \text{“}accepted\text{”} \| \mathsf{tx}_{open}.\mathsf{data}) = 1$ for $\sigma_{granted} \in \mathsf{tx}_{accept}.\mathsf{data}$, set $\mathsf{chalwindow_{req}}$ to be the ledger blocks from tx_{open} to tx_{accept}
       (b) Check for denial: Else if there exists $\mathsf{tx}_{denied} \in \mathsf{chalwindow_{req}}$ such that $\Sigma.\mathsf{Vf}(\mathsf{vk_C}, \sigma_{refuse}, \text{“}denied\text{”} \| \mathsf{tx}_{open}.\mathsf{data}) = 1$ for $\sigma_{refuse} \in \mathsf{tx}_{denied}.\mathsf{data}$, set $\mathsf{chalwindow_{req}}$ to be the ledger blocks from tx_{open} to tx_{denied}
       (c) Check for competing requests: Else if there exists $\mathsf{tx}'_{open} \in \mathsf{chalwindow_{req}}$ such that there exists $\mathsf{tx}'_{com} \in \mathsf{comwindow_{req}}$ where $\mathsf{tx}'_{open}.\mathsf{data}$ is a valid opening of $\mathsf{tx}'_{com}.\mathsf{data}$ for the same $\mathsf{perm\text{-}info}$ or a valid opening of tx_{com}, the distance between tx'_{open} and the commitment it opens is less than or equal to t_{open}, output $\perp$ and abort
    4. Set permission: Set $\mathsf{perm} = \mathsf{tx_S} \| \sigma_{\mathsf{S}} \| \mathsf{tx_C} \| \sigma_{\mathsf{C}} \| \mathsf{tx}_{reg} \| \sigma_{reg} \| \texttt{open} \| \mathsf{comwindow_{req}} \| \mathsf{chalwindow}_{req}$
    5. Post permission: Let tx_{com_1} be the first transaction in $\mathsf{comwindow_{req}}$. Let tx_{chal_1} be the first transaction in $\mathsf{chalwindow_{req}}$. Post $\mathsf{tx}_{fin}, \sigma_{fin} = \mathcal{L}.\mathsf{Post}(\mathsf{tx}_{reg}.\mathsf{data} \| \mathsf{tx}_{reg}.\mathsf{idx} \| \texttt{open} \| \mathsf{tx}_{com_1}.\mathsf{idx} \| \mathsf{tx}_{chal_1}.\mathsf{idx}, \mathsf{cid})$
- **Verify Permission**
  - A party P : Upon receipt of $(\texttt{verify permission}, \mathsf{perm\text{-}info}, \mathsf{S}, \mathsf{req}, \mathsf{perm})$ from $\mathcal{Z}$
    1. Run $\mathsf{VerifyPerm}(\mathsf{perm\text{-}info}, \mathsf{S}, \mathsf{req}, \mathsf{perm})$

Figure 10: Generate and Verify Permission Procedures of Verifiable Permission Protocol Π_{Perm}

Algorithm: $\{0,1\} \leftarrow \mathsf{VerifyPerm}(\mathsf{perm\text{-}info}, \mathsf{S}, \mathsf{req}, \mathsf{perm})$

1. Parse $\mathsf{perm} = \mathsf{tx_S}\|\sigma_\mathsf{S}\|\mathsf{tx_C}\|\sigma_\mathsf{C}\|\mathsf{tx}_{reg}\|\sigma_{reg}\|\mathtt{open}\|\mathsf{comwindow_{req}}\|\mathsf{chalwindow_{req}}$
2. Parse $\mathsf{tx_S.data} =$ "$register$"$\|\mathsf{vk_S}$ and $\mathsf{tx_C.data} = \mathsf{perm\text{-}info}'$
3. If $\mathsf{perm\text{-}info}' \neq \mathsf{perm\text{-}info} = (\mathsf{vk_C}, t_{open}, t_{wait}, t_{chal}, \mathsf{vk_S})$ output 0
4. Check perm-info of permission: If perm-info in $\mathsf{tx_C}$ is not the same as the perm-info in open, output 0
5. Check request: If $\mathsf{req} \notin \mathtt{open}$, output 0
6. Verify registration transaction: If $\mathcal{L}.\mathsf{Vf}(\mathsf{tx}_{reg}, \sigma_{reg}) \neq 1$ output 0
7. Verify transactions from commitment window and challenge window: If $\exists$ any pair (tx, σ) in $\mathsf{comwindow_{req}}$ or $\mathsf{chalwindow_{req}}$ s.t. $\mathcal{L}.\mathsf{Vf}(\mathsf{tx}, \sigma) \neq 1$ output 0
8. Verify order of the transactions: If the transactions in $\mathsf{comwindow_{req}}$ or $\mathsf{chalwindow_{req}}$ are not a direct sequence, output 0
9. Check client registration:
   (a) Parse $\mathsf{tx}_{reg} = \mathsf{tx_C.data}\|\sigma_{sig}$. If $\Sigma.\mathsf{Vf}(\mathsf{vk_S}, \sigma_{sig}, \mathsf{tx_C.data}) \neq 1$ output 0
10. Check opening and commitment: If there does not exists $\mathsf{tx}_{open} \in \mathsf{comwindow_{req}}$ s.t. $\mathsf{tx}_{open}.\mathsf{data} =$ open is a valid opening of $\mathsf{tx}_{com} \in \mathsf{comwindow_{req}}$, output 0
11. Else
    (a) If $\exists \mathsf{tx}_{denied} \in \mathsf{chalwindow_{req}}$ s.t. $\Sigma.\mathsf{Vf}(\mathsf{vk_{Client}}, \sigma_{refuse},$ "$denied$"$\|\mathsf{tx}_{open}.\mathsf{data}) = 1$ for $\sigma_{refuse} \in \mathsf{tx}_{denied}.\mathsf{data}$ output 0
    (b) If $\exists \mathsf{tx}_{accept} \in \mathsf{chalwindow_{req}}$ s.t. $\Sigma.\mathsf{Vf}(\mathsf{vk_{Client}}, \sigma_{granted},$ "$accepted$"$\|\mathsf{tx}_{open}.\mathsf{data})$ $= 1$ for $\sigma_{granted} \in \mathsf{tx}_{accept}.\mathsf{data}$ output 1
    (c) Else if $\exists \mathsf{tx}'_{open} \in \mathsf{comwindow_{req}}$ s.t. $\exists \mathsf{tx}'_{com} \in \mathsf{comwindow_{req}}$ where $\mathsf{tx}'_{open}.\mathsf{data}$ is a valid opening of $\mathsf{tx}'_{com}.\mathsf{data}$ for the same perm-info, and $\mathsf{tx}'_{com}.\mathsf{idx} < \mathsf{tx}_{open}.\mathsf{idx} + \zeta$, output 0
12. Else if $|\mathsf{comwindow_{req}}| = 2t_{open} + \zeta$, $|\mathsf{chalwindow_{req}}| = t_{chal}$, and $\mathsf{tx}_{chal_1}.\mathsf{idx} = \mathsf{tx}_{open}.\mathsf{idx} + t_{wait}$ output 1

Figure 11: The Algorithm VerifyPerm for Local Verification of Permission

Simulator: $\mathsf{Sim}_{\mathsf{S}^*}$ **- Registration**

- **Register - Server:** Upon query ("*register*"$\|\mathsf{vk}_{\mathsf{S}}$) to $\mathsf{OLedger}$
  1. Send $(\texttt{register server}, \mathsf{S}^*)$ to $\mathcal{G}_{Perm}$
  2. Forward the query to $\mathsf{OLedger}$ and receive $\mathsf{tx}_{\mathsf{S}^*}, \sigma_{\mathsf{S}^*}$, store this in L_{cid} and forward to S^*
- **Register - Client:** Upon receipt of $(\texttt{registration request}, \mathsf{C}, \mathsf{S}^*)$ from $\mathcal{G}_{Perm}$
  1. Set $(\mathsf{vk}_{\mathsf{sim}}, \mathsf{sk}_{\mathsf{sim}}) \leftarrow \Sigma.\mathsf{Gen}(1^\lambda)$
  2. Choose $t_{open}, t_{wait}, t_{chal}$ as the number of blocks allotted to open a commitment, the number of blocks to wait, and the number of blocks allotted to challenge a permission respectively
  3. Set $\mathsf{perm\text{-}info} = (\mathsf{vk}_{\mathsf{sim}}, t_{open}, t_{wait}, t_{chal}, \mathsf{vk}_{\mathsf{S}})$ and send $(\texttt{client perm-info}, \mathsf{C}, \mathsf{perm\text{-}info})$
  4. Query $\mathsf{OLedger}$ with $(\mathsf{perm\text{-}info})$, receive $\mathsf{tx}_{\mathsf{C}}, \sigma_{\mathsf{C}}$, store in L_{cid}, and send $\mathsf{tx}_{\mathsf{C}}, \sigma_{\mathsf{C}}$ to S^*
  5. Upon query $(\mathsf{tx}_{\mathsf{C}}.\mathsf{data}\|\sigma_{sig})$ to $\mathsf{OLedger}$ by S^*, if $\Sigma.\mathsf{Vf}(\mathsf{vk}_{\mathsf{S}}, \sigma_{sig}, \mathsf{tx}_{\mathsf{C}}.\mathsf{data}) \neq 1$ abort, else forward the query to receive $(\mathsf{tx}_{reg}, \sigma_{reg})$ and store in L_{cid}

Figure 12: Simulation of the Register Procedure of Π_{Perm} for a Malicious Server S^*

UC-Security of Credential-less Permission Protocol. We give our main theorem, Theorem 1, and a UC-proof of security.

Theorem 1. *If $\mathcal{L}$ is a* SUF-AUTH *ledger represented as an oracle* $\mathsf{OLedger}$*,* $\texttt{COM} = (\mathsf{Commit}, \mathsf{Open})$ *is a statistically hiding, computationally binding commitment scheme, and* $\Sigma = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vf})$ *is a* EUF-CMA *signature scheme, then* Π_{Perm} *realizes the ideal functionality* $\mathcal{G}_{Perm}$

Case: Malicious Server First we consider the case of a malicious server. To prove security in this case, we present the simulator $\mathsf{Sim}_{\mathsf{S}^*}$ (Fig. 12, 13, 14) that generates the view for a malicious S^* in the ideal world. We then prove, through a series of hybrids, that the view generated by this simulator is indistinguishable from the view of a malicious server in the real world executing Π_{Perm}.

Proof by Hybrids

We prove that the view simulated by $\mathsf{Sim}_{\mathsf{S}^*}$ is indistinguishable from a view of the adversary in the real world through a series of hybrids, starting from the real world protocol and moving step-by-step until we reach the ideal world. By proving that each hybrid is indistinguishable from the last, we will prove that the real and ideal world are indistinguishable to a malicious S^*.

- $\mathbf{Hyb}_0$: The real world protocol
- $\mathbf{Hyb}_1$: This is the same as $\mathbf{Hyb}_0$, except that $\mathsf{Sim}_{\mathsf{S}^*}$ aborts with **ForgeFail** when S^* submits forget blocks as permissions
- $\mathbf{Hyb}_2$: This is the same as $\mathbf{Hyb}_1$ except that $\mathsf{Sim}_{\mathsf{S}^*}$ aborts with **SigForge** when S^* submits a signature on behalf of an honest client
- $\mathbf{Hyb}_3$: This is the same as $\mathbf{Hyb}_2$ except that $\mathsf{Sim}_{\mathsf{S}^*}$ aborts with **CommitFail** when there are two valid requests for the same $\mathsf{perm\text{-}info}$ in the permissions

Lemma 1. *If* $\mathsf{OLedger}$ *is realized as a* SUF-AUTH *secure ledger $\mathcal{L}$,* $\mathbf{Hyb}_0$ *is indistinguishable from* $\mathbf{Hyb}_1$

Proof. Note that the concept of unforgeability here is the same as the concept of unforgeability of signatures. That is, the adversary wins if they are able to produce a pair $(\mathsf{tx}^*, \sigma^*)$ that verifies, but was not the result of a call $\mathcal{L}.\mathsf{Post}$.

Towards a contradiction, assume that there exists a PPT adversary $\mathcal{A}$ such that $|Pr[\mathcal{A}(\mathbf{Hyb}_1) = 1] - Pr[\mathcal{A}(\mathbf{Hyb}_0) = 1]| > \mathsf{negl}(\lambda)$. The only difference between these two hybrids is that in $\mathbf{Hyb}_1$, $\mathsf{Sim}_{\mathsf{S}^*}$ aborts if S^* submits permissions that include blocks that were not the result of a query to $\mathsf{OLedger}$. The only way $\mathcal{A}$ could distinguish between these two hybrids is if they can produce blocks $(\mathsf{tx}^*, \sigma^*)$ that verify but were not posted on the ledger.

Therefore, we can use $\mathcal{A}$ to construct a reduction $\mathcal{D}$ such that $\mathcal{D}$ can win the unforgeability game for a $\mathsf{SUF\text{-}AUTH}$ ledger. We define $\mathcal{D}$ in Fig. 15.

Simulator: $\mathsf{Sim}_{\mathsf{S}^*}$ **- Generate Permission Honest Request**

- **Generate Permission**: Upon receipt of ($\texttt{permission request}$, perm-info) from $\mathcal{G}_{Perm}$
  1. Upon receipt of (res, perm-info, S^*, req, t_{elapse}) from $\mathcal{G}_{Perm}$
  2. Query OLedger with com, where $(\texttt{com}, \texttt{open}) = \mathsf{Commit}(\text{perm-info}\|\text{req}\|\mathsf{tx}_{\mathsf{C}})$, receive $\mathsf{tx}_{com}, \sigma_{com}$ and store in L_{cid}
  3. After time t_{elapse}, query OLedger with open, and store $\mathsf{tx}_{open}, \sigma_{open}$ in L_{cid}
  4. If $\text{res} = \texttt{acceptedf}$, wait t_{wait} blocks then
     (a) Compute $\sigma_{granted} = \Sigma.\mathsf{Sign}(\mathsf{sk}_{\mathsf{sim}}, \text{“}accepted\text{”}\|\mathsf{tx}_{open}.\mathsf{data})$, query OLedger with $\text{“}accepted\text{”}\|\mathsf{tx}_{open}.\mathsf{data}\|\sigma_{granted}$, receive $\mathsf{tx}_{accept}, \sigma_{accept}$ and store in L_{cid}
     (b) Upon query $\mathsf{tx}_{reg}.\mathsf{data}\|\mathsf{tx}_{reg}.\mathsf{idx}\|\mathsf{open}\|\mathsf{tx}_{com_1}.\mathsf{idx}\|\mathsf{tx}_{chal_1}.\mathsf{idx}$ to OLedger by S^*, use L_{cid} to construct perm
        i. If $(\mathsf{tx}_{accept}, \sigma_{accept}) \notin \mathsf{chalwindow}_{\mathsf{req}}$ abort with **ForgeFail**
        ii. If there exists $(\mathsf{tx}_{denied}, \sigma_{denied}) \in \mathsf{chalwindow}_{\mathsf{req}}$ with $\sigma_{refuse} \in \mathsf{tx}_{denied}.\mathsf{data}$ and $\Sigma.\mathsf{Vf}(\mathsf{vk}_{\mathsf{sim}}, \sigma_{refuse}, \text{“}denied\text{”}\|\mathsf{tx}_{open}.\mathsf{data}) = 1$ abort with **SigForge**
  5. Else if $\text{res} = \texttt{denied}$, wait t_{wait} blocks then
     (a) Compute $\sigma_{refuse} = \Sigma.\mathsf{Sign}(\mathsf{sk}_{\mathsf{sim}}, \text{“}denied\text{”}\|\mathsf{tx}_{open}.\mathsf{data})$, query OLedger with $\text{“}denied\text{”}\|\mathsf{tx}_{open}.\mathsf{data}\|\sigma_{refuse}$, receive $\mathsf{tx}_{denied}, \sigma_{denied}$ and store in L_{cid}
     (b) Upon query $\mathsf{tx}_{reg}.\mathsf{data}\|\mathsf{tx}_{reg}.\mathsf{idx}\|\mathsf{open}\|\mathsf{tx}_{com_1}.\mathsf{idx}\|\mathsf{tx}_{chal_1}.\mathsf{idx}$ to OLedger by S^*, use L_{cid} to construct perm
        i. If $(\mathsf{tx}_{denied}, \sigma_{denied}) \notin \mathsf{chalwindow}$ abort with **ForgeFail**
        ii. If there exists $(\mathsf{tx}_{accept}, \sigma_{accept}) \in \mathsf{chalwindow}_{\mathsf{req}}$ with $\sigma_{granted} \in \mathsf{tx}_{accept}.\mathsf{data}$ and $\Sigma.\mathsf{Vf}(\mathsf{vk}_{\mathsf{sim}}, \sigma_{granted}, \text{“}accepted\text{”}\|\mathsf{tx}_{open}.\mathsf{data}) = 1$ abort with **SigForge**
  6. Else
     (a) Upon query $\mathsf{tx}_{reg}.\mathsf{data}\|\mathsf{tx}_{reg}.\mathsf{idx}\|\mathsf{open}\|\mathsf{tx}_{com_1}.\mathsf{idx}\|\mathsf{tx}_{chal_1}.\mathsf{idx}$ to OLedger by S^*, use L_{cid} to construct perm
        i. If there exists $(\mathsf{tx}_{accept}, \sigma_{accept}) \in \mathsf{chalwindow}_{\mathsf{req}}$ with $\sigma_{granted} \in \mathsf{tx}_{accept}.\mathsf{data}$ and $\Sigma.\mathsf{Vf}(\mathsf{vk}_{\mathsf{sim}}, \sigma_{granted}, \text{“}accepted\text{”}\|\mathsf{tx}_{open}.\mathsf{data}) = 1$ or there exists $(\mathsf{tx}_{denied}, \sigma_{denied}) \in \mathsf{chalwindow}_{\mathsf{req}}$ with $\sigma_{refuse} \in \mathsf{tx}_{denied}.\mathsf{data}$ and $\Sigma.\mathsf{Vf}(\mathsf{vk}_{\mathsf{sim}}, \sigma_{refuse}, \text{“}denied\text{”}\|\mathsf{tx}_{open}.\mathsf{data}) = 1$ abort with **SigForge**
  7. If there exists any pair $(\mathsf{tx}, \sigma) \in \mathsf{comwindow}_{\mathsf{req}}$ or $\mathsf{chalwindow}_{\mathsf{req}}$ such that $(\mathsf{tx}, \sigma) \notin L_{\mathsf{cid}}$, $(\mathsf{tx}_{com}, \sigma_{com}) \notin \mathsf{comwindow}_{\mathsf{req}}$, or $(\mathsf{tx}_{open}, \sigma_{open}) \notin \mathsf{comwindow}_{\mathsf{req}}$ abort with **ForgeFail**
  8. Else if there exists $\mathsf{tx}'_{open} \in \mathsf{chalwindow}_{\mathsf{req}}$ such that there exists $\mathsf{tx}'_{com} \in \mathsf{comwindow}_{\mathsf{req}}$ where $\mathsf{tx}'_{open}.\mathsf{data}$ is a valid opening of $\mathsf{tx}'_{com}.\mathsf{data}$ for a request for the same perm-info, and the distance between tx'_{open} and the commitment it opens is less than or equal to t_{open} abort with **CommitFail**
  9. Else forward the query to OLedger, receive $(\mathsf{tx}_{fin}, \sigma_{fin})$ and store in L_{cid} and store in L_{cid} and send perm to $\mathcal{G}_{Perm}$

Figure 13: Simulation of an Honest Request to Generate Permission in Π_{Perm} for a Malicious Server S^* and Party P^*

Simulator: $\mathsf{Sim}_{\mathsf{S}^*}$ **- Generate Permissions Malicious Request**

- **Generate Permission:** Upon query com to OLedger by P^*
  1. Forward com to OLedger, receive $(\mathsf{tx}_{com}, \sigma_{com})$, and store in L_{cid}
  2. Upon query open = perm-info‖req‖S‖tx_{C}; r to OLedger by P^*, forward open to OLedger, receive $(\mathsf{tx}_{open}, \sigma_{open})$ and store in L_{cid}
  3. Send (generate permissions, perm-info, S^*, req) to $\mathcal{G}_{Perm}$
  4. If res = accepted, wait t_{wait} blocks then
     (a) Compute $\sigma_{granted} = \Sigma.\mathsf{Sign}(\mathsf{sk}_{\mathsf{sim}}, \text{“}accepted\text{”}\|\mathsf{tx}_{open}.\mathsf{data})$, query OLedger with “*accepted*”‖tx_{open}.data‖$\sigma_{granted}$, receive $\mathsf{tx}_{accept}, \sigma_{accept}$ and store in L_{cid}
     (b) Upon query tx_{reg}.data‖tx_{reg}.idx‖open‖tx_{com_1}.idx‖tx_{chal_1}.idx to OLedger by S^*, use L_{cid} to construct perm
        i. If $(\mathsf{tx}_{accept}, \sigma_{accept}) \notin \mathsf{chalwindow}_{\mathsf{req}}$ abort with **ForgeFail**
        ii. If there exists $(\mathsf{tx}_{denied}, \sigma_{denied}) \in \mathsf{chalwindow}_{\mathsf{req}}$ with $\sigma_{refuse} \in \mathsf{tx}_{denied}$.data and $\Sigma.\mathsf{Vf}(\mathsf{vk}_{\mathsf{sim}}, \sigma_{refuse}, \text{“}denied\text{”}\|\mathsf{tx}_{open}.\mathsf{data}) = 1$ abort with **SigForge**
  5. Else if res = denied, wait t_{wait} blocks then
     (a) Compute $\sigma_{refuse} = \Sigma.\mathsf{Sign}(\mathsf{sk}_{\mathsf{sim}}, \text{“}denied\text{”}\|\mathsf{tx}_{open}.\mathsf{data})$, query OLedger with “*denied*”‖tx_{open}.data‖σ_{refuse}, receive $\mathsf{tx}_{denied}, \sigma_{denied}$ and store in L_{cid}
     (b) Upon query tx_{reg}.data‖tx_{reg}.idx‖open‖tx_{com_1}.idx‖tx_{chal_1}.idx to OLedger by S^*, use L_{cid} to construct perm
        i. If $(\mathsf{tx}_{denied}, \sigma_{denied}) \notin \mathsf{chalwindow}_{\mathsf{req}}$ abort with **ForgeFail**
        ii. If there exists $(\mathsf{tx}_{accept}, \sigma_{accept}) \in \mathsf{chalwindow}_{\mathsf{req}}$ with $\sigma_{granted} \in \mathsf{tx}_{accept}$.data and $\Sigma.\mathsf{Vf}(\mathsf{vk}_{\mathsf{sim}}, \sigma_{granted}, \text{“}accepted\text{”}\|\mathsf{tx}_{open}.\mathsf{data}) = 1$ abort with **SigForge**
  6. Else
     (a) Upon query tx_{reg}.data‖tx_{reg}.idx‖open‖tx_{com_1}.idx‖tx_{chal_1}.idx to OLedger by S^*, use L_{cid} to construct perm
        i. If there exists $(\mathsf{tx}_{accept}, \sigma_{accept}) \in \mathsf{chalwindow}_{\mathsf{req}}$ with $\sigma_{granted} \in \mathsf{tx}_{accept}$.data and $\Sigma.\mathsf{Vf}(\mathsf{vk}_{\mathsf{sim}}, \sigma_{granted}, \text{“}accepted\text{”}\|\mathsf{tx}_{open}.\mathsf{data}) = 1$ or there exists $(\mathsf{tx}_{denied}, \sigma_{denied}) \in \mathsf{chalwindow}_{\mathsf{req}}$ with $\sigma_{refuse} \in \mathsf{tx}_{denied}$.data and $\Sigma.\mathsf{Vf}(\mathsf{vk}_{\mathsf{sim}}, \sigma_{refuse}, \text{“}denied\text{”}\|\mathsf{tx}_{open}.\mathsf{data}) = 1$ abort with **SigForge**
  7. If there exists any pair $(\mathsf{tx}, \sigma) \in \mathsf{comwindow}_{\mathsf{req}}$ or $\mathsf{chalwindow}_{\mathsf{req}}$ such that $(\mathsf{tx}, \sigma) \notin L_{\mathsf{cid}}$, $(\mathsf{tx}_{com}, \sigma_{com}) \notin \mathsf{comwindow}_{\mathsf{req}}$, or $(\mathsf{tx}_{open}, \sigma_{open}) \notin \mathsf{comwindow}_{\mathsf{req}}$ abort with **ForgeFail**
  8. Else if there exists $\mathsf{tx}'_{open} \in \mathsf{chalwindow}_{\mathsf{req}}$ such that there exists $\mathsf{tx}'_{com} \in \mathsf{comwindow}_{\mathsf{req}}$ where tx'_{open}.data is a valid opening of tx'_{com}.data for a request for the same perm-info, and the distance between tx'_{open} and the commitment it opens is less than or equal to t_{open} abort with **CommitFail**
  9. Else forward the query to OLedger, receive $(\mathsf{tx}_{fin}, \sigma_{fin})$ and store in L_{cid} and store in L_{cid} and send perm to $\mathcal{G}_{Perm}$

Figure 14: Simulation of a Malicious Request to Generate Permission in Π_{Perm} for a Malicious Server S^* and Party P^*

$\mathcal{D}(\mathsf{cid})$:
1. Activate $\mathcal{A}(1^\lambda)$
2. Emulate $\mathbf{Hyb}_0$ for $\mathcal{A}$, posting all queries to $\mathsf{OLedger}$ to $\mathcal{L}$
3. If $\mathcal{A}$ submits a pair $(\mathsf{tx}^*, \sigma^*)$ as a part of the permissions such that $\mathcal{L}.\mathsf{Vf}(\mathsf{tx}^*, \sigma^*) = 1$ but was not a result of a query to $\mathsf{OLedger}$, submit $(\mathsf{tx}^*, \sigma^*)$ to the challenger, else abort

Figure 15: $\mathcal{D}$, the Adversary for the Unforgeability of a $\mathsf{SUF\text{-}AUTH}$ Ledger

Because $\mathcal{A}$ can distinguish between the two hybrids, we know that the pair $(\mathsf{tx}^*, \sigma^*)$ must verify. Therefore we know that $\mathcal{D}$ wins the unforgeability game for a $\mathsf{SUF\text{-}AUTH}$ ledger with the same non-negligible probability that $\mathcal{A}$ has of distinguishing between the two hybrids. □

Lemma 2. *If Σ is a EUF-CMA secure signature scheme, $\mathbf{Hyb}_2$ is indistinguishable from $\mathbf{Hyb}_1$*

Proof. Assume towards a contradiction that there exists an adversary $\mathcal{A}$ such that $|Pr[\mathcal{A}(\mathbf{Hyb}_2) = 1] - Pr[\mathcal{A}(\mathbf{Hyb}_1) = 1]| > \mathsf{negl}(\lambda)$. The only difference between these two hybrids is that the simulator aborts with **SigForge** when there is a signed acceptance or denial transaction that was not computed by the simulator.

Therefore, we can use $\mathcal{A}$ to construct a reduction $\mathcal{D}$ that can win the unforgeability game for a EUF-CMA signature scheme Σ. We define $\mathcal{D}$ in Fig. 16.

$\mathcal{D}(\mathsf{vk})$:
1. Activate $\mathcal{A}(1^\lambda)$
2. Emulate $\mathbf{Hyb}_1$ for $\mathcal{A}$
3. If $\mathcal{A}$ submits permissions containing a signed acceptance $\sigma^* = \sigma_{granted}$ or denial $\sigma^* = \sigma_{denied}$, send σ^* and the message it signs to the challenger, else abort

Figure 16: $\mathcal{D}$, the Adversary for the Unforgeability of Σ

Because $\mathcal{A}$ can distinguish between the two hybrids, we know that the signature σ^* will verify. Therefore we know that $\mathcal{D}$ wins the unforgeability game for a EUF-CMA signature scheme with the same non-negligible probability that $\mathcal{A}$ has of distinguishing between the two hybrids. □

Lemma 3. *If $\mathtt{COM} = (\mathsf{Commit}, \mathsf{Open})$ is a statistically hiding commitment scheme, $\mathbf{Hyb}_3$ is indistinguishable from $\mathbf{Hyb}_2$*

Proof. Because $\mathtt{COM}$ is statistically hiding, we know that an adversary cannot determine the contents of the commitment based on tx_{com}. Therefore, the only way the adversary can post a competing commitment is by guessing which client the request is for. □

Case: Malicious Client Next we consider the case of a malicious client. Towards this, we present the simulator $\mathsf{Sim}_{\mathsf{C}^*}$ (Fig. 17, 18). Note that the case of a malicious client covers the case of a malicious party P. This is because C can perform any procedure that P can and more, and with more knowledge. We do still, however, consider the possibility that the permissions were requested by some malicious party P^*.

Proof by Hybrids

We prove that the view simulated by $\mathsf{Sim}_{\mathsf{C}^*}$ is indistinguishable from a view of the adversary in the real world through a series of hybrids, starting from the real world protocol and moving step-by-step until we reach the ideal world. By proving that each hybrid is indistinguishable from the last, we will prove that the real and ideal world are indistinguishable to a malicious C^*.

Simulator: $\mathsf{Sim}_{\mathsf{C}^*}$ - Register

- **Register - Server:** Upon receipt of $(\texttt{registered}, \mathsf{S})$ from $\mathcal{G}_{Perm}$
  1. Set $\mathsf{vk}_{\mathsf{sim}}, \mathsf{sk}_{\mathsf{sim}} \leftarrow \Sigma.\mathsf{Gen}(1^\lambda)$
  2. Choose cid
  3. Query $\mathsf{OLedger}$ with "$register''\|\mathsf{vk}_{\mathsf{sim}}$, receive $(\mathsf{tx}_{\mathsf{S}}, \sigma_{\mathsf{S}})$ and store in L_{cid}
- **Register - Client:** Upon query $\mathsf{perm\text{-}info} = \mathsf{vk}_{\mathsf{C}}\|t_{open}\|t_{wait}\|t_{chal}\|\mathsf{vk}_{\mathsf{sim}}$ from C^* to $\mathsf{OLedger}$
  1. Forward the query, receive $(\mathsf{tx}_{\mathsf{C}^*}, \sigma_{\mathsf{C}^*})$, store in L_{cid} and receive $(\mathsf{tx}_{\mathsf{C}^*}, \sigma_{\mathsf{C}^*})$ from C^*
  2. Send $(\texttt{register client}, \mathsf{C}^*, \mathsf{S})$ to $\mathcal{G}_{Perm}$, receive $(\texttt{registration request}, \mathsf{C}^*, \mathsf{S})$ from $\mathcal{G}_{Perm}$ and respond with $(\texttt{client perm-info}, \mathsf{C}^*, \mathsf{perm\text{-}info})$
  3. If this is the first registration request from C^*, compute $\sigma_{sig} = \Sigma.\mathsf{Sign}(\mathsf{sk}_{\mathsf{sim}}, \mathsf{tx}_{\mathsf{C}^*}.\mathsf{data})$, query $\mathsf{OLedger}$ with $(\mathsf{tx}_{\mathsf{C}^*}.\mathsf{data}\|\sigma_{sig})$, receive $(\mathsf{tx}_{reg}, \sigma_{reg})$ and store in L_{cid}

Figure 17: Simulation of the Register Procedure of Π_{Perm} for a Malicious Client C^*

Simulator: $\mathsf{Sim}_{\mathsf{C}^*}$ - Generate Permission

- **Generate Permission**: Upon query $\texttt{com}$ to $\mathsf{OLedger}$ by C^* or P^*
  1. Forward $\texttt{com}$ to $\mathsf{OLedger}$, receive $(\mathsf{tx}_{com}, \sigma_{com})$, and store in L_{cid}
  2. Upon query $\texttt{open} = \mathsf{perm\text{-}info}\|\mathsf{req}\|\mathsf{S}\|\mathsf{tx}_{\mathsf{C}^*}; r$ to $\mathsf{OLedger}$ by C^* or P^*, forward $\texttt{open}$ to $\mathsf{OLedger}$, receive $(\mathsf{tx}_{open}, \sigma_{open})$, and store in L_{cid}
  3. Send $(\texttt{generate permission}, \mathsf{perm\text{-}info}, \mathsf{S}, \mathsf{req})$ to $\mathcal{G}_{Perm}$
  4. If C^* queries $\mathsf{OLedger}$ with "$accepted$"$\|\mathsf{tx}_{open}.\mathsf{data}\|\sigma_{granted}$ where $\Sigma.\mathsf{Vf}(\mathsf{vk}_{\mathsf{C}}, \sigma_{granted}, \text{"}accepted\text{"}\|\mathsf{tx}_{open}.\mathsf{data}) = 1$, send $(\texttt{accepted})$ to $\mathcal{G}_{Perm}$
     (a) Forward the query to $\mathsf{OLedger}$, receive $(\mathsf{tx}_{accept}, \sigma_{accept})$, and store in L_{cid}
     (b) Let $\mathsf{comwindow}_{\mathsf{req}}$ be the t_{open} blocks before and $t_{open} + \zeta$ blocks after tx_{open} and $\mathsf{chalwindow}_{\mathsf{req}}$ be the t_{chal} blocks occurring t_{wait} blocks after tx_{open}
  5. Else if C^* queries $\mathsf{OLedger}$ with "$denied$"$\|\mathsf{tx}_{open}.\mathsf{data}\|\sigma_{refuse}$ where $\Sigma.\mathsf{Vf}(\mathsf{vk}_{\mathsf{C}}, \sigma_{refuse}, \text{"}denied\text{"}\|\mathsf{tx}_{open}.\mathsf{data}) = 1$, send $(\texttt{denied})$ to $\mathcal{G}_{Perm}$
     (a) Forward the query to $\mathsf{OLedger}$, receive $(\mathsf{tx}_{denied}, \sigma_{denied})$, and store in L_{cid}
     (b) Let $\mathsf{comwindow}_{\mathsf{req}}$ be the $t_{open} + \zeta$ blocks before and t_{open} blocks after tx_{open} and $\mathsf{chalwindow}_{\mathsf{req}}$ be the t_{chal} blocks occurring t_{wait} blocks after tx_{open}
  6. Else if C^* does nothing, send $(\texttt{silent})$ to $\mathcal{G}_{Perm}$
     (a) If any party queries $\mathsf{OLedger}$ with $\texttt{com}'$ and $\texttt{open}'$ such that $\mathsf{req}' \in \texttt{open}'$ is a request for the same $\mathsf{perm\text{-}info}$, the distance between the queries is less than t_{open}, and $\texttt{com}'$ was queried no more than ζ blocks after $\texttt{open}$ output $\perp$ and abort
     (b) Else if any party queries $\mathsf{OLedger}$ with $\texttt{open}'$ such that $\texttt{open}' \neq \texttt{open}$ is an opening for $\texttt{com}$, abort with **BindingFail**
     (c) Let $\mathsf{comwindow}_{\mathsf{req}}$ be the t_{open} blocks before and $t_{open} + \zeta$ blocks after tx_{open} and $\mathsf{chalwindow}_{\mathsf{req}}$ be the t_{chal} blocks occurring t_{wait} blocks after tx_{open}
  7. Set $\mathsf{perm} = \mathsf{tx}_{\mathsf{S}}\|\sigma_{\mathsf{S}}\|\mathsf{tx}_{\mathsf{C}^*}\|\sigma_{\mathsf{C}^*}\|\mathsf{tx}_{reg}\|\sigma_{reg}\|\texttt{open}\|\mathsf{comwindow}_{req}\|\mathsf{chalwindow}_{req}$ query $\mathsf{OLedger}$ with $\mathsf{tx}_{reg}.\mathsf{data}\|\mathsf{tx}_{reg}.\mathsf{idx}\|\texttt{open}\|\mathsf{tx}_{com_1}.\mathsf{idx}\|\mathsf{tx}_{chal_1}.\mathsf{idx}$, receive $(\mathsf{tx}_{fin}, \sigma_{fin})$, store in L_{cid}, and send perm to $\mathcal{G}_{Perm}$

Figure 18: Simulation of the Generate Permission Procedure of Π_{Perm} for a Malicious Client C^*

- $\mathbf{Hyb}_0$: The real world protocol
- $\mathbf{Hyb}_1$: This is the same as $\mathbf{Hyb}_0$, except that $\mathsf{Sim}_{\mathsf{C}^*}$ aborts with **BindingFail** when a commitment is opened to a different value than the committed value

Lemma 4. *If* $\mathtt{COM} = (\mathsf{Commit}, \mathsf{Open})$ *is a binding commitment scheme,* $\mathbf{Hyb}_1$ *is indistinguishable from* $\mathbf{Hyb}_0$

Proof. Assume towards a contradiction that there exists an adversary $\mathcal{A}$ such that $|Pr[\mathcal{A}(\mathbf{Hyb}_1) = 1] - Pr[\mathcal{A}(\mathbf{Hyb}_0) = 1]| > \mathsf{negl}(\lambda)$. The only difference between these two hybrids is that in $\mathbf{Hyb}_1$, the simulator aborts when a commitment is opened to two different values. Therefore, we can use $\mathcal{A}$ to construct a reduction $\mathcal{D}$ that can break the binding property of $\mathtt{COM}$. We define $\mathcal{D}$ in Fig. 19.

$\mathcal{D}(1^\lambda)$:
1. Activate $\mathcal{A}(1^\lambda)$
2. Emulate $\mathbf{Hyb}_0$ for $\mathcal{A}$
3. If $\mathcal{A}$ provides $\mathtt{open}'$ such that $\mathtt{open}'$ is a second opening for $\mathtt{com}$, send $(\mathtt{com}, \mathtt{open}, \mathtt{open}')$ to the challenger, else abort

Figure 19: $\mathcal{D}$, the Adversary for Binding of $\mathtt{COM}$

Because $\mathcal{A}$ is able to distinguish between $\mathbf{Hyb}_0$ and $\mathbf{Hyb}_1$, we know that $\mathtt{open}$ and $\mathtt{open}'$ will be valid openings with the same probability that $\mathcal{A}$ has of distinguishing between the two hybrids. Therefore $\mathcal{D}$ breaks binding with the same non-negligible probability that $\mathcal{A}$ has of distinguishing between the two hybrids. □

Case: Malicious Server and Client Now we consider the case of a malicious server colluding with a malicious client.

Proof by Hybrids

We prove that the view simulated by $\mathsf{Sim}_{\mathsf{S}^*,\mathsf{C}^*}$ is indistinguishable from a view of the adversary in the real world through a series of hybrids, starting from the real world protocol and moving step-by-step until we reach the ideal world. By proving that each hybrid is indistinguishable from the last, we will prove that the real and ideal world are indistinguishable to a malicious S^* and C^*.

- $\mathbf{Hyb}_0$: The real world protocol
- $\mathbf{Hyb}_1$: This is the same as $\mathbf{Hyb}_0$ except $\mathsf{Sim}_{\mathsf{S}^*,\mathsf{C}^*}$ aborts with **BindingFail** when a commitment is opened to a different value than the committed value
- $\mathbf{Hyb}_2$: This is the same as $\mathbf{Hyb}_1$ except $\mathsf{Sim}_{\mathsf{S}^*,\mathsf{C}^*}$ aborts with **ForgeFail** if there are any forged transactions in the permissions

Simulator: $\mathsf{Sim}_{\mathsf{S}^*,\mathsf{C}^*}$ **- Register**

- **Register - Server:** Upon query $(\text{“}register\text{”} \| \mathsf{vk}_\mathsf{S})$ to the ledger
  1. Send $(\mathtt{register\ server}, \mathsf{S}^*)$ to $\mathcal{G}_{Perm}$
  2. Forward the query to $\mathsf{OLedger}$ and receive $\mathsf{tx}_{\mathsf{S}^*}, \sigma_{\mathsf{S}^*}$, store this in L_{cid} and forward to S^*
- **Register - Client:** Upon query $\mathsf{perm\text{-}info} = \mathsf{vk}_\mathsf{C} \| t_{open} \| t_{wait} \| t_{chal} \| \mathsf{vk}_\mathsf{S}$ from C^* to $\mathsf{OLedger}$
  1. Forward the query, receive $\mathsf{tx}_{\mathsf{C}^*}, \sigma_{\mathsf{C}^*}$ and store in L_{cid}
  2. Send $(\mathtt{register\ client}, \mathsf{C}^*, \mathsf{S}^*)$ to $\mathcal{G}_{Perm}$, receive $(\mathtt{registration\ request}, \mathsf{C}^*, \mathsf{S}^*)$ from $\mathcal{G}_{Perm}$, and respond with $(\mathtt{client\ perm\text{-}info}, \mathsf{C}^*, \mathsf{perm\text{-}info})$
  3. Upon query $(\mathsf{tx}_{\mathsf{C}^*}.\mathsf{data} \| \sigma_{sig})$ to $\mathsf{OLedger}$ by S^*, forward the query to receive $(\mathsf{tx}_{reg}, \sigma_{reg})$ and store in L_{cid}

Figure 20: Simulation of the Register Procedure of Π_{Perm} for a Malicious Server S^* and Client C^*

Simulator: $\mathsf{Sim}_{\mathsf{S}^*,\mathsf{C}^*}$ - Generate Permission

- **Generate Permission:** Upon query com to OLedger by C^* or P^*
  1. Forward com to OLedger, receive $(\mathsf{tx}_{com}, \sigma_{com})$ and store in L_{cid}
  2. Upon query open $=$ perm-info$\|$req$\|\mathsf{S}\|\mathsf{tx}_{\mathsf{C}^*}; r$ to OLedger from C^* or P^* forward open to OLedger, receive $(\mathsf{tx}_{open}, \sigma_{open})$ and store in L_{cid}
  3. Send (generate permissions, perm-info, S^*, req) to $\mathcal{G}_{Perm}$
  4. If C^* queries with $(\text{“}accepted\text{”}\|\mathsf{tx}_{open}.\mathsf{data}\|\sigma_{granted})$ where $\Sigma.\mathsf{Vf}(\mathsf{vk}_{\mathsf{C}}, \sigma_{granted}, \text{“}accepted\text{”}\|\mathsf{tx}_{open}.\mathsf{data}) = 1$, send (accepted) to $\mathcal{G}_{Perm}$
     (a) Forward the query to OLedger and receive $(\mathsf{tx}_{accept}, \sigma_{accept})$
     (b) Upon query $\mathsf{tx}_{reg}.\mathsf{data}\|\mathsf{tx}_{reg}.\mathsf{idx}\|\mathsf{open}\|\mathsf{tx}_{com_1}.\mathsf{idx}\|\mathsf{tx}_{chal_1}.\mathsf{idx}$ to OLedger by S^*, use L_{cid} to construct perm
  5. Else if C^* queries with $(\text{“}denied\text{”}\|\mathsf{tx}_{open}.\mathsf{data}\|\sigma_{refuse})$ where $\Sigma.\mathsf{Vf}(\mathsf{vk}_{\mathsf{C}}, \sigma_{refuse}, \text{“}denied\text{”}\|\mathsf{tx}_{open}.\mathsf{data}) = 1$, send (denied) to $\mathcal{G}_{Perm}$
     (a) Forward the query to OLedger and receive $(\mathsf{tx}_{denied}, \sigma_{denied})$
     (b) Upon query $\mathsf{tx}_{reg}.\mathsf{data}\|\mathsf{tx}_{reg}.\mathsf{idx}\|\mathsf{open}\|\mathsf{tx}_{com_1}.\mathsf{idx}\|\mathsf{tx}_{chal_1}.\mathsf{idx}$ to OLedger by S^*, use L_{cid} to construct perm
  6. Else if C^* does nothing, send (silent) to $\mathcal{G}_{Perm}$
     (a) If any party queries OLedger with com′ and open′ such that req′ $\in$ open′ is a request for the same perm-info, the distance between the queries is less than t_{open}, and com′ was queried no more than ζ blocks after com output $\perp$ and abort
     (b) Else if any party queries OLedger with open′ such that open′ $\neq$ open is an opening for com, abort with **BindingFail**
     (c) Upon query $\mathsf{tx}_{reg}.\mathsf{data}\|\mathsf{tx}_{reg}.\mathsf{idx}\|\mathsf{open}\|\mathsf{tx}_{com_1}.\mathsf{idx}\|\mathsf{tx}_{chal_1}.\mathsf{idx}$ to OLedger by S^*, use L_{cid} to construct perm
  7. If there exists any pair $(\mathsf{tx}, \sigma) \in \mathsf{comwindow}_{\mathsf{req}}$ or $\mathsf{chalwindow}_{\mathsf{req}}$ with $(\mathsf{tx}, \sigma) \notin L_{\mathsf{cid}}$, $(\mathsf{tx}_{com}, \sigma_{com}) \notin \mathsf{comwindow}_{\mathsf{req}}$, or $(\mathsf{tx}_{open}, \sigma_{open}) \notin \mathsf{comwindow}_{\mathsf{req}}$ abort with **ForgeFail**
  8. Else forward the query to OLedger, receive $(\mathsf{tx}_{fin}, \sigma_{fin})$ and store in L_{cid} and store in L_{cid} and send perm to $\mathcal{G}_{Perm}$

Figure 21: Simulation of the Generate Permission Procedure of Π_{Perm} for a Malicious Server S^*, Client C^*, and Party P^*

Simulator: Sim_H **- Register**

- **Registration - Server:** Upon receipt of $(\texttt{registered}, \mathsf{S})$ from $\mathcal{G}_{Perm}$
  1. Set $\mathsf{vk}_\mathsf{S}, \mathsf{sk}_\mathsf{S} \leftarrow \Sigma.\mathsf{Gen}(1^\lambda)$
  2. Choose cid
  3. Query $\mathsf{OLedger}$ with "$register$"$\|\mathsf{vk}_\mathsf{S}\|\mathsf{cid}$, receive $(\mathsf{tx}_\mathsf{S}, \sigma_\mathsf{S})$ and store in L_cid
- **Registration - Client:** Upon receipt of $(\texttt{register client}, \mathsf{C}, \mathsf{S})$ from $\mathcal{G}_{Perm}$
  1. Set $(\mathsf{vk}_\mathsf{C}, \mathsf{sk}_\mathsf{C}) \leftarrow \Sigma.\mathsf{Gen}(1^\lambda)$
  2. Choose $t_{open}, t_{wait}, t_{chal}$ as the number of blocks allotted to open a commitment, the number of blocks to wait, and the number of blocks allotted to challenge a permission respectively
  3. Set $\mathsf{perm\text{-}info} = (\mathsf{vk}_\mathsf{C}, t_{open}, t_{wait}, t_{chal}, \mathsf{vk}_\mathsf{S})$ and send $(\texttt{client perm-info}, \mathsf{C}, \mathsf{perm\text{-}info})$
  4. Query $\mathsf{OLedger}$ with $\mathsf{perm\text{-}info}$, receive $\mathsf{tx}_\mathsf{C}, \sigma_\mathsf{C}$, store in L_cid
  5. Compute $\sigma_{sig} = \Sigma.\mathsf{Sign}(\mathsf{sk}_\mathsf{S}, \mathsf{tx}_\mathsf{C}.\mathsf{data})$, query $\mathsf{OLedger}$ with $(\mathsf{tx}_\mathsf{C}.\mathsf{data}\|\sigma_{sig})$, receive $(\mathsf{tx}_{reg}, \sigma_{reg})$ and store in L_cid

Figure 22: Simulation of the Registration Procedure of Π_{Perm} for Honest Client C, Server S, and Party P

Lemma 5. *If* $\mathtt{COM} = (\mathsf{Commit}, \mathsf{Open})$ *is a binding commitment scheme,* $\mathbf{Hyb}_1$ *is indistinguishable from* $\mathbf{Hyb}_0$

Proof. Follows from the proof of Lemma 4. □

Lemma 6. *If* $\mathsf{OLedger}$ *is realized by a* $\mathsf{SUF\text{-}AUTH}$ *secure ledger* $\mathcal{L}$*,* $\mathbf{Hyb}_2$ *is indistinguishable from* $\mathbf{Hyb}_1$

Proof. Follows from the proof of Lemma 1. □

Case: All Honest Parties Finally, we consider the case where all parties are honest, and show the simulator Sim_H for this case.

Lemma 7. *The view generated by* Sim_H *is indistinguishable from the view generated by honest parties in the real world*

Proof. Sim_H honestly generates signing keys on behalf of S and C, honestly signs the correct messages based on the commands of $\mathcal{Z}$, honestly commits to the request and posts the opening, posts the correct information to the ledger, and honestly generates the permission. Therefore, the two views are statistically indistinguishable. □

Verifying Permission Next we prove that verification of permission in the real world is indistinguishable from verification of permission in the ideal world using the predicate $\mathsf{VerifyPerm}$ by presenting the simulator Sim.

Lemma 8. *The probability that* Sim_{vf} *aborts with* **VerifyFail** *is* $\mathsf{negl}(\lambda)$

Proof. Sim_{vf} only aborts with **VerifyFail** in two cases:

- **Case:** $\mathsf{res} = \texttt{accepted}$ **and** $b_{ver} = 0$
  In this case, $\mathcal{G}_{Perm}$ accepts the permission while $\mathsf{VerifyPerm}$ rejects the permission. $\mathcal{G}_{Perm}$ will accept permission only if the permission was generated through $\mathcal{G}_{Perm}$ and accepted by the client, or if the permission is generated through silence and $\mathsf{VerifyPerm}$ outputs 1. In the case of silence permission, $\mathcal{G}_{Perm}$ defers to $\mathsf{VerifyPerm}$, therefore the probability that the output differs is 0. In the case of accepted permission, an adversary would need to either forge ledger blocks to remove the acceptance transaction, or forge a denial signature on behalf of the client, which we have proved happens with negligible probability in the proofs of Lemmas 1 and 2 respectively.

Simulator: Sim_H **- Generate Permission**

- **Generate Permission** Upon receipt of ($\texttt{permission request}, \mathsf{perm\text{-}info}$) from $\mathcal{G}_{Perm}$
  1. Upon receipt of ($\mathsf{res}, \mathsf{perm\text{-}info}, \mathsf{S}, \mathsf{req}, t_{elapse}$) from $\mathcal{G}_{Perm}$
  2. Query $\mathsf{OLedger}$ with $\texttt{com}$ where $(\texttt{com}, \texttt{open}) = \mathsf{Commit}(\mathsf{perm\text{-}info} \| \mathsf{req} \| \mathsf{S} \| \mathsf{tx_C})$, receive $\mathsf{tx}_{com}, \sigma_{com}$, and store in L_{cid}
  3. After time t_{elapse}, query $\mathsf{OLedger}$ with ($\texttt{open}$), and store $\mathsf{tx}_{open}, \sigma_{open}$ in L_{cid}
  4. If $\mathsf{res} = \texttt{accepted}$
     (a) Compute $\sigma_{granted} = \Sigma.\mathsf{Sign}(\mathsf{sk_C}, \text{“}accepted\text{”} \| \mathsf{tx}_{open}.\mathsf{data})$, query $\mathsf{OLedger}$ with “$accepted$”$\|\mathsf{tx}_{open}.\mathsf{data}\|\sigma_{granted}$, receive $\mathsf{tx}_{accepted}, \sigma_{accepted}$, and store in L_{cid}
     (b) Let $\mathsf{comwindow_{req}}$ be the t_{open} blocks before and $t_{open} + \zeta$ blocks after tx_{open} and $\mathsf{chalwindow_{req}}$ be the t_{chal} blocks occurring t_{wait} blocks after tx_{open}
  5. If $\mathsf{res} = \texttt{denied}$
     (a) Compute $\sigma_{refuse} = \Sigma.\mathsf{Sign}(\mathsf{sk_C}, \text{“}denied\text{”} \| \mathsf{tx}_{open}.\mathsf{data})$, query $\mathsf{OLedger}$ with “$denied$”$\|\mathsf{tx}_{open}.\mathsf{data}\|\sigma_{refuse}$, receive $\mathsf{tx}_{denied}, \sigma_{denied}$, and store in L_{cid}
     (b) Let $\mathsf{comwindow_{req}}$ be the t_{open} blocks before and $t_{open} + \zeta$ blocks after tx_{open} and $\mathsf{chalwindow_{req}}$ be the t_{chal} blocks occurring t_{wait} blocks after tx_{open}
  6. Else
     (a) Let $\mathsf{comwindow_{req}}$ be the t_{open} blocks before and $t_{open} + \zeta$ blocks after tx_{open} and $\mathsf{chalwindow_{req}}$ be the t_{chal} blocks occurring t_{wait} blocks after tx_{open}
  7. Set $\mathsf{perm} = \mathsf{tx_S}\|\sigma_{\mathsf{S}}\|\mathsf{tx_C}\|\sigma_{\mathsf{C}}\|\mathsf{tx}_{reg}\|\sigma_{reg}\|\mathsf{open}\|\mathsf{comwindow_{req}}\|\mathsf{chalwindow_{req}}\|\mathsf{vk_S}$, query $\mathsf{OLedger}$ with $\mathsf{tx}_{reg}.\mathsf{data}\|\mathsf{tx}_{reg}.\mathsf{idx}\|\mathsf{open}\|\mathsf{tx}_{com_1}.\mathsf{idx}\|\mathsf{tx}_{chal_1}.\mathsf{idx}$, receive $(\mathsf{tx}_{fin}, \sigma_{fin})$, store in L_{cid}, and send perm to $\mathcal{G}_{Perm}$

Figure 23: Simulation of the Generate Permission Procedure of Π_{Perm} for Honest Client C, Server S, and Party P

Simulator: Sim_{vf} **- Verify Permission**

- **Verify Permission:** Upon receipt of ($\mathsf{res}, \mathsf{perm\text{-}info}, \mathsf{S}, \mathsf{req}, \mathsf{perm}$) from $\mathcal{G}_{Perm}$
  1. Run $\mathsf{VerifyPerm}(\mathsf{perm\text{-}info}, \mathsf{S}, \mathsf{req}, \mathsf{perm}) = b_{ver}$
  2. If $\mathsf{res} = \texttt{accepted}$ and $b_{ver} = 0$ or $\mathsf{res} = \texttt{denied}$ or $\texttt{not verified}$ and $b_{ver} = 1$ abort with **VerifyFail**
  3. Else output b_{ver}

Figure 24: Simulation of the Verification of Permission $\mathsf{VerifyPerm}$

Functionality: $\mathcal{F}_{SecRec}$

Participants: The Client, the Cloud, the adversary $\mathcal{A}$, and some party P
Variables: $\mathbb{T}_{\mathcal{F}}$ a table of keys indexed by identity
External Functionalities: $\mathcal{G}_{Perm}$ the ideal permissions functionality
Procedures:

- **Store** Upon receiving (`store`, perm-info, Cloud, s) from Client, send (`storage request`, perm-info) to Cloud and $\mathcal{A}$
  1. If $\mathbb{T}_{\mathcal{F}}$[perm-info] $\neq \perp$, output $\perp$ to Client and send (`existing entry`, perm-info) to Cloud and $\mathcal{A}$
  2. If Cloud is corrupt:
     (a) Upon receipt of (res, perm-info) from Cloud, if res = `denied` send $\perp$ to Client
     (b) Else store $\mathbb{T}_{\mathcal{F}}$[perm-info] $= (s, \perp)$, send (`stored`, perm-info) to Client, Cloud, and $\mathcal{A}$
  3. Else store $\mathbb{T}_{\mathcal{F}}$[perm-info] $= (s, \perp)$, send (`stored`, perm-info) to Client, Cloud, and $\mathcal{A}$
- **Remove** Upon receipt of (`remove`, perm-info) from Client, set $\mathbb{T}_{\mathcal{F}}$[perm-info] $= \perp$ and send (`removed`, perm-info) to Cloud and $\mathcal{A}$
- **Retrieve** Upon receipt of (`retrieve`, perm-info) from Client, if $\mathbb{T}_{\mathcal{F}}$[perm-info] $\neq \perp$, send (`retrieve`, perm-info) to Cloud and $\mathcal{A}$.
  1. If a retrieval request is received from any party other than Client, send (`retrieval denied`, perm-info) to Cloud and $\mathcal{A}$
  2. Else send (s) to Client and (`retrieved`, perm-info) to Cloud and $\mathcal{A}$
- **Recover** Upon receiving (`recover`, perm-info) from a party P, if $\mathbb{T}_{\mathcal{F}}$[perm-info] $\neq \perp$, send (`recover`, perm-info) to Client, Cloud, and $\mathcal{A}$
  1. Upon receiving (`recover`, perm-info, req, perm) from $\mathcal{A}$ send (`verify permission`, perm-info, Cloud, req, perm) to $\mathcal{G}_{Perm}$
  2. Upon receipt of (res, perm-info, Cloud, req‖perm) from $\mathcal{G}_{Perm}$, if res $\neq$ `accepted` output $\perp$ to P
  3. Else Send (s, req‖res‖perm) to P, set $\mathbb{T}_{\mathcal{F}}$[perm-info] $= (s,$ req‖res‖perm$)$, and send (`recovery accepted`, perm-info, req‖res‖perm) to Client, Cloud, $\mathcal{A}$

Figure 25: $\mathcal{F}_{SecRec}$ The Ideal Functionality for Secret Recovery

- **Case:** res = `denied` **and** $b_{ver} = 1$
  In this case, $\mathcal{G}_{Perm}$ denies the permission while VerifyPerm accepts. $\mathcal{G}_{Perm}$ will deny the permission only if the permission was not generated through $\mathcal{G}_{Perm}$, was generated through $\mathcal{G}_{Perm}$ but denied by the client, or was generated through silence and VerifyPerm output 0. Again in the case of silence $\mathcal{G}_{Perm}$ defers to VerifyPerm, therefore the probability that the output differs is 0. If the permission was not generated through $\mathcal{G}_{Perm}$, an adversary must have forged blocks from the ledger, as we have shown that permission generation can be simulated in all cases, and we know that blocks can only be forged with negligible probability in the proof of Lemma 1. In the case of denied permission, an adversary would again have to either forge ledger blocks to remove the denial transaction, or forge an acceptance signature on behalf of the client. We know this happens with negligible probability from the proofs of Lemmas 1 and 2 respectively.

  Therefore, in both cases, the probability that Sim_{vf} aborts with **VerifyFail** is negligible. □

6 Credential-less Secret Recovery

In this section, we present our definition of secret recovery, followed by our protocol realizing our definition and the proof of security.

Definition of Secret Recovery. In Fig. 25 we present the ideal functionality for secret recovery $\mathcal{F}_{SecRec}$. With this functionality we capture a client storing a secret with a cloud, such that this secret can be recovered only using permission obtained through $\mathcal{G}_{Perm}$. The cloud learns nothing about the secret during storage,

Protocol: Π_{SecRec} - Set Up, Store, Manage Permissions, and Remove

- **Set Up**
  - Cloud: Upon receipt of $(\texttt{register server}, \mathsf{Cloud})$ from $\mathcal{Z}$
    1. Send $(\texttt{register server}, \mathsf{Cloud})$ to $\mathcal{G}_{Perm}$
- **Store**
  - Client: Upon receipt of $(\texttt{store}, \mathsf{Client}, s, \mathsf{Cloud})$ from $\mathcal{Z}$
    1. Send $(\texttt{register client}, \mathsf{Client}, \mathsf{Cloud})$ to $\mathcal{G}_{Perm}$ and receive $(\mathsf{perm\text{-}info}, \mathsf{Cloud})$
    2. Set $(\mathsf{vk}_{\mathsf{Client}}, \mathsf{sk}_{\mathsf{Client}}) \leftarrow \Sigma.\mathsf{Gen}(1^\lambda)$, get $\mathsf{mvk} = \mathcal{G}_{att}.getpk()$, sample $a \xleftarrow{\$} \mathbb{Z}_q$, and let $A = g^a$
    3. Send $(A, \mathsf{perm\text{-}info}, \mathsf{vk}_{\mathsf{Client}})$ to Cloud
  - Cloud: Upon receipt of $(A, \mathsf{perm\text{-}info}, \mathsf{vk}_{\mathsf{Client}})$ from Client
    1. Let $\mathsf{eid} = \mathcal{G}_{att}.install(\mathsf{perm\text{-}info}, \mathsf{prog}_{SecRec})$
    2. Let $(\mathsf{eid}, \mathsf{prog}_{SecRec}, B, \mathsf{vk}_{ko}, A, \sigma_{att}) = \mathcal{G}_{att}.resume(\mathsf{eid}, (\text{“store”}, A, \mathsf{vk}_{\mathsf{Client}}))$
    3. Send $(\mathsf{eid}, \mathsf{prog}_{SecRec}, B, \mathsf{vk}_{ko}, A, \sigma_{att})$ to Client
  - Client : Upon receiving $(\mathsf{eid}, \mathsf{prog}_{SecRec}, B, \mathsf{vk}_{\mathsf{Client}}, A, \sigma_{att})$ from Cloud
    1. If $\Sigma_{att}.\mathsf{Vf}(\mathsf{mvk}, \mathsf{eid} \| \mathsf{prog}_{SecRec} \| B \| \mathsf{vk}_{\mathsf{Client}} \| A, \sigma_{att}) \neq 1$, output $\perp$ and abort, else store $\mathsf{retK} = B^a$
    2. Compute $c = \Pi.\mathsf{Enc}(\mathsf{retK}, (\mathsf{perm\text{-}info}, s))$ and send c to Cloud
  - Cloud: Upon receipt of c from Client
    1. Let $(b_{att}, \sigma_{att}) = \mathcal{G}_{att}.resume(\mathsf{eid}, (\text{“verify ciphertext”}, \mathsf{perm\text{-}info}, c))$
    2. If $b_{att} \neq 1$, output $\perp$ and abort
    3. Else set $\mathbb{T}[\mathsf{perm\text{-}info}] = (\mathsf{eid}, \mathsf{vk}_{\mathsf{Client}}, c)$
- **Manage Permissions** (Run continuously by Client upon registration)
  - Client : Upon receipt of any $(\texttt{generate permission}, \mathsf{perm\text{-}info}, \mathsf{Cloud}, \mathsf{req})$ from $\mathcal{G}_{Perm}$ such that Client did not request the permission
    1. If $\mathcal{Z}$ sends $(\texttt{accepted}, \mathsf{req})$, send $(\texttt{accepted})$ to $\mathcal{G}_{Perm}$
    2. Else if $\mathcal{Z}$ sends $(\texttt{denied}, \mathsf{req})$, send $(\texttt{denied})$ to $\mathcal{G}_{Perm}$
    3. Else send $(\texttt{silent})$ to $\mathcal{G}_{Perm}$
- **Remove**
  - Client : Upon receipt of $(\texttt{remove}, \mathsf{perm\text{-}info})$ from $\mathcal{Z}$
    1. Compute $\sigma_{remove} = \Sigma.\mathsf{Sign}(\mathsf{sk}_{\mathsf{Client}}, \text{“remove”} \| \mathsf{perm\text{-}info})$ and send $(\texttt{remove}, \mathsf{perm\text{-}info}, \sigma_{remove})$ to Cloud
  - Cloud : Upon receipt of $(\texttt{remove}, \mathsf{perm\text{-}info})$ from Client
    1. If $\Sigma.\mathsf{Vf}(\mathsf{vk}_{\mathsf{Client}}, \sigma_{remove}, \text{“remove”} \| \mathsf{perm\text{-}info}) \neq 1$ abort
    2. Else let $(\text{“removed”}, \mathsf{perm\text{-}info}, \sigma_{remove}, \sigma_{att}) = \mathcal{G}_{att}.resume(\mathsf{eid}, (\text{“remove”}, \mathsf{perm\text{-}info}, \sigma_{remove})$
    3. Set $\mathbb{T}[\mathsf{perm\text{-}info}] = \perp$ and send $(\text{“removed”}, \mathsf{perm\text{-}info}, \sigma_{att})$ to Client
  - Client : Upon receipt of $(\text{“removed”}, \mathsf{perm\text{-}info}, \sigma_{remove}, \sigma_{att})$ from Cloud
    1. If $\Sigma_{att}.\mathsf{Vf}(\mathsf{mvk}, \sigma_{att}, \text{“removed”} \| \mathsf{perm\text{-}info} \| \sigma_{remove}) \neq 1$ abort

Figure 26: Set Up, Store, Manage Permissions, and Remove Procedures for the Secret Recovery Protocol Π_{SecRec}

retrieval, or recovery. Further, a client is able to request removal from the secret recovery system, at which point the secret will no longer be stored.

Secret Recovery Protocol. In this protocol, we assume that Cloud, the party storing the secret, is fitted with a trusted execution environment (TEE) [47]. We model our TEE as the global functionality $\mathcal{G}_{att}$ [47] (Fig. 8). At a high level, a program (Fig. 28) is installed on the host, Cloud. Cloud runs this program, and any output provided by $\mathcal{G}_{att}$ includes an attestation of the form of an unforgeable signature. For details, see Appx. 4.7.

We present the our protocol Π_{SecRec} (Fig. 26, 27), as well as the program run by $\mathcal{G}_{att}$ (Fig. 28), and provide a sketch of the proof that Π_{SecRec} realizes $\mathcal{F}_{SecRec}$ in the $\mathcal{G}_{Perm}, \mathcal{G}_{att}$-hybrid world.

Protocol: Π_{SecRec} **- Retrieve and Recover**

- **Retrieve**
  - Client : Upon receipt of $(\texttt{retrieve}, \mathsf{perm\text{-}info})$ from $\mathcal{Z}$
    1. Compute $\sigma_{ret} = \Sigma.\mathsf{Sign}(\mathsf{sk}_{\mathsf{Client}}, \texttt{retrieve}\|\mathsf{perm\text{-}info})$
    2. Send $(\texttt{retrieve}, \mathsf{perm\text{-}info}, \sigma_{ret})$ to Cloud
  - Cloud: Upon receipt of $(\texttt{retrieve}, \mathsf{perm\text{-}info})$ from Client
    1. Get $\mathbb{T}[\mathsf{perm\text{-}info}] = (\mathsf{eid}, \mathsf{vk}_{\mathsf{Client}}, c)$
    2. If $\Sigma.\mathsf{Vf}(\mathsf{vk}_{\mathsf{Client}}, \sigma_{ret}, \texttt{retrieve}\|\mathsf{perm\text{-}info}) \neq 1$ abort
    3. Send c to Client
  - Client: Upon receipt of c from Cloud
    1. Compute $(\mathsf{perm\text{-}info}', s) = \Pi.\mathsf{Dec}(\mathsf{retK}, c)$, if $\mathsf{perm\text{-}info}' \neq \mathsf{perm\text{-}info}$ output $\perp$ and abort
- **Recover**
  - Client : Upon receipt of $(\texttt{recover}, \mathsf{perm\text{-}info})$ from $\mathcal{Z}$
    1. Let $(\mathsf{pk}, \mathsf{sk}) \leftarrow \Pi_{pub}.\mathsf{Gen}(1^{\lambda})$ and let $\mathsf{req} =$ "*recover*"$\|\mathsf{pk}$
    2. Send $(\texttt{generate permission}, \mathsf{perm\text{-}info}, \mathsf{Cloud}, \mathsf{req})$ to $\mathcal{G}_{Perm}$
    3. Upon receipt of $(\texttt{permission requested}, \mathsf{perm\text{-}info}, \mathsf{Cloud}, \mathsf{req})$ from $\mathcal{G}_{Perm}$, if the request was not made by Client defer to **Manage Permissions**
    4. If $\mathcal{Z}$ sends $(\texttt{accepted}, \mathsf{req})$ send $(\texttt{accepted})$ to $\mathcal{G}_{Perm}$
    5. Else if $\mathcal{Z}$ sends $(\texttt{denied}, \mathsf{req})$ send $(\texttt{denied})$ to $\mathcal{G}_{Perm}$
    6. Else send $(\texttt{silent})$ to $\mathcal{G}_{Perm}$
    7. Receive $(\texttt{permission}, \mathsf{perm\text{-}info}, \mathsf{Cloud}, \mathsf{req}\|\mathsf{perm})$ from $\mathcal{G}_{Perm}$
  - Cloud : Upon receipt of $(\texttt{permission}, \mathsf{perm\text{-}info}, \mathsf{Cloud}, \mathsf{req}\|\mathsf{perm})$ from $\mathcal{G}_{Perm}$
    1. Let $(\mathsf{eid}, c_{fin}, \sigma_{att}) = \mathcal{G}_{att}.\mathit{resume}(\mathsf{eid}, (\text{"recover"}, \mathsf{pk}, c, \mathsf{req}, \mathsf{perm}))$
    2. Set $\mathbb{T}[\mathsf{perm\text{-}info}] = (\mathsf{eid}, \mathsf{vk}_{\mathsf{Client}}, c_{fin})$
    3. Send $(\mathsf{eid}, c_{fin}, \sigma_{att})$ to Client
  - Client : Upon receipt of $(\mathsf{eid}, c_{fin}, \sigma_{att})$ from Cloud
    1. Get $\mathsf{mvk} = \mathcal{G}_{att}.\mathit{getpk}()$, if $\Sigma_{att}.\mathsf{Vf}(\mathsf{mvk}, \sigma_{att}, \mathsf{outp}) \neq 1$ output $\perp$ and abort
    2. Compute $(\mathsf{perm\text{-}info}', s) = \Pi_{pub}.\mathsf{Dec}(\mathsf{sk}, c_{fin})$
    3. If $\mathsf{perm\text{-}info}' \neq \mathsf{perm\text{-}info}$ output $\perp$ and abort

Figure 27: Retrieve and Recover Procedures for the Secret Recovery Protocol Π_{SecRec}

Program: prog_{SecRec}

- On input("store", $A, \mathsf{vk}_{\mathsf{Client}}$):
  1. Let $b \xleftarrow{\$} \mathbb{Z}_q$
  2. Let $B = g^b$
  3. Store $\mathsf{retK} = A^b$
  4. Output $(\mathsf{eid}, \mathsf{prog}_{SecRec}, B, \mathsf{vk}_{\mathsf{Client}}, A)$
- On input("verify ciphertext", $\mathsf{perm\text{-}info}, c$):
  1. Compute $(\mathsf{perm\text{-}info}', s) = \Pi.\mathsf{Dec}(\mathsf{retK}, c)$
  2. If $\mathsf{perm\text{-}info}' = \mathsf{perm\text{-}info}$ output 1 and store perm-info, else output 0
- On input("remove", $\mathsf{perm\text{-}info}, \sigma_{remove}$)
  1. If $\Sigma.\mathsf{Vf}(\mathsf{vk}_{\mathsf{Client}}, \sigma_{remove}$, "remove"$\|\mathsf{perm\text{-}info})$ $\neq 1$ output $\perp$
  2. Set $\mathsf{retK} = \perp$
  3. Output "removed" $\|\mathsf{perm\text{-}info}\|\sigma_{remove}$
- On input("recover", $\mathsf{pk}, c, \mathsf{req}, \mathsf{perm}$):
  1. If $\mathsf{VerifyPerm}(\mathsf{perm\text{-}info}, \mathsf{Cloud}, \mathsf{req}, \mathsf{perm}) = 0$ or $\mathsf{req} \neq$ "*recover*"$\|\mathsf{pk}$ output $\perp$ and abort
  2. Else $(\mathsf{perm\text{-}info}', s) = \Pi.\mathsf{Dec}(\mathsf{retK}, c)$
  3. If $\mathsf{perm\text{-}info}' \neq \mathsf{perm\text{-}info}$ output $\perp$ and abort
  4. $c_{fin} = \Pi_{pub}.\mathsf{Enc}(\mathsf{pk}, (\mathsf{perm\text{-}info}, s))$ store
  5. Output (eid, c_{fin})

Figure 28: The Program Run by $\mathcal{G}_{att}$ for Secret Recovery

UC-Security of Secret Recovery Protocol. Here we present our second main theorem, Theorem 2.

Theorem 2. *If* $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *is an INT-CTX and IND-CPA secure symmetric key encryption scheme,* $\mathcal{G}_{att}$ *is parameterized by an EUF-CMA signature scheme, the DDH assumption holds in group* G *of prime order* q*,* $\Pi_{pub} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *is an IND-CPA secure public key encryption scheme, and* $\Sigma = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vf})$ *is an EUF-CMA secure signature scheme, then* Π_{SecRec} *realizes the ideal functionality* $\mathcal{F}_{SecRec}$ *in the* $(\mathcal{G}_{att}, \mathcal{G}_{Perm})$ *- hybrid model*

To prove Theorem 2, we consider four cases: a malicious Cloud^*, a malicious Client^*, both malicious Cloud^* and Client^*, and both honest Cloud and Client. We make use of the same techniques of simulating a secure enclave presented by Pass et al. [47]. Specifically, we provide our simulator with a backdoor to the TEE that allows the simulator to obtain signatures on values that are not the true output of the program.

Case: Malicious Cloud^* In Fig. 29 we present the simulator in the case of a malicious Cloud^* and prove through a series of hybrids that the view generated by $\mathsf{Sim}_{\mathsf{Cloud}^*}$ is indistinguishable from the view generated in the real world.

Proof by Hybrids

We prove indistinguishability through a series of hybrids:

- $\mathbf{Hyb}_0$: The real world protocol
- $\mathbf{Hyb}_1$: This is the same as $\mathbf{Hyb}_0$ except that a random key is used for retK instead of the result of the Diffie Hellman Key Exchange
- $\mathbf{Hyb}_2$: This is the same as $\mathbf{Hyb}_1$ except that 0 is stored instead of the actual secret and the simulator aborts with **CTXFail** when the ciphertext retrieved is not the ciphertext that was stored
- $\mathbf{Hyb}_3$: This is the same as $\mathbf{Hyb}_2$ except that the simulator aborts with **AttestFail** if Cloud^* does not make the correct calls to $\mathcal{G}_{att}$
- $\mathbf{Hyb}_4$: This is the same as $\mathbf{Hyb}_3$ except that the simulator aborts with **SigForge** if Cloud^* makes a remove call to $\mathcal{G}_{att}$ using a signature that the simulator did not compute

Simulator: $\mathsf{Sim}_{\mathsf{Cloud}^*}$

- **Set Up:** Upon receipt of $(\texttt{register server}, \mathsf{Cloud}^*)$ from Cloud^* to $\mathcal{G}_{Perm}$
  1. Send $(\texttt{register server}, \mathsf{Cloud}^*)$ to internally simulated $\mathcal{G}_{Perm}$
- **Store:** Upon receipt of $(\texttt{registration request}, \mathsf{Client}, \mathsf{Cloud}^*)$ from $\mathcal{G}_{Perm}$
  1. Run internally simulated $\mathcal{G}_{Perm}$ to obtain $\mathsf{perm\text{-}info}$ and send $(\mathsf{perm\text{-}info}, \mathsf{S})$ to Cloud^*
  2. Upon receipt of $(\texttt{storage request}, \mathsf{perm\text{-}info})$ from $\mathcal{F}_{SecRec}$
  3. Set $(\mathsf{vk}_{\mathsf{Client}}, \mathsf{sk}_{\mathsf{Client}}) \leftarrow \Sigma.\mathsf{Gen}(1^\lambda)$
  4. Sample $a \xleftarrow{\$} \mathbb{Z}_q$ and set $A = g^a$
  5. Send $(A, \mathsf{perm\text{-}info}, \mathsf{vk}_{\mathsf{Client}})$ to Cloud^*
  6. Upon call $\mathcal{G}_{att}.\mathit{install}(\mathsf{perm\text{-}info}, \mathsf{prog}_{SecRec})$ by Cloud^*, run internally simulated $\mathcal{G}_{att}$ and forward response eid to Cloud^*
     (a) If no such call is made, abort with **AttestFail**
  7. Upon call $\mathcal{G}_{att}.\mathit{resume}(\mathsf{eid}, (\text{“store”}, A, \mathsf{vk}_{\mathsf{Client}}))$ by Cloud^*, run internally simulated $\mathcal{G}_{att}$ and forward response $(\mathsf{eid}, \mathsf{prog}_{SecRec}, B, \mathsf{vk}_{\mathsf{Client}}, A, \sigma_{att})$ to Cloud^*
     (a) If no such call is made, abort with **AttestFail**
  8. Upon receipt of $(\mathsf{eid}, \mathsf{prog}_{SecRec}, B, \mathsf{vk}_{\mathsf{Client}}, A, \sigma_{att})$ from Cloud^*, store $\mathsf{retK} \xleftarrow{\$} \mathbb{Z}_q$
  9. Compute $c = \Pi.\mathsf{Enc}(\mathsf{retK}, (\mathsf{perm\text{-}info}, 0, \perp))$ and send c to Cloud^*
  10. Upon call $\mathcal{G}_{att}.\mathit{resume}(\mathsf{eid}, (\text{“verify ciphertext”}, \mathsf{perm\text{-}info}, c))$ by Cloud^*, run internally simulated $\mathcal{G}_{att}$ and use the backdoor to obtain a signature on $b_{att} = 1$ if the correct inputs are supplied and forward response (b_{att}, σ_{att}) to Cloud^*
      (a) If no such call is made, abort with **AttestFail**
- **Remove:** Upon receipt of $(\texttt{removed}, \mathsf{perm\text{-}info})$ from $\mathcal{F}_{SecRec}$
  1. If Cloud^* calls $\mathcal{G}_{att}.\mathit{resume}(\mathsf{eid}, (\text{“remove”}, \mathsf{perm\text{-}info}, \sigma^*_{remove}))$ where σ^*_{remove} was not computed by the simulator, abort with **SigForge**
  2. Compute $\sigma_{remove} = \Sigma.\mathsf{Sign}(\mathsf{sk}_{\mathsf{Client}}, \text{“remove”} \| \mathsf{perm\text{-}info})$
  3. Send $(\texttt{remove}, \mathsf{perm\text{-}info}, \sigma_{remove})$ to Cloud^*
  4. Upon call $\mathcal{G}_{att}.\mathit{resume}(\mathsf{eid}, (\text{“remove”}, \mathsf{perm\text{-}info}, \sigma_{remove}))$ by Cloud^*, run internally simulated $\mathcal{G}_{att}$ to receive output $(\text{“removed”}, \mathsf{perm\text{-}info}, \sigma_{remove}, \sigma_{att})$ and forward the output to Cloud^*
     (a) If no such call is made, abort with **AttestFail**
- **Retrieve:** Upon receipt of $(\texttt{retrieve}, \mathsf{perm\text{-}info})$ from $\mathcal{F}_{SecRec}$
  1. Compute $\sigma_{ret} = \Sigma.\mathsf{Sign}(\mathsf{sk}_{\mathsf{Client}}, \texttt{retrieve} \| \mathsf{perm\text{-}info})$
  2. Send $(\texttt{retrieve}, \mathsf{perm\text{-}info}, \sigma_{ret})$ to Cloud^*
  3. Receive ciphertext c'
  4. If $c' \neq c$ abort with **CTXFail**
  5. Else compute $\Pi.\mathsf{Dec}(\mathsf{retK}, c) \neq (\mathsf{perm\text{-}info}', 0, \mathsf{req} \| \mathsf{res} \| \mathsf{perm})$
  6. If $\mathsf{perm\text{-}info}' \neq \mathsf{perm\text{-}info}$ or $\mathsf{req} \| \mathsf{res} \| \mathsf{perm} \neq \perp$ abort
- **Recover:** Upon receipt of $(\texttt{recover}, \mathsf{perm\text{-}info})$ from $\mathcal{F}_{SecRec}$
  1. Let $(\mathsf{pk}, \mathsf{sk}) \leftarrow \Pi_{pub}.\mathsf{Gen}(1^\lambda)$
  2. Let $\mathsf{req} = \text{“}recover\text{”} \| \mathsf{pk}$
  3. Send $(\texttt{generate permission}, \mathsf{perm\text{-}info}, \mathsf{S}, \mathsf{req})$ to internally simulated $\mathcal{G}_{Perm}$ and receive perm
  4. Send $(\texttt{recover}, \mathsf{perm\text{-}info}, \mathsf{req}, \mathsf{perm})$ to $\mathcal{F}_{SecRec}$
  5. Send perm to Cloud^*
  6. Upon call $\mathcal{G}_{att}.\mathit{resume}(\mathsf{eid}, (\text{“recover”}, \mathsf{pk}, c, \mathsf{req}, \mathsf{perm}))$ by Cloud^*, run internally simulated $\mathcal{G}_{att}$ and forward response $(\mathsf{eid} \| c_{fin}, \sigma_{att})$ where $c_{fin} = \Pi_{pub}.\mathsf{Enc}(\mathsf{pk}, 0)$, and σ_{att} is obtained via backdoor if perm is the correct permissions to Cloud^*
     (a) If no such call is made, abort with **AttestFail**
  7. Receive $(\mathsf{eid} \| c_{fin}, \sigma_{att})$ from Cloud^*

Figure 29: Simulation of Π_{SecRec} for a Malicious Cloud^*

Lemma 9. *If the DDH assumptions holds in the group G of prime order q,* $\mathbf{Hyb}_1$ *is indistinguishable from* $\mathbf{Hyb}_0$

Proof. Towards a contradiction assume that there exists an adversary $\mathcal{A}$ such that $|Pr[\mathcal{A}(\mathbf{Hyb}_1) = 1] - Pr[\mathcal{A}(\mathbf{Hyb}_0) = 1]| > \mathsf{negl}(\lambda)$. Then we can construct a reduction $\mathcal{D}$ that can distinguish between a random tuple and a DDH tuple with the same non-negligible probability. We define $\mathcal{D}$ in Fig. 30.

$\mathcal{D}(g, A, B, C)$:
1. Activate $\mathcal{A}(1^\lambda)$
2. Emulate $\mathbf{Hyb}_0$ for $\mathcal{A}$ using A in step 2 of the storage procedure, B in step 5 of the storage procedure[11] and C as retK
3. Output whatever $\mathcal{A}$ outputs

Figure 30: $\mathcal{D}$, the Adversary for the Decisional Diffie-Hellman Problem

The only difference between the two hybrids is whether retK is random or g^{ab}. Therefore, if C is random this is exactly $\mathbf{Hyb}_1$ and if $C = g^{ab}$ this is exactly $\mathbf{Hyb}_0$. Therefore $\mathcal{D}$ wins the DDH game with the same non-negligible probability that $\mathcal{A}$ has of distinguishing between the two hybrids. □

Lemma 10. *If* $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *is INT-CTXT and IND-CPA secure and* $\Pi_{pub} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *is IND-CPA secure,* $\mathbf{Hyb}_2$ *is indistinguishable from* $\mathbf{Hyb}_1$

Proof. Towards a contradiction assume that there exists an adversary $\mathcal{A}$ such that $|Pr[\mathcal{A}(\mathbf{Hyb}_2) = 1] - Pr[\mathcal{A}(\mathbf{Hyb}_1) = 1]| > \mathsf{negl}(\lambda)$. The only difference between these two hybrids is that the stored value is 0 and not the secret. Therefore, $\mathcal{A}$ must be able to distinguish between a ciphertext of 0 and an encryption of k under the encryption scheme Π_{pub} or under the encryption scheme Π.

Case: Π_{pub}

First suppose that $\mathcal{A}$ can distinguish between the two hybrids because they can distinguish between an encryption of 0 under Π_{pub} and an encryption of the secret s under Π_{pub}. Then we can construct a reduction $\mathcal{D}_{pub}$ that wins the IND-CPA game against the encryption scheme Π_{pub}. We define $\mathcal{D}_{pub}$ in Fig. 31.

$\mathcal{D}_{pub}(\mathsf{pk})$:
1. Activate $\mathcal{A}(1^\lambda)$
2. Emulate $\mathbf{Hyb}_1$ for $\mathcal{A}$
3. Upon recovery, query the challenger with $m_0 = (\mathsf{perm\text{-}info}, 0, \perp)$ and $m_1 = (\mathsf{perm\text{-}info}, s, \perp)$ to receive c^*, and use c^* as c_{fin}[12]
4. Output whatever $\mathcal{A}$ outputs

Figure 31: $\mathcal{D}_{pub}$, the CPA Adversary Against Π_{pub}

If c^* is an encryption of 0, this is exactly what $\mathcal{A}$ would expect to see at this point in $\mathbf{Hyb}_2$ and if c^* is an encryption of k, this is exactly what $\mathcal{A}$ expects to see in $\mathbf{Hyb}_1$. Therefore $\mathcal{D}_{pub}$ wins the CPA game with the same non-negligible probability that $\mathcal{A}$ has of distinguishing between the two hybrids, and we have reached our contradiction.

Case: Π

Now suppose that $\mathcal{A}$ is able to distinguish between the two hybrids by submitting a ciphertext $c' \neq c$. That is, $\mathcal{A}$ is able to produce a new ciphertext that decrypts, else $\mathbf{Hyb}_1$ would abort at the same point. Then we can construct a reduction $\mathcal{D}_{CTXT}$ that can win the INT-CTXT game [18] with the same non-negligible probability. We define $\mathcal{D}_{CTXT}$ in Fig. 32.

$\mathcal{D}(1^\lambda)$:
1. Activate $\mathcal{A}(1^\lambda)$
2. Emulate $\mathbf{Hyb}_1$ for $\mathcal{A}$
3. Upon storage, query the challenger with $(\mathsf{perm\text{-}info}, s, \perp)$ and receive c
4. Upon retrieval, if $\mathcal{A}$ submits a ciphertext $c' \neq c$, submit c' to the challenger
5. Submit "Finalize" to the challenger

Figure 32: $\mathcal{D}_{CTXT}$, the INT-CTXT Adversary against Π

Because $\mathcal{A}$ is able to distinguish between the two hybrids by submitting a ciphertext that is not equal to the ciphertext stored but decrypts, we know that $\mathcal{D}_{CTXT}$ wins the game, also by submitting a ciphertext that was never queried yet decrypts, with the same non-negligible probability that $\mathcal{A}$ has of distinguishing between the two hybrids, and have reached our contradiction.

Suppose instead that $\mathcal{A}$ distinguishes between the two hybrids by distinguishing between an encryption of 0 under Π and encryption of the secret s under Π. Then we can construct a reduction $\mathcal{D}_{CPA}$ that can win the IND-CPA game [18] with the same non-negligible probability. We define $\mathcal{D}_{CPA}$ in Fig. 33.

$\mathcal{D}_{CPA}(1^\lambda)$:
1. Activate $\mathcal{A}(1^\lambda)$
2. Emulate $\mathbf{Hyb}_1$ for $\mathcal{A}$
3. Upon storage, query the challenger with $m_0 = (\mathsf{perm\text{-}info}, 0, \perp)$ and $m_1 = (\mathsf{perm\text{-}info}, s, \perp)$ and receive c^*, and use c^* as the ciphertext for storage
4. Output whatever $\mathcal{A}$ outputs

Figure 33: $\mathcal{D}_{CPA}$, the CPA Adversary Against Π

If c^* is an encryption of 0, this is exactly what $\mathcal{A}$ would expect to see at this point in $\mathbf{Hyb}_2$ and if c^* is an encryption of s, this is exactly what $\mathcal{A}$ expects to see in $\mathbf{Hyb}_1$. Therefore $\mathcal{D}_{CPA}$ wins the CPA game with the same non-negligible probability that $\mathcal{A}$ has of distinguishing between the two hybrids, and we have reached our contradiction.

Therefore, in each case, we can construct a reduction that either wins the IND-CPA game against Π_{pub}, the INT-CTXT game against Π, or the CPA game against Π, and have a contradiction. Therefore $\mathbf{Hyb}_2$ is indistinguishable from $\mathbf{Hyb}_1$. □

Lemma 11. *If $\mathcal{G}_{att}$ is parameterized by a EUF-CMA secure signature scheme $\Sigma_{att} = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vf})$, $\mathbf{Hyb}_3$ is indistinguishable from $\mathbf{Hyb}_2$*

Proof. Towards a contradiction assume that there exists an adversary $\mathcal{A}$ such that $|Pr[\mathcal{A}(\mathbf{Hyb}_3) = 1] - Pr[\mathcal{A}(\mathbf{Hyb}_2) = 1]| > \mathsf{negl}(\lambda)$. Then we can construct a reduction $\mathcal{D}$ with the goal of winning the unforgeability game against the signature scheme Σ_{att}. We define $\mathcal{D}$ in Fig. 34.

$\mathcal{D}(\mathsf{vk})$:
1. Activate $\mathcal{A}(1^\lambda)$
2. Emulate $\mathbf{Hyb}_2$ for $\mathcal{A}$
3. Upon submission of a signature σ^* by $\mathcal{A}$ that was not the result of a call to $\mathcal{G}_{att}$, submit σ^* and the message it signs to the challenger

Figure 34: $\mathcal{D}$, the Adversary for the Unforgeability of Σ_{att}

The only difference between the two hybrids is that in $\mathbf{Hyb}_3$ the simulator aborts with **AttestFail** when the adversary does not make the proper calls to $\mathcal{G}_{att}$. Therefore, $\mathcal{A}$ must be able to produce forged signatures that verify, else $\mathbf{Hyb}_2$ would abort at the same point, and we know that $\mathcal{D}$ must then win the unforgeability game with the same non-negligible probability. □

Lemma 12. *If* $\Sigma = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vf})$ *is an EUF-CMA secure signature scheme, then* $\mathbf{Hyb}_4$ *is indistinguishable from* $\mathbf{Hyb}_3$

Proof. Towards a contradiction assume that there exists an adversary $\mathcal{A}$ such that $|Pr[\mathcal{A}(\mathbf{Hyb}_3) = 1] - Pr[\mathcal{A}(\mathbf{Hyb}_2) = 1]| > \mathsf{negl}(\lambda)$. Then we can construct a reduction $\mathcal{D}$ such that $\mathcal{D}$ can win the unforgeability game against the signature scheme Σ with the same non-negligible probability. We define $\mathcal{D}$ in Fig. 35.

$\mathcal{D}(\mathsf{vk})$:
1. Activate $\mathcal{A}(1^\lambda)$
2. Emulate $\mathbf{Hyb}_3$ for $\mathcal{A}$, querying the challenger to compute signatures
3. If $\mathcal{A}$ makes a remove call to $\mathcal{G}_{att}$ using a signature σ^* that is not the result of a query, submit $(\sigma^*, \text{“remove”} \| \mathsf{perm\text{-}info})$ to the challenger

Figure 35: $\mathcal{D}$, the Adversary for the Unforgeability of Σ

Because $\mathcal{A}$ can distinguish between the two hybrids, and the only difference is that in $\mathbf{Hyb}_4$ the simulator aborts when $\mathcal{A}$ submits a signature that was not computed by the simulator, $\mathcal{A}$ must be able to compute forgeries that verify, else $\mathbf{Hyb}_3$ would abort at the same point. Therefore $\mathcal{D}$ wins the unforgeability game against Σ with the same non-negligible probability that $\mathcal{A}$ has of distinguishing between the two hybrids. □

Case: Malicious Client^* In Fig. 36 we present the simulator in the case of a malicious Client^* and prove through a series of hybrids that the view generated by $\mathsf{Sim}_{\mathsf{Client}^*}$ is indistinguishable from the view generated in the real world.

Indistinguishability

Lemma 13. *The view generated by* $\mathsf{Sim}_{\mathsf{Client}^*}$ *is indistinguishable from the view generated by the real world adversary controlling a malicious* Client^*

Proof. $\mathsf{Sim}_{\mathsf{Client}^*}$ behaves as an honest Cloud, and needs no special abort cases. Therefore the view generated by $\mathsf{Sim}_{\mathsf{Client}^*}$ is indistinguishable from the view generated by a real world adversary controlling a malicious Client^*. □

Case: Malicious Cloud^* **and** Client^* In Fig. 37 present the simulator in the case where both Cloud^* and Client^* are malicious and prove through a series of hybrids that the view generated by Sim_{CC^*} is indistinguishable from the view generated in the real world.

Proof

Lemma 14. *The view generated by* Sim_{CC^*} *is indistinguishable from the view generated by the real world adversary*

Proof. Sim_{CC^*} honestly simulates $\mathcal{G}_{att}$ towards Cloud^*, and because we know that a simulator exists for $\mathcal{G}_{Perm}$, is able to generate $\mathsf{perm\text{-}info}$ as expected for Client^*. Therefore the view generated by Sim_{CC^*} is indistinguishable from that of a real world adversary. □

Case: Honest Cloud **and** Client Finally, in Fig. 38 we present the simulator in the case where all parties are honest and prove through a series of hybrids that the view generated by Sim_H is indistinguishable from the view generated in the real world.

Proof by Hybrids

We prove indistinguishability through a series of hybrids:

Simulator: $\mathsf{Sim}_{\mathsf{Client}^*}$

- **Set Up:** Upon receipt of $(\texttt{registered}, \mathsf{Cloud})$ from $\mathcal{G}_{Perm}$
  1. Send $(\texttt{registered}, \mathsf{Cloud})$ to Client^*
- **Store:** Upon receipt of $(\texttt{register client}, \mathsf{Client}^*, \mathsf{Cloud})$ from Client^* intended for $\mathcal{G}_{Perm}$
  1. Forward $(\texttt{register client}, \mathsf{Client}^*, \mathsf{Cloud})$ to internally simulated $\mathcal{G}_{Perm}$ to obtain $\mathsf{perm\text{-}info}$
  2. Upon receipt of $(A, \mathsf{perm\text{-}info}, \mathsf{vk}_{\mathsf{Client}})$ from Client^* run $\mathcal{G}_{att}.\mathit{install}(\mathsf{perm\text{-}info}, \mathsf{prog}_{SecRec}) = \mathsf{eid}$ on internally simulated $\mathcal{G}_{att}$
  3. Run $\mathcal{G}_{att}.\mathit{resume}(\mathsf{eid}, (\text{“store”}, A, \mathsf{vk}_{\mathsf{Client}})) = (\mathsf{eid}, \mathsf{prog}_{SecRec}, B, \mathsf{vk}_{\mathsf{Client}}, A, \sigma_{att})$ on internally simulated $\mathcal{G}_{att}$ and store retK
  4. Send $(\mathsf{eid}, \mathsf{prog}_{SecRec}, B, \mathsf{vk}_{\mathsf{Client}}, A, \sigma_{att})$ to Client^*
  5. Receive c from Client^* and compute $\Pi.\mathsf{Dec}(\mathsf{retK}, c) = (\mathsf{perm\text{-}info}', s, \mathsf{req}\|\mathsf{res}\|\mathsf{perm})$
  6. If $\mathsf{perm\text{-}info}' \neq \mathsf{perm\text{-}info}$ or $\mathsf{req}\|\mathsf{res}\|\mathsf{perm} \neq \perp$ output $\perp$ and abort
  7. Else send $(\texttt{store}, \mathsf{perm\text{-}info}, s)$ to $\mathcal{F}_{SecRec}$ and store $\mathbb{T}_{\mathsf{Sim}}[\mathsf{perm\text{-}info}] = (\mathsf{eid}, c, \perp)$
- **Remove:** Upon receipt of $(\texttt{remove}, \mathsf{perm\text{-}info}, \sigma_{remove})$ from Client^*
  1. If $\Sigma.\mathsf{Vf}(\mathsf{vk}_{\mathsf{Client}}, \sigma_{remove}, \text{“remove”}\|\mathsf{perm\text{-}info}) \neq 1$ abort
  2. Else send $(\texttt{remove}, \mathsf{perm\text{-}info})$ to $\mathcal{F}_{SecRec}$ and receive $(\texttt{removed}, \mathsf{perm\text{-}info})$
  3. Run internally simulated $\mathcal{G}_{att}.\mathit{resume}(\mathsf{eid}, (\text{“remove”}, \mathsf{perm\text{-}info}, \sigma_{remove})) = (\text{“removed”}, \mathsf{perm\text{-}info}, \sigma_{remove}, \sigma_{att})$ and send the output to Client^*
- **Retrieve:** Upon receipt of $(\texttt{retrieve}, \mathsf{perm\text{-}info}, \sigma_{ret})$ from Client^*
  1. If $\Sigma.\mathsf{Vf}(\mathsf{vk}_{\mathsf{Client}}, \sigma_{ret}, \texttt{retrieve}\|\mathsf{perm\text{-}info}) \neq 1$ abort
  2. Else send $(\texttt{retrieve}, \mathsf{perm\text{-}info})$ to $\mathcal{F}_{SecRec}$
  3. If $\mathcal{F}_{SecRec}$ sends $(\texttt{recovered}, \mathsf{perm})$ forward $(\texttt{recovered}, \mathsf{perm})$ to Client^*
  4. Else receive (s) from $\mathcal{F}_{SecRec}$ and send c to Client^*
- **Recover:** Upon receipt of $(\texttt{generate permission}, \mathsf{perm\text{-}info}, \mathsf{Cloud}, \mathsf{req})$ from Client^* intended for $\mathcal{G}_{Perm}$
  1. Send $(\texttt{generate permission}, \mathsf{perm\text{-}info}, \mathsf{Cloud}, \mathsf{req})$ to internally simulated $\mathcal{G}_{Perm}$, receive $(\texttt{permission}, \mathsf{perm\text{-}info}, \mathsf{Cloud}, \mathsf{req}\|\mathsf{res}\|\mathsf{perm})$, and send to Client^*
  2. Send $(\texttt{recover}, \mathsf{perm\text{-}info}, \mathsf{req}, \mathsf{perm})$ to $\mathcal{F}_{SecRec}$
  3. Upon receipt of $\mathsf{req}'\|\mathsf{res}'\|\mathsf{perm}'$ from Client^*, if $\mathsf{req}' \neq \mathsf{req}$, $\mathsf{res}' \neq \mathsf{res}$, or $\mathsf{perm}' \neq \mathsf{perm}$ abort
  4. Run $\mathcal{G}_{att}.\mathit{resume}(\mathsf{eid}, (\text{“recover”}, \mathsf{pk}, c, \mathsf{req}, \mathsf{perm})) = (\mathsf{eid}\|c_{fin}, \sigma_{att})$ on internally simulated $\mathcal{G}_{att}$
  5. Send $(\mathsf{eid}\|c_{fin}, \sigma_{att})$

Figure 36: Simulation of Π_{SecRec} for a Malicious Client^*

Simulator: Sim_{CC^*}

- **Set Up:** Upon receipt of $(\texttt{register server}, \mathsf{Cloud}^*)$ from Cloud^* intended for $\mathcal{G}_{Perm}$
  1. Send $(\texttt{register server}, \mathsf{Cloud}^*)$ to internally simulated $\mathcal{G}_{Perm}$
- **Store:** Upon receipt of $(\texttt{register client}, \mathsf{Client}^*, \mathsf{Cloud}^*)$ from Client^* intended for $\mathcal{G}_{Perm}$
  1. Forward $(\texttt{register client}, \mathsf{Client}^*, \mathsf{Cloud})$ to internally simulated $\mathcal{G}_{Perm}$ to obtain $\mathsf{perm\text{-}info}$, and send $(\mathsf{perm\text{-}info}, \mathsf{Cloud}^*)$ to Client^*
  2. Upon call $\mathcal{G}_{att}.\mathit{install}(\mathsf{perm\text{-}info}, \mathsf{prog}_{SecRec})$ by Cloud^*, run internally simulated $\mathcal{G}_{att}$ and forward response eid to Cloud^*
  3. Upon call $\mathcal{G}_{att}.\mathit{resume}(\mathsf{eid}, (\text{“store”}, A, \mathsf{vk}_{\mathsf{Client}}))$ by Cloud^*, run internally simulated $\mathcal{G}_{att}$ and forward response $(\mathsf{eid}, \mathsf{prog}_{SecRec}, B, \mathsf{vk}_{\mathsf{Client}}, A, \sigma_{att})$ to Cloud^* and store retK
  4. Upon call $\mathcal{G}_{att}.\mathit{resume}(\mathsf{eid}, (\text{“verify ciphertext”}, \mathsf{perm\text{-}info}, c))$ by Cloud^*
     (a) Compute $\Pi.\mathsf{Dec}(\mathsf{retK}, c) = (\mathsf{perm\text{-}info}, s, \mathsf{req}\|\mathsf{res}\|\mathsf{perm})$
     (b) Send $(\texttt{store}, \mathsf{perm\text{-}info}, s)$ to $\mathcal{F}_{SecRec}$
     (c) Run internally simulated $\mathcal{G}_{att}$ and forward response (b_{att}, σ_{att}) to Cloud^*
- **Remove:** Upon call $\mathcal{G}_{att}.\mathit{resume}(\mathsf{eid}, (\text{“remove”}, \mathsf{perm\text{-}info}, \sigma_{remove}))$ by Cloud^*
  1. If $\Sigma.\mathsf{Vf}(\mathsf{vk}_{\mathsf{Client}}, \sigma_{remove}, \text{“remove”}\|\mathsf{perm\text{-}info}) \neq 1$ abort
  2. Send $(\texttt{remove}, \mathsf{perm\text{-}info})$ to $\mathcal{F}_{SecRec}$
  3. Run internally simulated $\mathcal{G}_{att}$ and forward the output $(\text{“removed”}, \mathsf{perm\text{-}info}, \sigma_{remove}, \sigma_{att})$ to Cloud^*
- **Recover:** Upon receipt of $(\texttt{generate permission}, \mathsf{perm\text{-}info}, \mathsf{Cloud}^*, \mathsf{req})$ from Client^* intended for $\mathcal{G}_{Perm}$
  1. Send $(\texttt{generate permission}, \mathsf{perm\text{-}info}, \mathsf{Cloud}^*, \mathsf{req})$ to internally simulated $\mathcal{G}_{Perm}$, receive $(\texttt{permission}, \mathsf{perm\text{-}info}, \mathsf{Cloud}^*, \mathsf{req}\|\mathsf{res}\|\mathsf{perm})$, and send to Client^*
  2. Send $(\texttt{recover}, \mathsf{perm\text{-}info})$ to $\mathcal{F}_{SecRec}$
  3. Send $(\texttt{recover}, \mathsf{perm\text{-}info}, \mathsf{req}, \mathsf{perm})$ to $\mathcal{F}_{SecRec}$
  4. Upon call $\mathcal{G}_{att}.\mathit{resume}(\mathsf{eid}, (\text{“recover”}, \mathsf{pk}, c, \mathsf{req}, \mathsf{perm}))$ by Cloud^* run internally simulated $\mathcal{G}_{att}$ and forward response $(\mathsf{eid}\|c_{fin}, \sigma_{att})$ to Cloud^*

Figure 37: Simulation of Π_{SecRec} for a Malicious Cloud^* and Client^*

Simulator: Sim_H

- **Set Up:** Receive $(\texttt{registered}, \mathsf{Cloud})$ from $\mathcal{G}_{Perm}$
- **Store:** Upon receipt of $(\texttt{registration request}, \mathsf{Client}, \mathsf{Cloud}^*)$ from $\mathcal{G}_{Perm}$
  1. Run internally simulated $\mathcal{G}_{Perm}$ to obtain $\mathsf{perm\text{-}info}$ and send $(\mathsf{perm\text{-}info}, \mathsf{S})$ to Cloud^*
  2. Upon receipt of $(\texttt{storage request}, \mathsf{perm\text{-}info})$ from $\mathcal{F}_{SecRec}$
  3. Set $(\mathsf{vk}_{\mathsf{Client}}, \mathsf{sk}_{\mathsf{Client}}) \leftarrow \Sigma.\mathsf{Gen}(1^\lambda)$
  4. Sample $a \xleftarrow{\$} \mathbb{Z}_q$ and set $A = g^a$
  5. Run internally simulated $\mathcal{G}_{att}$ to receive $\mathsf{eid} = \mathcal{G}_{att}.\mathit{install}(\mathsf{perm\text{-}info}, \mathsf{prog}_{SecRec})$
  6. Run internally simulated $\mathcal{G}_{att}$ to receive $(\mathsf{eid}, \mathsf{prog}_{SecRec}, B, \mathsf{vk}_{\mathsf{Client}}, A\sigma_{att}) = \mathcal{G}_{att}.\mathit{resume}(\mathsf{eid}, (\text{“store”}, A, \mathsf{vk}_{\mathsf{Client}}))$ and store $\mathsf{retK} \xleftarrow{\$} \mathbb{Z}_q$
  7. Compute $c = \Pi.\mathsf{Enc}(\mathsf{retK}, (\mathsf{perm\text{-}info}, 0, \perp))$ and send c to Cloud^*
  8. Run internally simulated $\mathcal{G}_{att}$ to receive $(b_{att}, \sigma_{att}) = \mathcal{G}_{att}.\mathit{resume}(\mathsf{eid}, (\text{“verify ciphertext”}, \mathsf{perm\text{-}info}, c))$ using the backdoor to receive a signature on $b_{att} = 1$
- **Remove:** Upon receipt of $(\texttt{removed}, \mathsf{perm\text{-}info})$ from $\mathcal{F}_{SecRec}$
  1. Compute $\sigma_{remove} = \Sigma.\mathsf{Sign}(\mathsf{sk}_{\mathsf{Client}}, \text{“remove”} \| \mathsf{perm\text{-}info})$
  2. Run internally simulated $\mathcal{G}_{att}$ to receive $(\text{“removed”}, \mathsf{perm\text{-}info}, \sigma_{remove}, \sigma_{att}) = \mathcal{G}_{att}.\mathit{resume}(\mathsf{eid}, (\text{“remove”}, \mathsf{perm\text{-}info}, \sigma_{remove}))$
- **Recover:** Upon receipt of $(\texttt{recover}, \mathsf{perm\text{-}info})$ from $\mathcal{F}_{SecRec}$
  1. Upon receipt of $(\texttt{permission}, \mathsf{perm\text{-}info}, \mathsf{S}, \mathsf{req} \| \mathsf{res} \| \mathsf{perm})$
  2. Parse $\mathsf{req} = \text{“}\mathit{recover}\text{”} \| \mathsf{pk}$
  3. Send $(\texttt{recover}, \mathsf{perm\text{-}info}, \mathsf{req}, \mathsf{perm})$ to $\mathcal{F}_{SecRec}$
  4. Run internally simulated $\mathcal{G}_{att}$ to receive $(\mathsf{eid} \| c_{fin}, \sigma_{att}) = \mathcal{G}_{att}.\mathit{resume}(\mathsf{eid}, (\text{“recover”}, \mathsf{pk}, c, \mathsf{req}, \mathsf{perm}))$ where c_{fin} is an encryption of 0

Figure 38: Simulation of Π_{SecRec} for Honest Cloud and Client

- $\mathbf{Hyb}_0$: The real world protocol
- $\mathbf{Hyb}_1$: This is the same as $\mathbf{Hyb}_0$ except that a random key is used as retK instead of the result of the DHKE
- $\mathbf{Hyb}_2$: This is the same as $\mathbf{Hyb}_1$ except that the value stored upon storage is 0 instead of the secret

Lemma 15. *If the DDH assumption holds in the group G of prime order q, $\mathbf{Hyb}_1$ is indistinguishable from $\mathbf{Hyb}_0$*

Proof. Follows from the proof of Lemma 9 □

Lemma 16. *If $\Pi_{pub} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ is a CPA secure encryption scheme and $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ is a CCA secure encryption scheme, $\mathbf{Hyb}_2$ is indistinguishable from $\mathbf{Hyb}_1$*

Proof. Follows from the proof of Lemma 10 □

Proof Sketch The secret is stored under an INT-CTXT and IND-CPA secure encryption scheme, so we know that a malicious Cloud cannot learn anything about the secret during storage, and cannot modify the ciphertext at any point. Since $\mathcal{G}_{att}$ is parameterized by an EUF-CMA secure signature scheme Σ, we know that the output of $\mathcal{G}_{att}$ cannot be forged, unless the adversary can compute forged signatures that verify. Our Client executes a DHKA with $\mathcal{G}_{att}$ to obtain a shared key. Because we assume that the DDH assumption holds in the group G, the simulator is able to use a random key in place of this shared key, which is indistinguishable from a key computed during the key exchange. The ciphertext returned upon recovery is encrypted under an IND-CPA secure encryption scheme, therefore a malicious Cloud cannot learn anything about the secret upon recovery, and we know that this ciphertext cannot be modified during recovery, as it is the output of $\mathcal{G}_{att}$. Finally, because Client signs removal requests, we know that a malicious Cloud cannot perform a removal without the client's consent.

7 Evaluation

In this section, we provide the results of our experimental evaluation of our protocols. We simulated the Hyperledger fabric blockchain with real-world parameters and implemented Π_{Perm} and Π_{SecRec} using the Python programming language. Our implementation shows that all procedures, aside from **Recover**, are very fast (tens of milliseconds). **Recover** is a slower procedure, with the cloud/enclave side taking about 66 seconds, excluding the waiting period and the latency of the blockchain. However, **Recover** would only be run in the rare case that the client has lost their retrieval key or signing key. See Table 1.

Our implementation only measures the processing time of the client and the cloud/TEE and does not include the time it takes for transactions to be posted. This is because this time depends on the latency of the client and the blockchain network.

We provide our implementation through an anonymous repository `https://anonymous.4open.science/r/SKR-DB10/README.md`

7.1 Implementation

Our implementation used existing libraries from the `pypi` [5] repository and we implemented both Π_{Perm} and Π_{SecRec} as Python scripts. We use symmetric and asymmetric encryption schemes provided by the `pycryptodome` [38] library and digital signatures by `blspy` [2] library . Finally, we used the `oblivious` [46] Python package to implement the Diffie-Helman key exchange and commitment schemes.

Cryptographic Primitives. For symmetric encryption, we used the AES cipher in EAX (encrypt-then-authenticate-then-translate) mode with 256 bits private key size. With this, each ciphertext is associated with a Message Authentication Code (MAC) for authentication. For asymmetric encryption, we used a hybrid approach where we sample a random 2048-bit RSA private key and a 256-bit session key, then encrypt the plaintext under AES with the session key, and finally encrypt the session key with RSA. Here, the AES encryption process is also in EAX mode with an associated MAC. For our digital signatures scheme, we use BLS signatures implemented by the `blspy` library with a private key size of 256 bits.

Hyperledger Simulation. Our protocols are designed to be independent of any specific ledger systems. Π_{Perm} uses the blockchain simply as a public bulletin board. Therefore, we chose to forego deploying the

Hyperledger testbed and opted to simulate the Hyperledger chain instead. This choice further allowed us to run more experiments as the simulation eliminated the need to wait according to t_{wait}.

Our simulation adheres to the real-world parameters of Hyperledger implementations. The structure of transactions in our simulation matches those found in actual Hyperledger transactions [1]. We have configured each transaction in our simulation to require 15 endorsements from a pool of 25 peers before it is submitted to the Ordering Service Nodes (OSN). We have 7 OSNs collaborating to form blocks at 2-second intervals, the default batch timeout time. Each block in our simulation is 2MB, which is the preferred size in the default configuration of Hyperledger[31]. To simulate normal transaction traffic, we have also implemented several parallel processes that generate noise transactions – transactions that do not contribute to our protocol – thereby ensuring continuous activity on our blockchain.

In Sec. 7.3, we discuss how, if an instance of Hyperledger is launched with the specific purpose of supporting secret recovery, Hyperledger can be configured to further reduce the runtime of our protocols.

Protocol Configuration. In our implementation and benchmarks, we intentionally omitted the communication latency between the client and the cloud, as well as the latency incurred by posting to the blockchain, to focus on the compute time and interactions between the cloud and the enclave. Specifically, we included the latency occurring between the cloud and its enclave to better understand the impact of transferring data from the cloud's memory to the enclave's memory on performance.

We configured the enclave as a non-blocking virtual socket server that continuously listens to incoming connections at a designated port. This setup ensures that the server can handle requests asynchronously without delay. We also established a simple socket communication protocol to facilitate interactions between the cloud and the enclave. Each communication cycle begins with transmitting a 64-byte header, which specifies the total length of the data payload that the enclave should expect. Following this, the data transfer proceeds while the socket connection remains open. The connection is only terminated once the cloud confirms the reception of a success or failure response from the enclave. In practical deployment considerations, both the enclave and the cloud could be configured to make such a request asynchronously and thus can continue to execute other processes for other clients.

7.2 Protocol Runtime

Table 1: Performance for Procedures Measured in Seconds, Taken Over 100 Experiments

	Store		**Retrieve**		**Recover**		**Remove**	
	Mean	SD	Mean	SD	Mean	SD	Mean	SD
Client	0.0363	0.0034	0.0011	0.0002	0.5617	0.3332	0.0177	0.0015
Cloud	0.0538	0.0036	0.0198	0.0030	66.2481	1.7835	0.0412	0.0047
Enclave	0.0301	0.0002	0.00	0.00	48.9244	0.1139	0.0015	~ 0.0

Experimental Setup. Our experiments were run on an Amazon EC2 `m5.xlarge` parent VM instance having 4 vCPU, 16GiB of memory, and EBS-only (Elastic Block Store) storage. This machine is the bare minimum specification (specs) required to run the AWS nitro system. We allocated 2 of the 4 vCPU and 4GiB out of 16GiB of RAM to the enclave, leaving the parent with 2 vCPU and 12GiB of memory.

The primary objective of our experiment is to measure the runtime performance of **Register**, **Store**, **Retrieve**, **Recover**, and **Remove** protocols in Π_{SecRec}. For **Register**, we configured the client to have a t_{open} of 30 seconds corresponding to 15 blocks, t_{wait} of 1 week corresponding to 302,400 blocks, and a t_{chal} of 10 minutes corresponding to 300 blocks. For **Store** protocol, we presumed that the secret information that the client prefers to store is a Bitcoin wallet private key with a size of 256 bits. We chose to store Bitcoin private keys due to the prevalence of the problem of recovery in the cryptocurrency space. We ran each experiment 100 times and calculated the mean and standard deviation (SD) of the runtime for each protocol.

Table 1 presents our results. Note that the runtime of **Recover** omits the waiting period as there are no computations performed during the waiting period. Note also that cloud runtime includes the runtime of the enclave.

Store Procedure. The **Store** procedure took 0.0363seconds (s) for the client with an SD of 0.0034s. It took 0.0538s with an SD of 0.0036s for the cloud and 0.0301s and an SD of 0.0002s for the enclave. Here, the cloud runtime includes the cloud's computations, the bidirectional socket communication latency with the enclave, and the enclave's own processing time.

Retrieve Procedure. The **Retrieve** procedure took 0.0011s for the client with an SD of 0.0002s. It took 0.0198s with an SD of 0.0030s for the cloud. The enclave does not run for retrieval.

Recover Procedure. The **Recover** procedure took 0.5617s for the client with an SD of 0.3332s. It took 66.2481s with an SD of 1.7835s for the cloud and 48.9244s and an SD of 0.1139s for the enclave. Like the **Store** protocol above, the runtime of the cloud includes that of the enclave and communication latency. The times provided exclude the waiting period, as no computation is performed during this time. It is important to note that the compute time of **Recover** is *independent of* t_{wait}.

Remove Procedure. Finally, the **Remove** procedure took 0.0177s for the client with an SD of 0.0015s. It took 0.0412s with an SD of 0.0047s for the cloud and 0.0015s and an SD of ~ 0.0 s for the enclave. Again, the cloud runtime includes the that of the enclave and communication latency.

It is essential to mention that the runtimes discussed here for the client and cloud include the latency associated with accessing data from the local ledger. These latencies can vary depending on the proximity of the client and server within the Hyperledger network and the operations of the ledger.

7.3 Deployment Considerations

Our implementation shows that credential-less secret recovery is not only feasible, but fast. Here, we discuss potential improvements to further improve the runtime of the protocols.

The AWS Nitro system supports running multiple enclave instances under one parent instance. In a practical deployment, the cloud can process requests from multiple clients simultaneously, and enclave operations could also be parallelized to handle various clients at once. Specifically, a major bottleneck is in **Recover** during the verification of the blocks comprising comwindow and chalwindow. Block verification consists of verifying the signatures of the OSNs and verifying continuity when given a sequence of blocks. This verification can be parallelized to allow the enclave to parse permission more efficiently and optimize resource utilization, leading to more efficient protocol execution.

Further, in deployment, the default configuration of Hyperledger would not be optimal. Specifically, with the parameters we set, permission consists of blocks totaling over 600MB. This can be reduced by reducing the size of the windows (reducing t_{open} and/or t_{chal}), or by reducing the overall block size. It is important to note that while we suggest reducing the size of t_{open} and/or t_{chal}, it is paramount that these values be set high enough that a client has enough time to post an opening and commitment respectively. An advantage of Hyperledger is that it is extremely configurable, so one can reduce the preferred block size to reduce the overall size of permission. The largest transaction posted in our implementation is tx_{fin}, consisting of about 4206 bits, therefore a block size of 2MB may be unnecessarily large. Reducing the block size is more attainable in the case that an instance of Hyperledger is launched with the sole purpose of supporting Secret Recovery.

Lastly, our implementation can be improved on the client side through the use of smart contracts. Our protocol requires that a client wait for t_{wait} blocks to pass before responding to a malicious request. With smart contracts, the client could instead schedule their denial transaction to be posted within the denial window.

References

[1] https://hyperledger-fabric.readthedocs.io/en/latest/index.html. pages 3, 5, 38

[2] https://pypi.org/project/blspy/. pages 37

[3] Hardware protected signer faqs. https://braavos.notion.site/Hardware-Protected-Signer-FAQs-5d5ae07e999e45ddaf8a7f5c4abbad80. pages 9

[4] How to recover my wallet with guardians: complete guide. https://support.argent.xyz/hc/en-us/articles/360007338877-How-to-recover-my-wallet-with-guardians-complete-guide. pages 9

[5] Python package index - pypi. https://pypi.org/. pages 37

[6] Recovery process for the trezor model t. https://trezor.io/learn/a/recover-wallet-on-model-t. pages 9

[7] Sequence documentation: Key management. https://docs.sequence.xyz/key-management/. pages 9

[8] Set up wallet recovery wizard. https://developers.bitgo.com/guides/wallets/recover. pages 9

[9] Tor project anonymity online. https://www.torproject.org/. pages 4

[10] Torus labs: Open-source key management. https://tor.us/. pages 9

[11] Understand the difference: Zengo wallet recovery vs. ledger recover. https://zengo.com/understand-the-difference-zengo-wallet-recovery-vs-ledger-recover/. pages 9

[12] Amazon Web Services. The security design of the aws nitro system. https://docs.aws.amazon.com/pdfs/whitepapers/latest/security-design-of-aws-nitro-system/security-design-of-aws-nitro-system.pdf, 2022. pages 3, 5

[13] Mehmet Aydar, Salih Cemil Cetin, Serkan Ayvaz, and Betül Aygün. Private key encryption and recovery in blockchain. *CoRR*, abs/1907.04156, 2019. pages 10

[14] Artem Barger, Yacov Manevich, Hagar Meir, and Yoav Tock. A byzantine fault-tolerant consensus library for hyperledger fabric. In *IEEE International Conference on Blockchain and Cryptocurrency, ICBC 2021, Sydney, Australia, May 3-6, 2021*, pages 1–9. IEEE, 2021. pages 5

[15] M. Bellare and S. Goldwasser. Encapsulated key escrow, 1996. pages 2, 9

[16] Mihir Bellare and Shafi Goldwasser. Verifiable partial key escrow. In Richard Graveman, Philippe A. Janson, Clifford Neuman, and Li Gong, editors, *CCS '97, Proceedings of the 4th ACM Conference on Computer and Communications Security, Zurich, Switzerland, April 1-4, 1997*, pages 78–91. ACM, 1997. pages 2, 9

[17] Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In Tatsuaki Okamoto, editor, *Advances in Cryptology - ASIACRYPT 2000, 6th International Conference on the Theory and Application of Cryptology and Information Security, Kyoto, Japan, December 3-7, 2000, Proceedings*, volume 1976 of *Lecture Notes in Computer Science*, pages 531–545. Springer, 2000. pages 10

[18] Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. *J. Cryptol.*, 21(4):469–491, 2008. pages 31, 32

[19] Alysson Neves Bessani, Ricardo Mendes, Tiago Oliveira, Nuno Ferreira Neves, Miguel Correia, Marcelo Pasin, and Paulo Veríssimo. SCFS: A shared cloud-backed file system. In Garth Gibson and Nickolai Zeldovich, editors, *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*, pages 169–180. USENIX Association, 2014. pages 9

[20] Sam Blackshear, Konstantinos Chalkias, Panagiotis Chatzigiannis, Riyaz Faizullabhoy, Irakliy Khaburzaniya, Eleftherios Kokoris-Kogias, Joshua Lind, David Wong, and Tim Zakian. Reactive key-loss protection in blockchains, 2021. pages 4, 6, 8, 9

[21] Vitalik Buterin, Yoav Weiss, Dror Tirosh, Shahaf Nacson, Alex Forshtat, Kristof Gazso, and Tjaden Hess. EIP-4337: Basefee Network Upgrade, 2021. pages 8

[22] Ran Canetti, Kyle Hogan, Aanchal Malhotra, and Mayank Varia. A universally composable treatment of network time. In *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017*, pages 360–375. IEEE Computer Society, 2017. pages 12

[23] Melissa Chase, Hannah Davis, Esha Ghosh, and Kim Laine. Acsesor: A new framework for auditable custodial secret storage and recovery. *IACR Cryptol. ePrint Arch.*, page 1729, 2022. pages 9

[24] Panagiotis Chatzigiannis, Konstantinos Chalkias, Aniket Kate, Easwar Vivek Mangipudi, Mohsen Minaei, and Mainack Mondal. Sok: Web3 recovery mechanisms. *IACR Cryptol. ePrint Arch.*, page 1575, 2023. pages 9

[25] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah M. Johnson, Ari Juels, Andrew Miller, and Dawn Song. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In *IEEE European Symposium on Security and Privacy, EuroS&P 2019, Stockholm, Sweden, June 17-19, 2019*, pages 185–200. IEEE, 2019. pages 2

[26] Ivan Damgård and Eiichiro Fujisaki. A statistically-hiding integer commitment scheme based on groups with hidden order. In Yuliang Zheng, editor, *Advances in Cryptology - ASIACRYPT 2002, 8th International Conference on the Theory and Application of Cryptology and Information Security, Queenstown, New Zealand, December 1-5, 2002, Proceedings*, volume 2501 of *Lecture Notes in Computer Science*, pages 125–142. Springer, 2002. pages 12

[27] Emma Dauterman, Henry Corrigan-Gibbs, and David Mazières. Safetypin: Encrypted backups with human-memorable secrets. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 1121–1138. USENIX Association, 2020. pages 9

[28] Tien Tuan Anh Dinh, Prateek Saxena, Ee-Chien Chang, Beng Chin Ooi, and Chunwang Zhang. M2R: enabling stronger privacy in mapreduce computation. In Jaeyeon Jung and Thorsten Holz, editors, *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, pages 447–462. USENIX Association, 2015. pages 2

[29] Ravi Ganesan. How to use key escrow (introduction to the special section). *Commun. ACM*, 39(3):32–33, 1996. pages 2, 9

[30] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM J. Comput.*, 18(1):186–208, 1989. pages 12

[31] Hyperledger. Hyperledger/fabric. `https://github.com/hyperledger/fabric`. pages 38

[32] Stanislaw Jarecki, Aggelos Kiayias, Hugo Krawczyk, and Jiayu Xu. TOPPSS: cost-minimal password-protected secret sharing based on threshold OPRF. In Dieter Gollmann, Atsuko Miyaji, and Hiroaki Kikuchi, editors, *Applied Cryptography and Network Security - 15th International Conference, ACNS 2017, Kanazawa, Japan, July 10-12, 2017, Proceedings*, volume 10355 of *Lecture Notes in Computer Science*, pages 39–58. Springer, 2017. pages 10

[33] Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. OPAQUE: an asymmetric PAKE protocol secure against pre-computation attacks. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part III*, volume 10822 of *Lecture Notes in Computer Science*, pages 456–486. Springer, 2018. pages 10

[34] Gabriel Kaptchuk, Matthew Green, and Ian Miers. Giving state to the stateless: Augmenting trustworthy computation with ledgers, 2019. pages 2, 12, 14

[35] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography, Second Edition*. CRC Press, 2014. pages 10, 11, 12

[36] Ranjit Kumaresan, Tal Moran, and Iddo Bentov. How to use bitcoin to play decentralized poker. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, pages 195–206. ACM, 2015. pages 2

[37] Pascal Lafourcade, Lola-Baie Mallordy, Charles Olivier-Anclin, and Léo Robert. Secure keyless multi-party storage scheme. In *ESORICS*, 2024. pages 9

[38] Legrandin. Legrandin/pycryptodome: A self-contained cryptographic library for python. `https://github.com/Legrandin/pycryptodome?tab=readme-ov-file`. pages 37

[39] Yehuda Lindell. Cryptography and mpc in the coinbase prime web3 wallet. Online, 2023. pages 9

[40] Deepak Maram, Mahimna Kelkar, and Ittay Eyal. Interactive authentication. *IACR Cryptol. ePrint Arch.*, page 1682, 2022. pages 10

[41] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In Ruby B. Lee and Weidong Shi, editors, *HASP 2013, The Second Workshop on Hardware and Architectural Support for Security and Privacy, Tel-Aviv, Israel, June 23-24, 2013*, page 10. ACM, 2013. pages 2

[42] Leila Megouache, Abdelhafid Zitouni, and Mahieddine Djoudi. Ensuring user authentication and data integrity in multi-cloud environment. *Hum. centric Comput. Inf. Sci.*, 10:15, 2020. pages 9

[43] Silvio Micali. Fair public-key cryptosystems. In Ernest F. Brickell, editor, *Advances in Cryptology - CRYPTO '92, 12th Annual International Cryptology Conference, Santa Barbara, California, USA, August 16-20, 1992, Proceedings*, volume 740 of *Lecture Notes in Computer Science*, pages 113–138. Springer, 1992. pages 2, 9

[44] Ujan Mukhopadhyay, Anthony Skjellum, Oluwakemi Hambolu, Jon Oakley, Lu Yu, and Richard R. Brooks. A brief survey of cryptocurrency systems. In *14th Annual Conference on Privacy, Security and Trust, PST 2016, Auckland, New Zealand, December 12-14, 2016*, pages 745–752. IEEE, 2016. pages 1

[45] Ahad Niknia, Miguel Correia, and Jaber Karimpour. Secure cloud-of-clouds storage with space-efficient secret sharing. *Journal of Information Security and Applications*, 59:102826, 2021. pages 9

[46] Nthparty. Nthparty/oblivious: Python library that serves as an api for common cryptographic primitives used to implement oprf, ot, and psi protocols. `https://github.com/nthparty/oblivious`. pages 37

[47] Rafael Pass, Elaine Shi, and Florian Tramèr. Formal abstractions for attested execution secure processors. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part I*, volume 10210 of *Lecture Notes in Computer Science*, pages 260–289, 2017. pages 2, 3, 5, 13, 27, 29

[48] Riccardo Di Pietro, Marco Scarpa, Maurizio Giacobbe, and Antonio Puliafito. Secure storage as a service in multi-cloud environment. In Antonio Puliafito, Dario Bruneo, Salvatore Distefano, and Francesco Longo, editors, *Ad-hoc, Mobile, and Wireless Networks - 16th International Conference on Ad Hoc Networks and Wireless, ADHOC-NOW 2017, Messina, Italy, September 20-22, 2017, Proceedings*, volume 10517 of *Lecture Notes in Computer Science*, pages 328–341. Springer, 2017. pages 9

[49] Raluca A. Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: processing queries on an encrypted database. *Commun. ACM*, 55(9):103–111, 2012. pages 1

[50] Alessandra Scafuro. Break-glass encryption. In Dongdai Lin and Kazue Sako, editors, *Public-Key Cryptography - PKC 2019 - 22nd IACR International Conference on Practice and Theory of Public-Key Cryptography, Beijing, China, April 14-17, 2019, Proceedings, Part II*, volume 11443 of *Lecture Notes in Computer Science*, pages 34–62. Springer, 2019. pages 1, 2, 3, 8, 10

[51] Ricardo Guilherme Schmidt, Miguel Mota, Vitalik Buterin, and naxe. Eip-2429: Add ability for recovery of burned keys used in multisig wallets, 2019. pages 8

[52] Adi Shamir. Partial key escrow: A new approach to software key escrow. In *Key escrow conference*, 1995. pages 2, 9

[53] Jacob Swambo and Antoine Poinsot. Risk framework for bitcoin custody operation with the revault protocol. In Matthew Bernhard, Andrea Bracciali, Lewis Gudgeon, Thomas Haines, Ariah Klages-Mundt, Shin'ichiro Matsuo, Daniel Perez, Massimiliano Sala, and Sam Werner, editors, *Financial Cryptography and Data Security. FC 2021 International Workshops - CoDecFin, DeFi, VOTING, and WTSC, Virtual Event, March 5, 2021, Revised Selected Papers*, volume 12676 of *Lecture Notes in Computer Science*, pages 3–20. Springer, 2021. pages 8

[54] Shaileshh Bojja Venkatakrishnan, Giulia Fanti, and Pramod Viswanath. Dandelion: Redesigning the bitcoin network for anonymity. *Proc. ACM Meas. Anal. Comput. Syst.*, 1(1):22:1–22:34, 2017. pages 4

[55] Zooko Wilcox-O'Hearn and Brian Warner. Tahoe: the least-authority filesystem. In Yongdae Kim and William Yurcik, editors, *Proceedings of the 2008 ACM Workshop On Storage Security And Survivability, StorageSS 2008, Alexandria, VA, USA, October 31, 2008*, pages 21–26. ACM, 2008. pages 9

[56] Elizabeth Nathania Witanto, Brian Stanley, and Sang-Gon Lee. Correction: Witanto et al. distributed data integrity verification scheme in multi-cloud environment. sensors 2023, 23, 1623. *Sensors*, 23(12):5566, 2023. pages 9

[57] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In Aditya Akella and Jon Howell, editors, *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, pages 283–298. USENIX Association, 2017. pages 2