Cryptographically Secure Digital Consent

F. Betül Durak[1], Abdullah Talayhan[2], and Serge Vaudenay[2]

[1] Microsoft Research, Redmond, USA
[2] EPFL, Lausanne, Switzerland

Abstract. In the digital age, the concept of consent for online actions executed by third parties is crucial for maintaining trust and security in third-party services. This work introduces the notion of cryptographically secure digital consent, which aims to replicate the traditional consent process in the online world. We provide a flexible digital consent solution that accommodates different use cases and ensures the integrity of the consent process.

The proposed framework involves a client (referring to the user or their devices), an identity manager (which authenticates the client), and an agent (which executes the action upon receiving consent). It supports various applications and ensures compatibility with existing identity managers. We require the client to keep no more than a password. The design addresses several security and privacy challenges, including preventing offline dictionary attacks, ensuring non-repudiable consent, and preventing unauthorized actions by the agent. Security is maintained even if either the identity manager or the agent is compromised, but not both.

Our notion of an identity manager is broad enough to include combinations of different authentication factors such as a password, a smartphone, a security device, biometrics, or an e-passport. We demonstrate applications for signing PDF documents, e-banking, and key recovery.

1 Introduction

In a non-digital world, giving consent for an action to be executed by a third party is a common social and legal interaction. One can give power of attorney to an agent to sign a document on their behalf, or one can ask to store their keys in a storage service for a period of time. Any action performed by the agent on behalf of a person may require an explicit permission (a.k.a. consent) from that person, for liability reasons. An explicit consent (in the form of writing or signing) from the person to an agent serves as non-repudiable proof of intention that can be shown in the case of a dispute. In this work, we introduce the notion of digital consent, where we study this problem in the digital world.

Building a framework of consent in the digital world requires several steps: defining the parties involved, making careful choices of the constraints we put in the design, and realizing the security and privacy requirements. One trivial description of a digital consent is as follows: a user casts an *order* to their online *agent* who behaves as a delegate to perform some actions on behalf of the user. The order is kept as a proof that the authentic and authorized user has indeed given permission to execute the order. Such consent should be non-repudiable. At first, the functionality described above seems to be immediately realized using digital signatures, where the order for the action becomes digitally signed by the client. However, the client would need to maintain the signature key, and to protect against leakage or loss. Giving consent this way does not offload any burden from the client. Hence, it requires careful thinking on constructions where the client only keeps a low entropy secret such as a password.

We wonder how we can model cryptographically secure digital consent so that it provides a flexible framework accommodating applications without needing to change from one to another. To achieve this, first, we consider a model where there are (at least) three participants: an Identity Manager (IdM), a client (like a browser or an app) which works for its authentic user, and an Agent. There will be several types of use-cases determined by the usage of consent. Consent should support various applications with straightforward extensions such as multiple IdMs; multiple modalities such as standard use with a client having secure hardware (smartphone), but requiring backup solutions when the device is broken, lost, or stolen; and distributed Agents as detailed in section 5.

We additionally have to work with several constraints to make the proposed protocol deployable in the real world. First, we must not make any changes to how the existing IdMs, such as Microsoft or Google, work. For example, we would need to be compatible with JSON Web Token (JWT) generation in the Open ID Connect (OIDC) protocol and standard JWT verification by the Client and Agent. Second, it would be beneficial to leverage already widely deployed cryptography. More precisely, we should not require any changes to certain cryptographic operations such as digital signature verification. Finally, we should not rely on long-term storage on the client side. [3]

We designed a straw-man application for digital document signing where the user delegates their signing secret key to the cloud (referred to as the Agent). Whenever a document needs to be signed, the user logs into the cloud and requests the Agent to sign the document on their behalf. The cloud is held accountable for any documents signed without the user's consent. Therefore, the Agent must maintain a record of consents to resolve any dispute. The process involves two phases: (1) enrollment to the cloud, which needs a password for the Agent to authenticate the user and a binding contract which registers the information on how to verify the consent for the user, and (2) consent generation, which needs authentication with another password to the IdM and a signed OIDC token which serves as an *order*. We assert that our concrete protocol design, along with formal security notions and proofs, consolidates the level of security and functionality that products like DocuSign aim to achieve [Doc24].

Such a design comes with several security and privacy challenges. First and foremost, we need to prevent offline dictionary attacks on the password during the enrollment phase so that neither IdM nor any other party can run it. Second, our protocol must prevent forging the digital consent without the client's involve-

[3] Except a password which is the most common authentication method used in today's digital world.

ment and prevent IdM from requesting an action without the client. Moreover, IdM should not be able to request an action with the client that is different from the intended client action. Another important security vulnerability to consider is that the Agent may run an action without it being triggered by the client. However, depending on the use-case, our protocol design will prevent it by (1) a simple dispute protocol; (2) a trusted back-end; or (3) a specific application. Finally, what we may need, but do not provide in our current work, is the unlinkability between the order request from the IdM and the consent generated by the Agent. Anonymous tokens seem like a natural solution to this, but it requires changes to the IdM (in a sense that the protocol would not be OIDC compliant anymore) and pairing operations (implies a more inefficient dispute protocol) for public verifiability. Hence, in this work, we stick to the design requirements and security goals we defined without the unlinkability notion.

2 Related Work

Password-based signatures (PBS). The aim of digital consent is to give a non-repudiable authorization for an attribute (input) to be used along with a pre-defined key material. Our consent is already a signature in itself. This signature is jointly computed by all participants.

PBS schemes are the most similar primitive to digital consent. In PBS, the client (in possession of a password) and the Agent jointly compute a signature σ on a message m which would verify under the client's public key pk. The requirement of PBS to be secure is that the password should be protected from offline dictionary attacks to keep the entropy requirement of the password in memorable levels. [GT12] formalized the notion of PBS and provided two constructions based on RSA and CL blind signatures the latter being secure under Generic Group Model (GGM). [JKR13] improved on [GT12] by providing a PBS scheme using the BLS signature scheme proven secure under the random oracle model (ROM). They further construct a strongly secure variant of PBS in which smaller entropy passwords can be used without losing security. However, their construction is prone to offline dictionary attacks by the Agent that is helping with the signing (insider attack) and also by any adversary that is able to send a single signing request to an honest Agent (outsider attack). We refer to section B for the details.

Earlier, [HWF05] introduced server-aided digital signatures (SADS) which utilizes a two-server mechanism. The utilization of two non-colluding servers solves the problem of offline dictionary attacks in [GT12], assuming that at least one server remains non corrupted. [Cam+16] takes a similar approach to SADS where one of the servers is modeled as a user device.

Proxy signatures (PS). One of the main applications of digital consent is the signing service where Agent signs documents on behalf of the client. This application shares similarities with PS but has some differences. Proxy signatures use the following terminology: a delegator (which corresponds to the client), a delegate (which is the Agent), and the delegation (which is the consent in our terminology). PS schemes such as [AN23] require the anonymity of delegation which we do not. Delegation in PS is message-agnostic while our consent can be message-specific: we authorize to sign a specific message. In PS, the delegator needs to keep a high-entropy key to sign a warrant while we only require to keep a password. PS are specific to one signature scheme while our notion of consent is independent from the signature which is produced by the Agent in the end. Furthermore, PS imply a strong binding between the delegation (consent) and signature capability. Last but not least, PS require "unframability", meaning the delegate cannot sign without the delegator's consent. One drawback of unframibility is the recoverability: if the delagator is not able to delegate (because of lost devices or keys), the delegate cannot sign at all. In our work, the Agent may sign without the client's consent (the signing is done independent of the consent procedure) but we ensure unframeability by a dispute protocol in which an abusive agent can be faced to evidence of misbehavior, implying some legal consequences or loss of reputation in their business.

Some works to be cited under this theme include: [DHS14], [HS13], [FP09].

Key recovery. Another application of digital consent is the key recovery, which allows a client to recover their keys without depending on a personal device. The notion of credential retrieval was studied by Boyen [Boy09]. It allows a client to store their credentials on a server and to retrieve them using a password. The idea behind credential retrieval is to use an oblivious pseudorandom function (OPRF). This is a 2-party protocol allowing a client to compute $K = f_k(\mathsf{pw})$ when pw is the input of the client and k is the input of the server, for a specific pseudorandom function f. This way, the client can retrieve a key K from a password pw which allows to decrypt some storage. This is not enough to protect against a malicious server running an offline dictionary attack as it is very likely that they can test a value K offline. In our model, we rely on an additional participant (the identity provider) and assume that at this participant is honest when the server (or Agent as we call it) is not. Similarly, SADS [HWF05] uses two OPRFs with two servers to retrieve a signing key. Acar et al. [ABK13] separate the OPRF server and the storage for similar reasons. Likewise, Camenisch et al. [Cam+14] use several servers.

With digital consent, we rely on a secure hardware (for this specific application). Strictly speaking, credential retrieval does not rely on a trusted hardware and does not require explicit authentication of the client to the server. However, it has the fragility to offline dictionary attack. With digital consent, we rely on a secure hardware (for this specific application).

The Boyen method was improved by Miyaji et al. [MRS10] without random oracles and more recently by Davies et al. [Dav+23] and Faller et al. [Fal+24] with the Password Protected Key Retrieval (PPKR) protocol, considering several corruption models of the secure hardware being used. Compared to PPKR, we propose a protocol which allows to work with a stateless server, we offer more options for the authentication model, and we make sure that the server can keep undeniable evidence of consent.[4]

Password-based credentials (PBC). Another application of digital consent is to provide credentials to access to services. It goes beyond password-based access control as the service may require more than just *passing a gate*. For example, in smart contracts, it is hard to authenticate an order to a smart contract by using a password. The authors of [Bal+24] studies the problem of helping a client to make a transaction on a blockchain (or a smart contract running on the blockchain). Essentially, making a transaction is signing it; except that now, the client cannot miss that the signature is made as it will appear as a transaction on the blockchain. With our digital consent formalism, we can have the transaction made by the agent once the consent is there. Moreover, our consent is easily verifiable by a smart contract and can be used like zkLogin [Bal+24]. Their version also needs two servers but uses an expensive zk-SNARK to provide additional privacy guarantees.

Dayanikli and Lehmann's PBC scheme [DL24] uses the terminology differently. In their framework, the client keeps a high-entropy private credential which allows them to sign with a password. However, signatures are not publicly verifiable: the verification key is private. Security assumes that either the credential or the verification key is unknown to the adversary.

3 The Consent Protocol

We consider a simplified model with the following entities to illustrate how consents work:

3.1 Entities

We have 4 entities in our design:

1. Client is the device working for its authentic user who is giving the consent. We do not distinguish the Client and the user.
2. IdM is the identity provider that Client can interact with to receive access tokens for their identities.[5]
3. Agent is the entity that acts on behalf of a Client with a given consent.
4. Judge is the entity that verifies the consent.[6]

[4] In PPKR, the client sends a signature as a last message to reset the counter of bad attempts and avoid a future denial of service. This can play the role of a digital consent, but a malicious client may skip this last step.

[5] There can be more than one IdM, an IdM can be a trusted hardware (actually, an identity manager for a single client), IdM can be a complex combination of several elementary ones. We discuss it in subsection 5.2.

[6] In our protocol, Judge can be anyone as we allow public verification.

3.2 Interfaces

We define the interfaces as follows and depict the flow of the procedures in Figure 1.

- $(\mathsf{sk}_{\mathsf{IdM}}, \mathsf{pk}_{\mathsf{IdM}}) \leftarrow \mathsf{IdM.Setup}(1^\lambda)$: IdM generates their secret/public keys. It is run by the IdM and re-run every time the keys are rotated.
- $(s_\mathsf{A}, \mathsf{contract}) \leftarrow \mathsf{Enroll}(\mathsf{ID}, \mathsf{pw}, \mathsf{pk}_{\mathsf{IdM}})$: enrolls the Client under pw to an Agent. Generates s_A and contract. s_A and contract are sent to the Agent. The contract element defines how a consent can be verified. It is binding.
- $(\mathsf{query}, \mathsf{state}) \leftarrow \mathsf{Launch}(\mathsf{pw}, att)$: Client initiates the consent protocol by sending a query to IdM and keeping the state. Client wants to consent to some action which is defined by att.
- $\mathsf{resp} \leftarrow \mathsf{IdM}(\mathsf{ID}, \mathsf{sk}_{\mathsf{IdM}}, \mathsf{query})$: IdM receives the query from the Client and sends back resp.
- $\mathsf{order} \leftarrow \mathsf{Commit}(\mathsf{state}, \mathsf{resp})$: Client creates an order of a consent with resp and state. order is sent to Agent. This is the final operation by which the Client gives consent on att.
- $\mathsf{consent} \leftarrow \mathsf{Agent}(s_\mathsf{A}, \mathsf{contract}, att, \mathsf{order})$: Agent receives the order and outputs a consent with $(\mathsf{contract}, att)$.
- $0/1 \leftarrow \mathsf{Verify}(\mathsf{contract}, att, \mathsf{consent})$: The verification procedure takes as input the contract and a consent issued on att. Returns 1 if the consent is valid. 0 otherwise.

Figure 1 shows the interface between the IdM, Client, and Agent. Enrollment phase enrolls the Client to Agent through login and pw picked by the client. ID is the identity under which the Client is known to IdM. This could be identical to login but does not need to. We keep the distinction between ID and login for clarity.

The Client is stateless. In the enrollment phase, both s_A and contract is sent to the Agent to store. During the consent phase, Client is only required to enter the pw which corresponds to the login for Agent and authenticate itself to the IdM.

We say that the protocol is *correct* if for any pw, ID, att and any result of Setup, running all these protocols in sequence leads to Verify to return 1.

3.3 Security Assumptions and Adversarial Model

We assume secure communication between participants. More precisely, an adversary must not interfere with communication. Furthermore, communications to Agent are confidential. For instance, this can be done with a regular TLS connection, which implies trust in the browser/app on the client side or the PKI root certificates.

We assume that Client has secure means to authenticate to IdM. How it is done depends on how IdM is implemented. We discuss it in subsection 5.2.

We assume that Judge has means to trust that contract is correct. This can be done by some kind of notary service. It could be an additional external service

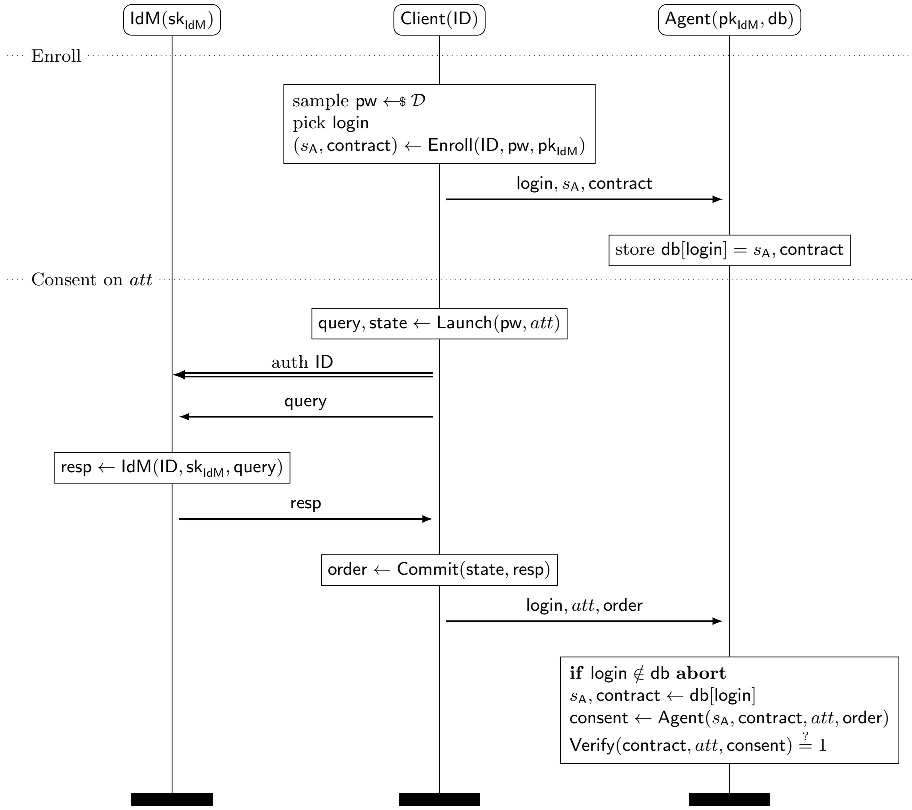

Fig. 1. Consent Protocol Interface

with a trusted third party or a blockchain. We could also use IdM and Agent to sign contract. We would need a collusion between a malicious IdM and a malicious Agent to forge a rogue contract to be used in a consent protocol.

We assume that the storage of Agent does not leak. Hence, s_{A} is kept safe.

Our threat model captures the possible corruption of IdM or Agent but not both at the same time, as well as the existence of corrupted clients. We want to protect a honest Client from being shown a valid consent, relative to some contract that they agreed on, to some action *att* which was not intended by them to consent.

3.4 Unforgeability

We model Consent UnForgeability (CUF) in two different scenarios: (1) an adversary controls the IdM and can corrupt arbitrarily many client and tries to forge a valid consent for an honest client (victim) and (2) an adversary controls the Agent, corrupts as many clients as possible and tries to forge a valid consent for an honest client.

Definition 1 (Consent Unforgeability with corrupted IdM). *Let $\mathcal{D}$ be a distribution with min-entropy $\mathcal{D}_{min}$. In the $\mathsf{CUF}_{\mathit{IdM}}$ game defined in Figure 2, we define the advantage of the $\mathcal{A}$ as follows:*

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{CUF}_{\mathit{IdM}}}(\lambda, \mathcal{D}) = \Pr[\mathsf{CUF}_{\mathit{IdM}}(\mathcal{A}, \mathcal{D}) \to 1]$$

We say that a Digital Consent is secure under $\mathsf{CUF}_{\mathit{IdM}}$ if for any PPT adversary $\mathcal{A}$ limited to q queries to OOrder [7], we have:

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{CUF}_{\mathit{IdM}}}(\lambda, \mathcal{D}) \leq \frac{1+q}{2^{\mathcal{D}_{min}}} + \mathsf{negl}(\lambda)$$

In the $\mathsf{CUF}_{\mathsf{IdM}}$ security notion, the adversary controls IdM. The victim client is enrolled at the beginning of the game by running Enroll and can issue consents using OLaunch and OCommit. It returns the final consent. Other clients can be assumed to be corrupted by default and be enrolled with OCorruptEnroll. They can cast orders using OOrder which returns the consent.

Contrarily, in the $\mathsf{CUF}_{\mathsf{Agent}}$ security notion, the adversary controls Agent. Clients can be enrolled honestly with OEnroll or maliciously with OCorruptEnroll. The consent issuance protocol is triggered for a honest client with OLaunch. It returns the communication between Client and IdM (which are not assumed to be private) and order to be submitted to Agent. Since pw can be recovered offline by the corrupted Agent, it plays no role in security and can be chosen by the adversary for honest clients.

Definition 2 (Consent Unforgeability with corrupted Agent). *A Digital Consent scheme is unforgeable against a corrupted Agent if for any PPT adversary $\mathcal{A}$, we have*

$$\Pr[\mathsf{CUF}_{\mathit{Agent}}(\mathcal{A}) \to 1] \leq \mathsf{negl}(\lambda)$$

where the $\mathsf{CUF}_{\mathit{Agent}}$ game is defined in Figure 3.

4 Our Construction

We first introduce the building blocks: the commitment scheme and zero-knowlege protocol.

4.1 Additively Homomorphic Commitment

We use an additively homomorphic multi-message commitment scheme, i.e. given tuples of m messages $(x_1, x_2, \ldots, x_m)$ and $(y_1, y_2, \ldots, y_m)$ with randomness r and s, we have

$$\mathsf{Com}(x_1, \ldots, x_m; r) + \mathsf{Com}(y_1, \ldots, y_m; s) = \mathsf{Com}(x_1 + y_1, \ldots, x_m + y_m; r + s)$$

We require that Com is perfectly hiding and computationally binding.

In our construction, we use $m = 2$.

[7] We count the OOrder queries since each query to OOrder can be utilized as a pw correctness check.

$\mathsf{CUF}_{\mathsf{IdM}}(\mathcal{A}, \mathcal{D})$

$\mathsf{given} \leftarrow \emptyset$
$\mathsf{cst} \leftarrow \{\}$ // challenge state
$\mathsf{pk}_{\mathsf{IdM}}, \mathsf{st}, \mathsf{login}^*, \mathsf{ID}^* \leftarrow \mathcal{A}_1(1^\lambda)$
$\mathsf{pw}^* \leftarrow\$ \mathcal{D}$
$\mathsf{enrolled} \leftarrow \{\mathsf{login}^*\}$
$(s^*_{\mathsf{A}}, \mathsf{contract}^*) \leftarrow \mathsf{Enroll}(\mathsf{ID}^*, \mathsf{pw}^*, \mathsf{pk}_{\mathsf{IdM}})$
$\mathsf{db}[\mathsf{login}^*] \leftarrow s^*_{\mathsf{A}}, \mathsf{contract}^*$
$\mathcal{A}_2^{\text{oracles}}(\mathsf{contract}^*, \mathsf{st}) \rightarrow att^*, \mathsf{consent}^*$
if $\mathsf{Verify}(\mathsf{contract}^*, att^*, \mathsf{consent}^*)$
$\wedge\ \mathsf{consent}^* \notin \mathsf{given}$:
  return 1
return 0

$\mathsf{OCorruptEnroll}(\mathsf{login}, s_{\mathsf{A}}, \mathsf{contract})$

if $\mathsf{login} \in \mathsf{enrolled}$
  abort
$\mathsf{enrolled} \leftarrow \mathsf{enrolled} \cup \mathsf{login}$
$\mathsf{db}[\mathsf{login}] \leftarrow s_{\mathsf{A}}, \mathsf{contract}$
return $\perp$

$\mathsf{OLaunch}(sid, att)$

if $sid \in \mathsf{cst}$
  abort
$(\mathsf{query}, \mathsf{state}) \leftarrow \mathsf{Launch}(\mathsf{pw}^*, att)$
$\mathsf{cst}[sid].att \leftarrow att$
$\mathsf{cst}[sid].\mathsf{state} \leftarrow \mathsf{state}$
return query

$\mathsf{OCommit}(sid, \mathsf{resp})$

if $sid \notin \mathsf{cst}$
  abort
$\mathsf{order} \leftarrow \mathsf{Commit}(\mathsf{cst}[sid].\mathsf{state}, \mathsf{resp})$
$\mathsf{db}[\mathsf{login}^*] \rightarrow s^*_{\mathsf{A}}, \mathsf{contract}^*$
$\mathsf{cst}[sid].\mathsf{state} \rightarrow att$
remove sid from cst
$\mathsf{consent} \leftarrow \mathsf{Agent}(s^*_{\mathsf{A}}, \mathsf{contract}^*,$
  $, att, \mathsf{order})$
if $\mathsf{consent} = \perp$
  return false
$\mathsf{given} \leftarrow \mathsf{given} \cup \mathsf{consent}$
return $\mathsf{consent}$

$\mathsf{OOrder}(\mathsf{login}, att, \mathsf{order})$

$\mathsf{db}[\mathsf{login}] \rightarrow s_{\mathsf{A}}, \mathsf{contract}$
$\mathsf{consent} \leftarrow \mathsf{Agent}(s_{\mathsf{A}}, \mathsf{contract}, att, \mathsf{order})$
return $\mathsf{consent}$

Fig. 2. $\mathsf{CUF}_{\mathsf{IdM}}$ Game.

$\mathsf{CUF}_{\mathsf{Agent}}(\mathcal{A})$

```
corrupted ← ∅
ordered ← ∅
(sk_IdM, pk_IdM) ← IdM.Setup(1^λ)
A^oracles(pk_IdM) → ID*, att*, consent*
abort if ID* ∉ db
db[ID*] → contract*
if Verify(contract*, att*, consent*)
 ∧ (ID*, att*) ∉ ordered :
   return 1
return 0
```

$\mathsf{OEnroll}(\mathsf{ID}, \mathsf{pw})$

```
if ID ∈ db ∨ ID ∈ corrupted
   abort
(s_A, contract) ← Enroll(ID, pw, pk_IdM)
db[ID] ← (contract, pw)
return (s_A, contract)
```

$\mathsf{OLaunch}(\mathsf{ID}, att)$

```
if ID ∉ db
   abort
db[ID] → pw
(query, state) ← Launch(pw, att)
resp ← IdM(ID, sk_IdM, query)
order ← Commit(state, resp)
ordered ← ordered ∪ (ID, att)
return query, resp, order
```

$\mathsf{OCorruptQuery}(\mathsf{ID}, \mathsf{query})$

```
if ID ∉ corrupted
   abort
resp ← IdM(ID, sk_IdM, query)
return resp
```

$\mathsf{OCorruptEnroll}(\mathsf{ID})$

```
if ID ∈ db
   abort
corrupted ← corrupted ∪ ID
return ⊥
```

Fig. 3. $\mathsf{CUF}_{\mathsf{Agent}}$ Game

Example. Let (G_1, G_2, G_3) be three generators of a group $\mathbb{G}$. We use a multi-message Pedersen commitment to commit to 2 messages (x_1, x_2) as follows:

$$\mathsf{Com}(x_1, x_2; r) = x_1 \cdot G_1 + x_2 \cdot G_2 + r \cdot G_3$$

The above scheme is additively homomorphic, perfectly hiding, and computationally binding (assuming the hardness of the discrete logarithm problem).

4.2 NIZK for Commitment to the Same Value

Given two commitments $\mathsf{com}_{\mathsf{ID}}$ and com and a value att, we would like to prove the following statement:

$$\mathsf{NIZK}\{(a, r_1, r_2) : \underbrace{\mathsf{Com}(a, 0; r_1) = \mathsf{com}_{\mathsf{ID}} \wedge \mathsf{Com}(a, att; r_2) = \mathsf{com}}_{\mathcal{R}((\mathsf{com}_{\mathsf{ID}}, \mathsf{com}, att), (a, r_1, r_2))}\}$$

The relation $\mathcal{R}(x, w)$ is between a statement $x = (\mathsf{com}_{\mathsf{ID}}, \mathsf{com}, att)$ and a witness $w = (a, r_1, r_2)$.

We use a generalized Schnorr proof [Sch89; Sch91] compiled with Fiat-Shamir [FS86]. We denote the prover and verifier for the NIZK above as $\Pi_{\mathsf{MULTEQ}}.\mathsf{PoK}^H$, $\Pi_{\mathsf{MULTEQ}}.\mathsf{Verify}$ respectively which are explicitly defined in Figure 4. We further denote the HVZK-simulator $\Pi_{\mathsf{MULTEQ}}.\mathsf{Sim}$ which on input a statement x, returns an accepting view $(\alpha, \mathsf{ch}, \gamma)$ with $\alpha = (\mathsf{com}_1, \mathsf{com}_2)$ and $\gamma = (\mathsf{resp}_1, \mathsf{resp}_2, \mathsf{resp}_3)$. The generalized Schnorr proof is a non-trivial Σ-protocol with s unique response in the sense of [Fau+12].

$\Pi_{\mathsf{MULTEQ}}.\mathsf{PoK}^H(\mathsf{com}_{\mathsf{ID}}, \mathsf{com}, att, h_{\mathsf{ID}}, r_{\mathsf{ID}}, r)$

- $a, b, c \leftarrow\!\!\$\ \mathbb{Z}_q^3$
- $\mathsf{com}_1 \leftarrow \mathsf{Com}(a, 0; b)$
- $\mathsf{com}_2 \leftarrow \mathsf{Com}(a, 0; c)$
- $\mathsf{ch} \leftarrow H(\mathsf{com}_{\mathsf{ID}}, \mathsf{com}, att, \mathsf{com}_1, \mathsf{com}_2)$
- $\mathsf{resp}_1 \leftarrow a + \mathsf{ch} \cdot h_{\mathsf{ID}}$
- $\mathsf{resp}_2 \leftarrow b + \mathsf{ch} \cdot r_{\mathsf{ID}}$
- $\mathsf{resp}_3 \leftarrow c + \mathsf{ch} \cdot r$
- $\pi \leftarrow (\mathsf{ch}, \mathsf{resp}_1, \mathsf{resp}_2, \mathsf{resp}_3)$
- **return** π

$\Pi_{\mathsf{MULTEQ}}.\mathsf{Verify}(\pi, \mathsf{com}_{\mathsf{ID}}, \mathsf{com}, att)$

- $\pi \rightarrow \mathsf{ch}, \mathsf{resp}_1, \mathsf{resp}_2, \mathsf{resp}_3$
- $\mathsf{com}'_1 \leftarrow \mathsf{Com}(\mathsf{resp}_1, 0; \mathsf{resp}_2) - \mathsf{ch} \cdot \mathsf{com}_{\mathsf{ID}}$
- $\mathsf{com}'_2 \leftarrow \mathsf{Com}(\mathsf{resp}_1, \mathsf{ch} \cdot att; \mathsf{resp}_3) - \mathsf{ch} \cdot \mathsf{com}$
- $\mathsf{ch}' \leftarrow H(\mathsf{com}_{\mathsf{ID}}, \mathsf{com}, att, \mathsf{com}'_1, \mathsf{com}'_2)$
- **return** $\mathsf{ch} = \mathsf{ch}'$

$\Pi_{\mathsf{MULTEQ}}.\mathsf{Sim}(\mathsf{com}_{\mathsf{ID}}, \mathsf{com}, att)$

- $\mathsf{ch}, \mathsf{resp}_1, \mathsf{resp}_2, \mathsf{resp}_3 \leftarrow\!\!\$\ \mathbb{Z}_q^4$
- $\mathsf{com}_1 \leftarrow \mathsf{Com}(\mathsf{resp}_1, 0; \mathsf{resp}_2) - \mathsf{ch} \cdot \mathsf{com}_{\mathsf{ID}}$
- $\mathsf{com}_2 \leftarrow \mathsf{Com}(\mathsf{resp}_1, \mathsf{ch} \cdot att; \mathsf{resp}_3) - \mathsf{ch} \cdot \mathsf{com}$
- $\alpha \leftarrow (\mathsf{com}_1, \mathsf{com}_2)$
- $\gamma \leftarrow (\mathsf{resp}_1, \mathsf{resp}_2, \mathsf{resp}_3)$
- **return** $(\alpha, \mathsf{ch}, \gamma)$

Fig. 4. NIZK for Π_{MULTEQ}

We define three oracles $S_1(x, \alpha)$, $S'_2(x)$, and $S_2(x, w)$. The oracle $S_1(x, \alpha)$ simulates $H(x, \alpha)$ by lazy sampling: it maintains a table LH which is originally empty and, upon a query (x, α), returns $\mathsf{LH}(x, \alpha)$, possibly by first defining it at random if not already defined. The oracle $S_2'(x)$ runs $\mathsf{Sim}(x) \to (\alpha, \mathsf{ch}, \gamma)$. Then, it aborts if $\mathsf{LH}(x, \alpha)$ is already defined. Otherwise, it defines $\mathsf{LH}(x, \alpha) = \mathsf{ch}$ (this is called *programming* the random oracle) and returns $\pi = (\mathsf{ch}, \gamma)$. The oracle $S_2(x, w)$ first verifies that $\mathcal{R}(x, w)$ is true (it returns $\perp$ otherwise), then continues with $S_2'(x)$.

We use the fact that the NIZK is zero-knowledge in the random oracle model, following [Fau+12, thm. 1]. It means that running a PPT adversary $\mathcal{A}$ interacting with either the two oracles H and PoK or the two oracles S_1 and S_2 produce indistinguishable outputs. The advantage of the distinguisher is the probability that S_2' ever aborts. In our NIZK case, the probability of a matching between two hash queries is $\frac{1}{q^2}$.

We also use the fact that the NIZK is weakly simulation extractable, following [Fau+12, thm. 3]. This means that for any PPT adversary $\mathcal{A}$ interacting with S_1 and S_2' with a total of q_S queries, and forging with probability p_{acc} a proof (x^*, π^*) which is valid and not the result of a query to S_2', there exists an extractor $\mathcal{E}$ such that running $\mathcal{A}$ then $\mathcal{E}$ with the coins of $\mathcal{A}$ and the input/output queries to S_1 and S_2', the probability that $\mathcal{A}$ forges a new valid proof (x^*, π^*) and $\mathcal{E}$ extracts a witness w^* satisfying $\mathcal{R}(x^*, w^*)$ is p_{ext} such that

$$p_{ext} \geq \frac{1}{q_S} \left(p_{acc} - \frac{q_S}{q} \right)^2$$

4.3 Our Protocol Design

Our goal in the protocol design is to make no modifications to the existing IdM structure. For example, we do not assume any new secrets or new operations done by the IdM. The IdM interface is used to generate OpenID Connect (OIDC) tokens [8].

In Figure 6, the Enroll protocol computes the hash of the pw and commitment on that with randomness r_{ID}. The hash and r_{ID} forms the s_{A} secret. The $\mathsf{contract}$ is specified as ID (as known by the IdM), the commitment on the hash of the password, and public key of the IdM.

Essentially, Client commits to pw and the commitment is put inside $\mathsf{contract}$. The value s_{A} includes the opening information for the commitment. To consent to *att*, Client commits to pw and to *att* and the commitment is sent as a nonce to the OIDC service of IdM. Hence, after authenticating ID, IdM signs this commitment together with ID. The signed commitment is sent to Agent together with the opening information. Then, Agent can make a non-interactive zero-knowledge proof that the commitment in $\mathsf{contract}$ and the signed commitment commit to the same secret pw. The final $\mathsf{consent}$ consists of the signed commitment and this proof. The opening information is then destroyed.

We recall that it is essential that the communication between Client and Agent is confidential, as it would reveal $H(\mathsf{pw})$ otherwise. We use an hash instead of

[8] OIDC is generated in the form of a JSON Web Token (JWT) as a signature on a "nonce" along with other parameters for user identification.

a PRF. It is because PRFs require an additional round-trip communication in order to evaluate them in an oblivious manner.

The client runs two algorithms in the $\mathsf{Consent}$ phase in Figure 6: Launch, and Commit. Launch algorithm requires the client to log in to the IdM every time the client consents to the tasks to be run by the Agent and to enter pw. This inevitably creates some friction for the user and workload for the IdM. Instead, the client can request an order in batch to consent to the tasks. It works as follows: the client generates a short-living secret/public key pair, it sets this short term public key as an *att* in the Launch algorithm to get it signed by the IdM. Later when the client wants to order a task from the Agent, it can use any attribute it needs as att_i which is signed with the short-living secret key and appends this signature to the order with signature on the public key from the IdM. This allows batch consent generation with less interaction with the IdM. Such approach is already used by zkLogin [Bal+24].

Another way to see this approach is to take the consent on *att* as a sub-contract $\mathsf{contract}'$ where s_{A} is set to the obtained consent. This contract registers a new IdM' which is run by the client and a password pw set to void. Hence, resp is actually a signature on att_i. We set $\mathsf{order} = \mathsf{resp}$ in the Commit algorithm. The consent on att_i becomes the order_i concatenated with s_{A}.

5 Applications and Variants

5.1 Use Cases

In our consent protocol, Client aims at giving consent to an Agent to perform an action defined by *att*. We discuss here how applications can use the consent. We consider three types of applications.

Type I. Applications where nothing prevents the action from being executed by the Agent even when there is no consent but any such action is visible by the Client and provable to a Judge. Here, the $\mathsf{contract}$ should also be signed by the Agent to be undeniable to the Judge. That is, there is a trivial dispute protocol which can be triggered by the Client because the Client can easily prove that the action was done by the Agent without their consent. This would require the Agent to keep a log of all $(att, \mathsf{consent})$ associated to $\mathsf{contract}$ for possible disputes for a short time. [9]

Type II. Applications where the action leaves no evidence, but the Agent is split in two parts: a frontend part which runs the consent protocol and a backend part which executes the action. The backend must be trusted (as being a smart contract or a TEE) to execute the action only when the consent is shown.

Type III. Applications where the action cannot be executed without consent by design because it is not valid without the consent. It is not necessary to keep logs. Typically, the action cannot be validated or executed without a valid consent.

[9] The Client will be given a time interval for the dispute.

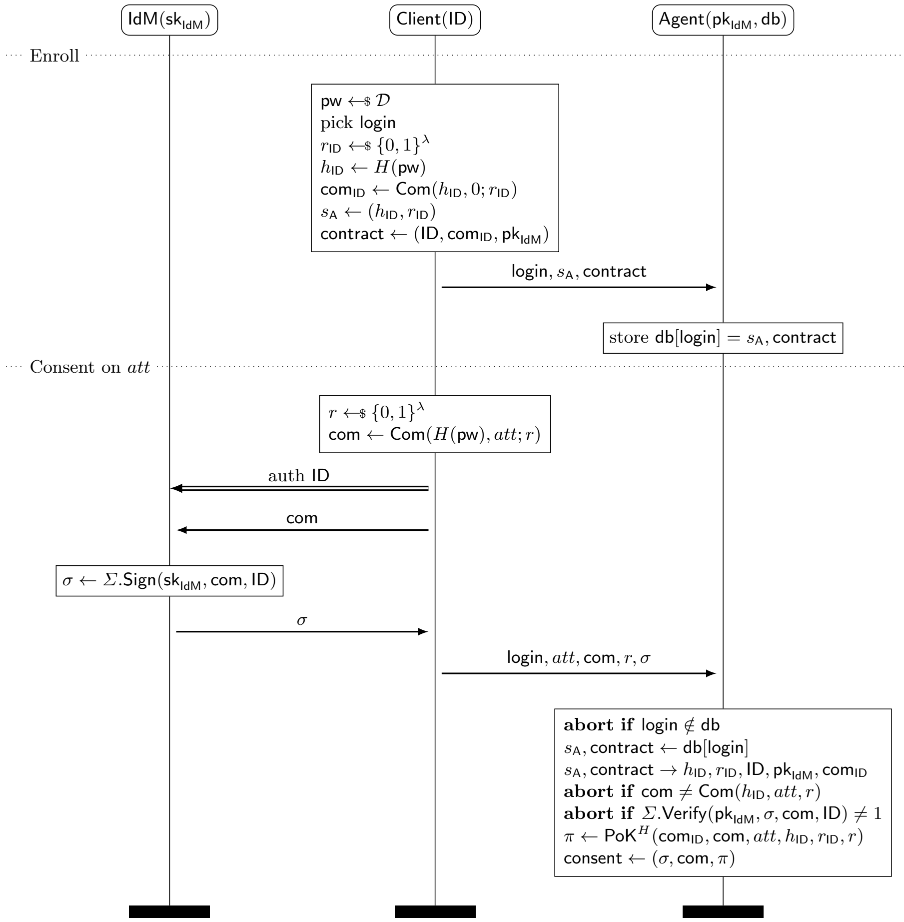

Fig. 5. Consent Protocol with $PoK\{(h_{\mathsf{ID}}, r_{\mathsf{ID}}, r) : \mathsf{com}_{\mathsf{ID}} = \mathsf{Com}(h_{\mathsf{ID}}, 0; r_{\mathsf{ID}}) \land \mathsf{com} = \mathsf{Com}(h_{\mathsf{ID}}, att; r)\}$

The distinction with Type II is that the validation of the action happens outside of Agent. It is done by an independent service executing the action or by other participants who must verify the legitimacy of the action.

In Type I and Type II applications, it is easy to have some form of resilience for credential losses. A Client with lost pw or credentials to authenticate to IdM (or unavailable IdM) can activate a rescue process to define a new contract. This process must be cumbersome to prevent from being maliciously used.

Signing Service. As an example, we propose a service by which an Agent digitally signs documents on behalf of the Client by holding the signing key of the Client. In this way, the Agent effectively has the power of attorney.

$\mathsf{Enroll}(\mathsf{ID}, \mathsf{pw}, \mathsf{pk}_{\mathsf{IdM}})$

- $r_{\mathsf{ID}} \leftarrow\!\!\$\ \{0,1\}^\lambda$
- $h_{\mathsf{ID}} \leftarrow H(\mathsf{pw})$
- $\mathsf{com}_{\mathsf{ID}} \leftarrow \mathsf{Com}(h_{\mathsf{ID}}, 0; r_{\mathsf{ID}})$
- $s_{\mathsf{A}} \leftarrow (h_{\mathsf{ID}}, r_{\mathsf{ID}})$
- $\mathsf{contract} \leftarrow (\mathsf{ID}, \mathsf{com}_{\mathsf{ID}}, \mathsf{pk}_{\mathsf{IdM}})$
- **return** $s_{\mathsf{A}}, \mathsf{contract}$

$\mathsf{Launch}(\mathsf{pw}, att)$

- $r \leftarrow\!\!\$\ \{0,1\}^\lambda$
- $\mathsf{com} \leftarrow \mathsf{Com}(H(\mathsf{pw}), att; r)$
- $\mathsf{query} \leftarrow \mathsf{com}$
- $\mathsf{state} \leftarrow (r, \mathsf{com})$
- **return** $\mathsf{query}, \mathsf{state}$

$\mathsf{IdM}(\mathsf{ID}, \mathsf{sk}_{\mathsf{IdM}}, \mathsf{query})$

- $\mathsf{query} \rightarrow \mathsf{com}$
- $\mathsf{resp} \leftarrow \Sigma.\mathsf{Sign}(\mathsf{sk}_{\mathsf{IdM}}, \mathsf{com}, \mathsf{ID})$
- **return** resp

$\mathsf{Commit}(\mathsf{state}, \mathsf{resp})$

- $\mathsf{state} \rightarrow r, \mathsf{com}$
- $\mathsf{resp} \rightarrow \sigma$
- $\mathsf{order} \leftarrow \mathsf{com}, r, \sigma$
- **return** order

$\mathsf{Agent}(s_{\mathsf{A}}, \mathsf{contract}, att, \mathsf{order})$

- $\mathsf{order} \rightarrow \mathsf{com}, r, \sigma$
- $s_{\mathsf{A}}, \mathsf{contract} \rightarrow h_{\mathsf{ID}}, r_{\mathsf{ID}}, \mathsf{ID}, \mathsf{com}_{\mathsf{ID}}, \mathsf{pk}_{\mathsf{IdM}}$
- **abort if** $\mathsf{com} \neq \mathsf{Com}(h_{\mathsf{ID}}, att, r)$
- **abort if** $\Sigma.\mathsf{Verify}(\mathsf{pk}_{\mathsf{IdM}}, \sigma, \mathsf{com}, \mathsf{ID}) \neq 1$
- $\pi \leftarrow \Pi_{\mathsf{MULTEQ}}.\mathsf{PoK}(\mathsf{com}_{\mathsf{ID}}, \mathsf{com}, att, h_{\mathsf{ID}}, r_{\mathsf{ID}}, r)$
- $\mathsf{consent} \leftarrow (\sigma, \mathsf{com}, \pi)$

$\mathsf{Verify}(\mathsf{contract}, att, \mathsf{consent})$

- $\mathsf{consent} \rightarrow (\sigma, \mathsf{com}, \pi)$
- $\mathsf{contract} \rightarrow (\mathsf{ID}, \mathsf{com}_{\mathsf{ID}}, \mathsf{pk}_{\mathsf{IdM}})$
- **return** false **if** $\Sigma.\mathsf{Verify}(\mathsf{pk}_{\mathsf{IdM}}, \sigma, \mathsf{com}, \mathsf{ID}) \neq 1$
- **return** $\Pi_{\mathsf{MULTEQ}}.\mathsf{Verify}(\pi, \mathsf{com}_{\mathsf{ID}}, \mathsf{com}, att)$

Fig. 6. Consent Protocol

In this application, the att value can be the hash of the document to be signed, so that Agent does not even see the document itself. [10] For signed documents based on the sign-and-hash paradigm, nothing needs to be changed on the side of the signature verifiers.

For liability reasons, the Agent must be the only repository of the signing key, even if the key is connected to the Client. So, any valid signature must have been created by the Agent. Assuming that signed documents eventually come back to Client, we are in the case of Type I.

More precisely, we assume that Agent signs att together with ID to notify that it is a signature on behalf of ID. The enrollment of a new user would work as follows. The user triggers our enrollment protocol to define a new contract. The Client also launches the consent protocol with att set to a special setup message with a reference to contract. This produces a consent assuring that contract was set by the ID owner, as certified by IdM. The Agent signs $(\mathsf{contract}, \mathsf{consent})$ to get a signature σ and sets $\mathsf{cert} = (\mathsf{contract}, \mathsf{consent}, \sigma)$. Then, cert is returned to the Client. The Client and the Agent keep cert for reference. This cert can be given to a signature verifier to show that Agent has the power of attorney for ID. It can also be given to the Judge to prove that Agent took the responsibility of a signing key for ID and committed to keep valid consents for each valid signature.

[10] This is compliant with how Adobe signs PDFs with RSA-PSS.

Note that a malicious IdM can impersonate the Client to register a contract. Hence, the Agent should never register two contracts with the same ID. Assuming that signed documents come back to the Client, it will be visible if the IdM registered a contract and the Client did not.

If signed documents do not come back to the Client, we are in Type II and we must use a trusted Agent backend.

If we allow modifications on signature verifiers, we can declare that verification of a signature must also verify an existing consent. In that case, we obtain an application of Type III.

Bank Transaction. In e-banking applications, the Agent is the bank which is executing orders on behalf of the Client. Actions can be payments or other transactions on the account of the Client. It is important for liability reasons that the Agent keeps logs with evidence of their consents.

With a traditional bank, the Client would clearly see on their balance that an action was made. We are in a Type I case.

With crypto-currencies, we can mandate that actions require a consent, to be verified by a smart contract on which contract was set up. We are in a Type III case. This generalizes to other services based on smart contracts.

The zkLogin [Bal+24] is offering a similar functionality. It uses an OpenID Connect provider as (IdM), a *salt server* (comparable to Agent), and assumes that they are not corrupted at the same time. They also need a ZKP service to offload the computation of the proof (the consent). In zkLogin, the *salt* is some Client-specific secret which is comparable to s_{A}.

Cryptographic Secret Key Storage. Instead of signing or authorizing, the Agent could be a service to keep secret keys safe. The action would be an order to retrieve one specific key. Here, the Client cannot control if a key is retrieved without consent so we are in a Type II case and we need a trusted backend.

Key storage can be used as a safe repository, like another banking service. It can be used to access to a crypto wallet or a password repository.

5.2 Variants for IdM

So far, in the paper, IdM is treated as an abstract *identity manager* but it could be more general than a regular identity provider. Here, we discuss possible instances.

Identity provider IdM. In the Single-Sign-On (SSO) design, services can rely on a common and independent identity provider. In the OpenID Connect (OIDC) protocol, this service provides access tokens which are signed by the identity provider together with some identity ID. The signature is made on a nonce which is provided by the client. Hence, this perfectly matches to our proposed protocol.

Composite IdM. Instead of relying on a single IdM, we could treat IdM as a composition of several IdM_1, IdM_2 with a specific composition rule. Hence, $\mathsf{pk}_{\mathsf{IdM}} = (\mathsf{pk}_{\mathsf{IdM}_1}, \mathsf{pk}_{\mathsf{IdM}_2})$.

For instance, if $\mathsf{IdM} = \mathsf{IdM}_1 \wedge \mathsf{IdM}_2$ (the AND composition), a valid order must be endorsed by both IdM_1 and IdM_2 to be valid. One benefit is that we would need to rely *less* on $\mathsf{CUF}_{\mathsf{IdM}}$ security as it is unlikely that both IdM_1 and IdM_2 would be corrupted.

If $\mathsf{IdM} = \mathsf{IdM}_1 \vee \mathsf{IdM}_2$ (the OR composition), a valid order must be endorsed by either IdM_1 or IdM_2 to be valid. It makes $\mathsf{CUF}_{\mathsf{IdM}}$ more essential but also makes the functionality of our protocol more reliable as Client would be able to handle the unavailability of either IdM_1 or IdM_2.

Any other monotone composition rule can be imagined.

Smartphone IdM. A client may use their smartphone as an IdM which provides identity to a single client. In that case, we may wonder why not using a trivial consent protocol where the smartphone directly signs a consent. However, this approach is not secure in the sense of $\mathsf{CUF}_{\mathsf{IdM}}$ (if the smartphone is corrupted, then consents may be forged). Also, this type of IdM may find more sense when combined with others. The AND composition may capture the notion of multifactor authentication. The OR composition may allow the client to still give consents when their smartphone is lost, stolen, or broken.

Secure hardware IdM. In general, any personal secure hardware to which we can send a nonce and which returns a signature of it can be used *in lieu* of our IdM.

Using biometry in IdM. Adding a virtual IdM which authenticates the Client based on biometry is trivial. This could be done by the smartphone. As biometric systems may have false non-matches, it is advisable to combine it with another modality.

Password-based IdM. The fact that our protocol uses a password pw can actually be seen as an extra IdM where the signature of com is π and the public key to verify it is $\mathsf{com}_{\mathsf{ID}}$. To sign, this virtual IdM uses the secret s_{A}.[11] Our proposed protocol can be seen as an AND composition of a regular IdM with this password-based one. This virtual IdM is integrated in Agent.

E-passport-based IdM. We can build an IdM which authenticates the Client by their e-passport, following the ICAO MRTD standard. Assuming that Chip Authentication is implemented with the PACE-CAM protocol, the IdM can remotely interact with the IC chip of the passport, go through passive authentication: get the certificate to have the public verification key, read the Security Object of the Document (SOD), read the DG1 and DG14 files containing the IC

[11] Technically, the virtual IdM makes a pre-proof $(\mathsf{ch}, \mathsf{resp}_1, \mathsf{resp}_2, c)$ and π is finished by setting $\mathsf{resp}_3 = c + \mathsf{ch} \cdot r$.

public key, authenticate it using the previous information, then run the PACE-CAM protocol to prove to IdM that it is interacting with the e-passport, then deduce from DG1 the identity of Client. After this, IdM may sign an OIDC token.

Unfortunately, it does not seem possible to have the e-passport to directly sign an OIDC token. This could be done with the Active Authentication (AA) protocol but it is not commonly available and probably outdated. It seems that we need a trusted intermediary.

5.3 Variants for Agent

Password-less Agent. We can replace pw to authenticate to the Agent by a constant. In that case, $\mathsf{com}_{\mathsf{ID}}$ is no longer needed in contract and s_{A} is void. The NIZK proof can be replaced by $\pi = r$. Consequently, we no longer have $\mathsf{CUF}_{\mathsf{IdM}}$ security: a corrupted IdM can impersonate the Client. This is fine if we can rely on the integrity of IdM, for instance when we use a composite IdM to make it stronger. However, the $\mathsf{CUF}_{\mathsf{Agent}}$ security reduces directly to the unforgeability of σ and the binding security of com:

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{CUF}_{\mathsf{Agent}}}(\lambda) \leq \mathsf{Adv}_{\mathcal{B}}^{\mathsf{UF}}(\lambda) + \mathsf{Adv}_{\mathcal{C}}^{\mathsf{BIND}}(\lambda)$$

This is better than Theorem 2.

We can further replace com by *att* and rely on the unforgeability of σ only. However, the honest (but curious) IdM would see *att*.

Biometric-based Agent. We can replace pw to authenticate to the Agent by a biometry. Enrollment would define a biometric template to be put in s_{A} and contract would commit to it. Then, consent issuance would commit to a biometric probe and consent would prove that both commitments commit to matching templates. It makes π much more complex. Another option (which is not advisable for privacy reasons) is to let the template in contract and the probe in consent in clear. But, this implies to keep both contract and consent private for a Judge in the case of a dispute.

6 Implementation

We implemented[12] our digital consent scheme in Rust with Ristretto group using curve25519-dalek library (SIMD and BasePointTable optimizations disabled). We implement Com as Pedersen Commitment with multiscalar multiplication. We use eddsa25519-dalek for Σ.Sign and Σ.Verify which is compliant with OIDC JWTs. We use SHA256 for H. The benchmarks have been obtained using the Criterion.rs statistical benchmarking suite on a laptop with Apple M1 processor. We report the benchmark results in Table 1.

[12] The source code is available at: `https://github.com/bufferhe4d/consent`

Table 1. Benchmark Results (with *att* of 1KB).

	Client.Enroll	Client.Launch	IdM	Agent	Verify
Running Time (μs)	94.63	146.75	23.58	387.64	323.87

Note on real world deployments: OpenPubKey [Hei+23] introduced the clever idea of using existing OIDC providers to sign ephemeral metadata (a public key in their case) by replacing the nonce field of the OIDC token with a public key. This idea is also used by zkLogin [Bal+24] and Aptos Keyless Accounts[13] where both applications replace the nonce field with an ephemeral public key to sign transactions on a blockchain. Similarly in our case, we are able to replace the nonce field with a commitment to the hash of a password and the attribute we are giving a consent to. Hence, the reason that our IdM only having a single signing operation is by design and aims to facilitate this out of the box compatibility with existing OIDC providers. Note that compared to zkLogin and Aptos Keyless Accounts, the relation we prove on the Agent side is a simple sigma protocol compared to a generic zk-SNARK.

7 Security of Our Protocol

We provide here security results as well as sketches of proofs. The full proofs are provided in section A.

Theorem 1. *Let* Com *be an additively homomorphic, computationally binding and perfectly hiding commitment scheme,* H *be a hash function with output domain* $\mathbb{Z}_q$*,* $\mathcal{D}_{min}$ *be the min-entropy of* $\mathsf{Com}(H(\mathsf{pw}), 0; 0)$ *and* Π_{MULTEQ} *be our NIZK proof in the random oracle model. For any* PPT *adversary* $\mathcal{A}$ *playing the* $\mathsf{CUF}_{\mathit{IdM}}$ *game, the following advantage against the construction on Figure 5*

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{CUF}_{\mathit{IdM}}}(\lambda, \mathcal{D}) \leq \frac{(q_{\mathsf{H}} + q_{\mathsf{Com}} + q_{\mathsf{Ord}}) \cdot (q_{\mathsf{Com}} + q_{\mathsf{Ord}})}{q^2} + \frac{q_{\mathsf{Ord}}}{2^{\mathcal{D}_{min}}} + \frac{q_{\mathsf{H}} + q_{\mathsf{Com}} + q_{\mathsf{Ord}}}{q} + \sqrt{\frac{q_{\mathsf{H}} + q_{\mathsf{Com}} + q_{\mathsf{Ord}}}{q} \cdot \mathsf{Adv}_{\mathcal{B}}^{\mathsf{BIND}}(\lambda)}$$

where $\mathcal{B}$ *is an adversary playing the binding-security game against* Com*,* q_{H}*,* q_{Com}*, and* q_{Ord} *are the number of queries for NIZK to the random oracle* H*, to* OCommit*, and* OOrder*, respectively.*

In practice, is it reasonable to assume that the min-entropy of $\mathsf{Com}(H(\mathsf{pw}), 0; 0)$ is the same as the min-entropy of pw, as it is unlikely what we have a collision on $\mathsf{pw} \mapsto \mathsf{Com}(H(\mathsf{pw}), 0; 0)$ in the password domain.

Proof (Sketch of proof). We first use the zero-knowledge property of π by simulating the proofs, adding an oracle to program the random oracle. Then, we

[13] `https://aptos.dev/en/build/guides/aptos-keyless`

realize that commitments can be replaced by random ones to remove some usages of pw. The last part which still uses pw is the OOrder oracle when verifying that com commits to h_{ID}. We simulate it by an extra test oracle which verifies whether a guess for $\mathsf{com}(h_{\mathsf{ID}}, 0; 0)$ is correct. Then, we can use the min-entropy of $\mathsf{com}(h_{\mathsf{ID}}, 0; 0)$ to replace OOrder by an oracle which always rejects. We then use the knowledge soundness of π to get openings of commitments. The game then boils down to a binding game in the commitment. See section A for details.

Theorem 2. *Let* Com *be an additively homomorphic, computationally binding commitment scheme,* Σ *be a signature scheme which is secure against existential forgeries,* H *be a hash function with output domain* $\mathbb{Z}_q$*, and* Π_{MULTEQ} *be our NIZK proof in the random oracle model. For any* PPT *adversary* $\mathcal{A}$ *playing the* CUF_{Agent} *game, the following advantage against the construction on Figure 5*

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{CUF}_{Agent}}(\lambda) \leq \mathsf{Adv}_{\mathcal{B}}^{\mathsf{UF}}(\lambda) + \frac{q_{\mathsf{H}}}{q} + \sqrt{\frac{q_{\mathsf{H}}}{q} \cdot \mathsf{Adv}_{\mathcal{C}}^{\mathsf{BIND}}(\lambda)}$$

where $\mathcal{B}$ *is an adversary playing the existential forgery game against* Σ*,* $\mathcal{C}$ *is an adversary playing the binding-security game against* Com*, and* q_{H} *and the number of queries for NIZK to the random oracle* H*.*

Proof (Sketch of proof). We first use the unforgeability of the signature scheme in order to reduce to a game where the final consent was signed by the $\mathsf{OLaunch}$ oracle. The signature holds on a commitment. We use the knowledge soundness of π in order to get opening information on the commitment, and deduce a breach in the binding security of the commitment. See section A for details.

8 Conclusion

In this work, we have introduced the concept of cryptographically secure digital consent, aiming to replicate traditional consent processes in the digital realm. Our proposed framework involves key participants such as an Identity Manager, ensuring compatibility with existing systems and addressing various security and privacy challenges. By leveraging existing cryptographic operations and avoiding long-term client-side storage, our protocol provides a flexible and secure solution for digital consent, accommodating different use cases and ensuring the integrity of the consent process.

References

[ABK13] Tolga Acar, Mira Belenkiy, and Alptekin Küpçü. "Single password authentication". In: *Computer Networks* 57.13 (2013), pp. 2597–2614. ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2013.05.007. URL: https://www.sciencedirect.com/science/article/pii/S1389128613001667.

[AN23] Ghada Almashaqbeh and Anca Nitulescu. *Anonymous, Timed and Revocable Proxy Signatures*. Cryptology ePrint Archive, Paper 2023/833. 2023. URL: `https://eprint.iacr.org/2023/833`.

[Bal+24] Foteini Baldimtsi et al. *zkLogin: Privacy-Preserving Blockchain Authentication with Existing Credentials*. 2024. eprint: `2401.11735`. URL: `https://arxiv.org/abs/2401.11735`.

[Boy09] Xavier Boyen. "Hidden Credential Retrieval from a Reusable Password". In: *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*. ASIACCS '09. Sydney, Australia: Association for Computing Machinery, 2009, pp. 228–238.

[Cam+14] Jan Camenisch et al. "Memento: How to Reconstruct Your Secrets from a Single Password in a Hostile Environment". In: *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part II*. Ed. by Juan A. Garay and Rosario Gennaro. Vol. 8617. Lecture Notes in Computer Science. Springer, 2014, pp. 256–275. DOI: `10.1007/978-3-662-44381-1\_15`. URL: `https://doi.org/10.1007/978-3-662-44381-1%5C_15`.

[Cam+16] Jan Camenisch et al. "Virtual Smart Cards: How to Sign with a Password and a Server". In: *Proceedings of the 10th International Conference on Security and Cryptography for Networks - Volume 9841*. Berlin, Heidelberg: Springer-Verlag, 2016, pp. 353–371. ISBN: 9783319446172. DOI: `10.1007/978-3-319-44618-9_19`. URL: `https://doi.org/10.1007/978-3-319-44618-9_19`.

[CL01] Jan Camenisch and Anna Lysyanskaya. "An Efficient System for Non-transferable Anonymous Credentials with Optional Anonymity Revocation". In: *Advances in Cryptology — EUROCRYPT 2001*. Ed. by Birgit Pfitzmann. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 93–118. ISBN: 978-3-540-44987-4.

[Dav+23] Gareth T. Davies et al. "Security Analysis of the WhatsApp End-to-End Encrypted Backup Protocol". In: *Advances in Cryptology – CRYPTO 2023: 43rd Annual International Cryptology Conference, CRYPTO 2023, Santa Barbara, CA, USA, August 20–24, 2023, Proceedings, Part IV*. Santa Barbara, CA, USA: Springer-Verlag, 2023, pp. 330–361. ISBN: 978-3-031-38550-6. DOI: `10.1007/978-3-031-38551-3_11`. URL: `https://doi.org/10.1007/978-3-031-38551-3_11`.

[DHS14] David Derler, Christian Hanser, and Daniel Slamanig. "Privacy-Enhancing Proxy Signatures from Non-Interactive Anonymous Credentials". In: *Data and Applications Security and Privacy XXVIII*. Springer Berlin Heidelberg, 2014, pp. 49–65.

[DL24] Dennis Dayanikli and Anja Lehmann. "Password-Based Credentials with Security Against Server Compromise". In: Springer-Verlag, 2024, pp. 147–167. ISBN: 978-3-031-50593-5.

[Doc24] DocuSign. *Security for DocuSign eSignature*. 2024. URL: `https://www.docusign.com/trust/security/esignature`.

[Fal+24] Sebastian Faller et al. *Password-Protected Key Retrieval with(out) HSM Protection*. Cryptology ePrint Archive, Paper 2024/1384. To appear at ACM CCS 2024. 2024. URL: `https://eprint.iacr.org/2024/1384`.

[Fau+12] Sebastian Faust et al. "On the Non-Malleability of the Fiat-Shamir Transform". In: *Progress in Cryptology - INDOCRYPT 2012*. Springer Berlin Heidelberg, 2012, pp. 60–79.

[FP09] Georg Fuchsbauer and David Pointcheval. "Anonymous Consecutive Delegation of Signing Rights: Unifying Group and Proxy Signatures". In: *Formal to Practical Security: Papers Issued from the 2005-2008 French-Japanese Collaboration*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 95–115.

[FS86] Amos Fiat and Adi Shamir. "How to Prove Yourself: Practical Solutions to Identification and Signature Problems". In: *Advances in Cryptology - CRYPTO '86*. Vol. 263. Lecture Notes in Computer Science. Springer, 1986, pp. 186–194.

[GT12] Kristian Gjøsteen and Øystein Thuen. "Password-Based Signatures". In: *Public Key Infrastructures, Services and Applications*. Springer Berlin Heidelberg, 2012, pp. 17–33.

[Hei+23] Ethan Heilman et al. *OpenPubkey: Augmenting OpenID Connect with User held Signing Keys*. Cryptology ePrint Archive, Paper 2023/296. 2023. URL: `https://eprint.iacr.org/2023/296`.

[HS13] Christian Hanser and Daniel Slamanig. "Blank Digital Signatures". In: *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*. ASIA CCS '13. New York, NY, USA: Association for Computing Machinery, 2013, pp. 95–106. ISBN: 9781450317672.

[HWF05] Yong-Zhong He, Chuan-Kun Wu, and Deng-Guo Feng. "Server-Aided Digital Signature Protocol Based on Password". In: *Proceedings 39th Annual 2005 International Carnahan Conference on Security Technology*. 2005, pp. 89–92.

[JKR13] Sangeetha Jose, Preetha Mathew K., and C. Pandu Rangan. *Strongly Secure Password Based Blind Signature for Real World Applications*. 2013.

[MRS10] Atsuko Miyaji, Mohammad Shahriar Rahman, and Masakazu Soshi. "Hidden Credential Retrieval without Random Oracles". In: *Web Information System and Application Conference*. 2010. URL: `https://api.semanticscholar.org/CorpusID:46197738`.

[Sch89] Claus-Peter Schnorr. "Efficient Identification and Signatures for Smart Cards". In: *Advances in Cryptology - EUROCRYPT '89, Workshop on the Theory and Application of of Cryptographic Techniques*. Vol. 434. Lecture Notes in Computer Science. 1989, pp. 688–689.

[Sch91] Claus-Peter Schnorr. "Efficient Signature Generation by Smart Cards". In: *Journal of Cryptology* 4.3 (1991), pp. 161–174.

Supplementary Materials

A Security of Our Protocol

A.1 $\mathsf{CUF_{IdM}}$ Security: Proof of Theorem 1

Proof. For the games we construct during the proof, we denote the *i-th* game with $\mathsf{G}i$. We set $\mathsf{G0} = \mathsf{CUF_{IdM}}$.

G1 (Figure 7): We expand the subprocedures Launch, Commit and Agent. We have,

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{G0}} = \mathsf{Adv}_{\mathcal{A}}^{\mathsf{G1}}$$

G2 (Figure 8): Since all the enrolled users except the victim enrolled with login^* are corrupted, if $\mathsf{login} \neq \mathsf{login}^*$ OOrder is simulatable by the adversary $\mathcal{A}_2$, hence we rewrite it with login^* only. We also remove line 2 of Agent in $\mathsf{OCommit}$ in $\mathsf{G1}$ as it always passes the check. We have,

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{G1}} = \mathsf{Adv}_{\mathcal{A}_2}^{\mathsf{G2}}$$

G3 (Figure 9): We drop the db structure along with the $\mathsf{OCorruptEnroll}$ oracle and the $\mathsf{enrolled}$ set, as the corrupted enrollments are not used anymore and simulate these calls in $\mathcal{A}_3$. We have,

$$\mathsf{Adv}_{\mathcal{A}_2}^{\mathsf{G2}} = \mathsf{Adv}_{\mathcal{A}_3}^{\mathsf{G3}}$$

G4 (Figure 10): We replace π in $\mathsf{OCommit}$ and OOrder with a simulated proof by equipping the technique used in [Fau+12]. The technique works as follows: First we introduce a lazy sampling table LH along with two oracles S_1 which simulates the random oracle using LH and S_2 which on input a valid statement x and witness w, checks the validity and returns a simulated proof while programming LH respectively. We use the fact that Π_{MULTEQ} is perfect zero-knowledge, as discussed in subsection 4.2. The advantage loss is introduced by the abort cases in S_1 and S_2. For each new query to S_2 and each existing LH entry, the probability to match (so to abort) is the probability of a collision on the random selection of $\alpha = (\mathsf{com}_1, \mathsf{com}_2)$. There are up to $q_{S_1} + q_{S_2}$ entries in LH, and q_{S_2} queries to S_2. We have:

$$\mathsf{Adv}_{\mathcal{A}_3}^{\mathsf{G3}} \leq \mathsf{Adv}_{\mathcal{A}_3}^{\mathsf{G4}} + \frac{(q_{\mathsf{S}_1} + q_{\mathsf{S}_2}) \cdot q_{\mathsf{S}_2}}{q^2}$$

```
G1(A, D)
given ← ∅
cst ← {}    // challenge state
pk_IdM, st, login*, ID* ← A_1(1^λ)
pw* ←$ D
enrolled ← {login*}
s*_A, contract* ← Enroll(ID*, pw*, pk_IdM)
db[login*] ← s*_A, contract*
A_2^oracles(contract*, st) → att*, consent*
if Verify(contract*, att*, consent*)
 ∧ consent* ∉ given :
   return 1
return 0
```

```
OCorruptEnroll(login, s_A, contract)
if login ∈ enrolled
   abort
enrolled ← enrolled ∪ login
db[login] ← s_A, contract
return ⊥
```

```
OLaunch(sid, att)
if sid ∈ cst
   abort
Launch
   r ←$ {0,1}^λ
   com ← Com(H(pw*), att; r)
   query ← com
   state ← (r, com)
cst[sid].att ← att
cst[sid].state ← state
return query
```

```
OCommit(sid, resp)
if sid ∉ cst
   abort
Commit
   cst[sid].state → r, com
   cst[sid].att → att
   resp → σ
   order ← com, r, σ
remove sid from cst
Agent
   s*_A, contract* → h_ID, r_ID, pk_IdM, com_ID
   return false if com ≠ Com(h_ID, att, r)
   return false if Σ.Verify(pk_IdM, σ, com) ≠ 1
   π ← Π_MULTEQ.PoK^H(com_ID, com, att, h_ID, r_ID, r)
   consent ← (σ, com, π)
given ← given ∪ consent
return consent
```

```
OOrder(login, att, order)
db[login] → s_A, contract
order → com, r, σ
Agent
   s_A, contract → h_ID, r_ID, pk_IdM, com_ID
   return false if com ≠ Com(h_ID, att, r)
   return false if Σ.Verify(pk_IdM, σ, com) ≠ 1
   π ← Π_MULTEQ.PoK^H(com_ID, com, att, h_ID, r_ID, r)
   consent ← (σ, com, π)
return consent
```

Fig. 7. $\mathsf{CUF}_{\mathsf{IdM}}$ G1

$\mathsf{G2}(\mathcal{A}_2, \mathcal{D})$

given $\leftarrow \emptyset$
cst $\leftarrow \{\}$ // challenge state
$\mathsf{pk}_{\mathsf{IdM}}, \mathsf{st}, \mathsf{login}^*, \mathsf{ID}^* \leftarrow \mathcal{A}_{2,1}(1^\lambda)$
$\mathsf{pw}^* \leftarrow\$ \mathcal{D}$
enrolled $\leftarrow \{\mathsf{login}^*\}$
$s_{\mathsf{A}}^*, \mathsf{contract}^* \leftarrow \mathsf{Enroll}(\mathsf{ID}^*, \mathsf{pw}^*, \mathsf{pk}_{\mathsf{IdM}})$
$\mathsf{db}[\mathsf{login}^*] \leftarrow s_{\mathsf{A}}^*, \mathsf{contract}^*$
$\mathcal{A}_{2,2}^{\text{oracles}}(\mathsf{contract}^*, \mathsf{st}) \rightarrow att^*, \mathsf{consent}^*$
if $\mathsf{Verify}(\mathsf{contract}^*, att^*, \mathsf{consent}^*)$
$\wedge\ \mathsf{consent}^* \notin \mathsf{given}$:
return 1
return 0

$\mathsf{OCorruptEnroll}(\mathsf{login}, s_{\mathsf{A}}, \mathsf{contract})$

if login $\in$ enrolled
abort
enrolled $\leftarrow$ enrolled $\cup$ login
$\mathsf{db}[\mathsf{login}] \leftarrow s_{\mathsf{A}}, \mathsf{contract}$
return $\perp$

$\mathsf{OLaunch}(sid, att)$

if $sid \in$ cst
abort
$r \leftarrow\$ \{0,1\}^\lambda$
$\mathsf{com} \leftarrow \mathsf{Com}(H(\mathsf{pw}^*), att; r)$
query $\leftarrow$ com
state $\leftarrow (r, \mathsf{com})$
$\mathsf{cst}[sid].att \leftarrow att$
$\mathsf{cst}[sid].\mathsf{state} \leftarrow \mathsf{state}$
return query

$\mathsf{OCommit}(sid, \mathsf{resp})$

if $sid \notin$ cst
abort
$\mathsf{cst}[sid].\mathsf{state} \rightarrow r, \mathsf{com}$
$\mathsf{cst}[sid].att \rightarrow att$
$\mathsf{resp} \rightarrow \sigma$
remove sid from cst
$s_{\mathsf{A}}^*, \mathsf{contract}^* \rightarrow h_{\mathsf{ID}}, r_{\mathsf{ID}}, \mathsf{pk}_{\mathsf{IdM}}, \mathsf{com}_{\mathsf{ID}}$
return false **if** $\Sigma.\mathsf{Verify}(\mathsf{pk}_{\mathsf{IdM}}, \sigma, \mathsf{com}) \neq 1$
$\pi \leftarrow \Pi_{\mathsf{MULTEQ}}.\mathsf{PoK}^H(\mathsf{com}_{\mathsf{ID}}, \mathsf{com}, att, h_{\mathsf{ID}}, r_{\mathsf{ID}}, r)$
$\mathsf{consent} \leftarrow (\sigma, \mathsf{com}, \pi)$
given $\leftarrow$ given $\cup$ consent
return consent

$\mathsf{OOrder}(att, \mathsf{order})$

$\mathsf{order} \rightarrow \mathsf{com}, r, \sigma$
$\boxed{s_{\mathsf{A}}^*, \mathsf{contract}^*} \rightarrow h_{\mathsf{ID}}, r_{\mathsf{ID}}, \mathsf{pk}_{\mathsf{IdM}}, \mathsf{com}_{\mathsf{ID}}$
return false **if** $\mathsf{com} \neq \mathsf{Com}(h_{\mathsf{ID}}, att, r)$
return false **if** $\Sigma.\mathsf{Verify}(\mathsf{pk}_{\mathsf{IdM}}, \sigma, \mathsf{com}) \neq 1$
$\pi \leftarrow \Pi_{\mathsf{MULTEQ}}.\mathsf{PoK}^H(\mathsf{com}_{\mathsf{ID}}, \mathsf{com}, att, h_{\mathsf{ID}}, r_{\mathsf{ID}}, r)$
$\mathsf{consent} \leftarrow (\sigma, \mathsf{com}, \pi)$
return consent

Fig. 8. $\mathsf{CUF}_{\mathsf{IdM}}$ G2

G3($\mathcal{A}_3, \mathcal{D}$)

```
given ← ∅
cst ← {}    // challenge state
pk_IdM, st, login*, ID* ← A_{3,1}(1^λ)
pw* ←$ D
s*_A, contract* ← Enroll(ID*, pw*, pk_IdM)
A_{3,2}^{oracles}(contract*, st) → att*, consent*
if Verify(contract*, att*, consent*)
 ∧ consent* ∉ given :
   return 1
return 0
```

OLaunch(sid, att)

```
if sid ∈ cst
   abort
r ←$ {0,1}^λ
com ← Com(H(pw*), att; r)
query ← com
state ← (r, com)
cst[sid].att ← att
cst[sid].state ← state
return query
```

OCommit(sid, resp)

```
if sid ∉ cst
   abort
cst[sid].state → r, com
cst[sid].att → att
resp → σ
remove sid from cst
s*_A, contract* → h_ID, r_ID, pk_IdM, com_ID
return false if Σ.Verify(pk_IdM, σ, com) ≠ 1
π ← Π_MULTEQ.PoK^H(com_ID, com, att, h_ID, r_ID, r)
consent ← (σ, com, π)
given ← given ∪ consent
return consent
```

OOrder(att, order)

```
order → com, r, σ
s*_A, contract* → h_ID, r_ID, pk_IdM, com_ID
return false if com ≠ Com(h_ID, att, r)
return false if Σ.Verify(pk_IdM, σ, com) ≠ 1
π ← Π_MULTEQ.PoK^H(com_ID, com, att, h_ID, r_ID, r)
consent ← (σ, com, π)
return consent
```

Fig. 9. $\mathsf{CUF}_{\mathsf{IdM}}$ G3

$\mathsf{G4}(\mathcal{A}_3, \mathcal{D})$

$\mathsf{given} \leftarrow \emptyset$
$\mathsf{cst} \leftarrow \{\}$ // challenge state
$\mathsf{LH} \leftarrow \{\}$ // lazy sampling oracle
$\mathsf{pk}_{\mathsf{IdM}}, \mathsf{st}, \mathsf{login}^*, \mathsf{ID}^* \leftarrow \mathcal{A}_{3,1}(1^\lambda)$
$\mathsf{pw}^* \leftarrow\!\!\$\; \mathcal{D}$
$s^*_{\mathsf{A}}, \mathsf{contract}^* \leftarrow \mathsf{Enroll}(\mathsf{ID}^*, \mathsf{pw}^*, \mathsf{pk}_{\mathsf{IdM}})$
$\mathcal{A}_{3,2}^{\mathsf{O}*, \mathsf{S}_1}(\mathsf{contract}^*, \mathsf{st}) \to att^*, \mathsf{consent}^*$
$\mathbf{if}\ \mathsf{Verify}(\mathsf{contract}^*, att^*, \mathsf{consent}^*)$
$\quad \wedge\ \mathsf{consent}^* \notin \mathsf{given}:$
$\quad\quad \mathbf{return}\ 1$
$\mathbf{return}\ 0$

$\mathsf{OLaunch}(sid, att)$

$\mathbf{if}\ sid \in \mathsf{cst}$
$\quad \mathbf{abort}$
$r \leftarrow\!\!\$\; \{0,1\}^\lambda$
$\mathsf{com} \leftarrow \mathsf{Com}(H(\mathsf{pw}^*), att; r)$
$\mathsf{query} \leftarrow \mathsf{com}$
$\mathsf{state} \leftarrow (r, \mathsf{com})$
$\mathsf{cst}[sid].att \leftarrow att$
$\mathsf{cst}[sid].\mathsf{state} \leftarrow \mathsf{state}$
$\mathbf{return}\ \mathsf{query}$

$\mathsf{S}_2(x, w)$

$\mathbf{if}\ \mathcal{R}(x, w) = \bot$
$\quad \mathbf{return}\ \bot$
$\pi \leftarrow\!\!\$\; \Pi_{\mathsf{MULTEQ}}.\mathsf{Sim}(x)$
$\pi \to (\alpha, \mathsf{ch}, \gamma)$
$\mathbf{if}\ (x, \alpha) \in \mathsf{LH}$
$\quad \mathbf{abort}$
$\mathsf{LH}[(x, \alpha)] \leftarrow \mathsf{ch}$
$\mathbf{return}\ \pi$

$\mathsf{OCommit}(sid, \mathsf{resp})$

$\mathbf{if}\ sid \notin \mathsf{cst}$
$\quad \mathbf{abort}$
$\mathsf{cst}[sid].\mathsf{state} \to r, \mathsf{com}$
$\mathsf{cst}[sid].att \to att$
$\mathsf{resp} \to \sigma$
remove sid from cst
$s^*_{\mathsf{A}}, \mathsf{contract}^* \to h_{\mathsf{ID}}, r_{\mathsf{ID}}, \mathsf{pk}_{\mathsf{IdM}}, \mathsf{com}_{\mathsf{ID}}$
$\mathbf{return}\ \mathsf{false}\ \mathbf{if}\ \Sigma.\mathsf{Verify}(\mathsf{pk}_{\mathsf{IdM}}, \sigma, \mathsf{com}) \neq 1$
$\boxed{\pi \leftarrow \mathsf{S}_2((\mathsf{com}_{\mathsf{ID}}, \mathsf{com}, att), (h_{\mathsf{ID}}, r_{\mathsf{ID}}, r))}$
$\mathsf{consent} \leftarrow (\sigma, \mathsf{com}, \pi)$
$\mathsf{given} \leftarrow \mathsf{given} \cup \mathsf{consent}$
$\mathbf{return}\ \mathsf{consent}$

$\mathsf{OOrder}(att, \mathsf{order})$

$\mathsf{order} \to \mathsf{com}, r, \sigma$
$s^*_{\mathsf{A}}, \mathsf{contract}^* \to h_{\mathsf{ID}}, r_{\mathsf{ID}}, \mathsf{pk}_{\mathsf{IdM}}, \mathsf{com}_{\mathsf{ID}}$
$\mathbf{return}\ \mathsf{false}\ \mathbf{if}\ \mathsf{com} \neq \mathsf{Com}(h_{\mathsf{ID}}, att, r)$
$\mathbf{return}\ \mathsf{false}\ \mathbf{if}\ \Sigma.\mathsf{Verify}(\mathsf{pk}_{\mathsf{IdM}}, \sigma, \mathsf{com}) \neq 1$
$\boxed{\pi \leftarrow \mathsf{S}_2((\mathsf{com}_{\mathsf{ID}}, \mathsf{com}, att), (h_{\mathsf{ID}}, r_{\mathsf{ID}}, r))}$
$\mathsf{consent} \leftarrow (\sigma, \mathsf{com}, \pi)$
$\mathbf{return}\ \mathsf{consent}$

$\mathsf{S}_1(x, \alpha)$

$\mathbf{if}\ (x, \alpha) \notin \mathsf{LH}$
$\quad \mathsf{LH}[(x, \alpha)] \leftarrow\!\!\$\; \mathbb{Z}_q$
$\mathbf{return}\ \mathsf{LH}[(x, \alpha)]$

Fig. 10. $\mathsf{CUF}_{\mathsf{IdM}}$ G4

G5 (Figure 11): We observe that since S_2 is only accessed through $\mathsf{OCommit}$ and OOrder the instance witness pair (x, ω) will always be valid. Hence we remove the check in S_2 in order to remove $h_{\mathsf{ID}}, r_{\mathsf{ID}}, r$ from the input to S_2. We obtain the oracle S_2'.

$$\mathsf{Adv}_{\mathcal{A}_3}^{\mathsf{G4}} = \mathsf{Adv}_{\mathcal{A}_3}^{\mathsf{G5}}$$

G6 (Figure 12): Note that r is not used in $\mathsf{OCommit}$ anymore, hence we remove it from state. Moreover, we use the fact that Com is perfectly hiding and replace com with a random group element. We further remove the generation of r. Note that $\mathsf{OLaunch}$ does not contain pw^* anymore.

$$\mathsf{Adv}_{\mathcal{A}_3}^{\mathsf{G5}} = \mathsf{Adv}_{\mathcal{A}_3}^{\mathsf{G6}}$$

G7 (Figure 13): We want to remove h_{ID} from line 3 in OOrder. Note that $\mathsf{com} = \mathsf{Com}(h_{\mathsf{ID}}, att; r) = \mathsf{Com}(0, att; r) + \mathsf{Com}(h_{\mathsf{ID}}, 0; 0)$. Instead of checking $\mathsf{com} = \mathsf{Com}(h_{\mathsf{ID}}, att; r)$. We rewrite this test whether a given h_{ID} is correct using the following: $\mathsf{com} - \mathsf{Com}(0, att; r) = \mathsf{Com}(h_{\mathsf{ID}}, 0; 0)$. We introduce an $\mathsf{OTest}(w)$ oracle that helps to rewrite this in the game. Since h_{ID} and r_{ID} is not used, we remove them along with s_{A}^* from OOrder:

$$\mathsf{Adv}_{\mathcal{A}_3}^{\mathsf{G6}} = \mathsf{Adv}_{\mathcal{A}_3}^{\mathsf{G7}}$$

G8 (Figure 14): We expand Enroll and since r_{ID} is removed we replace $\mathsf{com}_{\mathsf{ID}}$ by a random group element by using the hiding property of Com. Since h_{ID} is also removed, we remove s_{A}^* entirely. Now pw^* is only used in OTest:

$$\mathsf{Adv}_{\mathcal{A}_3}^{\mathsf{G7}} = \mathsf{Adv}_{\mathcal{A}_3}^{\mathsf{G8}}$$

G9 (Figure 15): Observe that passing the OTest oracle is equivalent to recovering $Com(H(\mathsf{pw}^*), 0; 0)$. We can replace OTest with an always reject oracle by introducing a loss of $\frac{q_{\mathsf{Ord}}}{2^{\mathcal{D}_{min}}}$ which is the probability of correctly guessing pw^* with q_{Ord} queries to the OOrder oracle:

$$\mathsf{Adv}_{\mathcal{A}_3}^{\mathsf{G8}} = \mathsf{Adv}_{\mathcal{A}_3}^{\mathsf{G9}} + \frac{q_{\mathsf{Ord}}}{2^{\mathcal{D}_{min}}}$$

$\mathsf{G5}(\mathcal{A}_3, \mathcal{D})$

$\mathsf{given} \leftarrow \emptyset$
$\mathsf{cst} \leftarrow \{\}$ // challenge state
$\mathsf{LH} \leftarrow \{\}$ // lazy sampling oracle
$\mathsf{pk}_{\mathsf{IdM}}, \mathsf{st}, \mathsf{login}^*, \mathsf{ID}^* \leftarrow \mathcal{A}_{3,1}(1^\lambda)$
$\mathsf{pw}^* \leftarrow\!\!\$\ \mathcal{D}$
$s^*_{\mathsf{A}}, \mathsf{contract}^* \leftarrow \mathsf{Enroll}(\mathsf{ID}^*, \mathsf{pw}^*, \mathsf{pk}_{\mathsf{IdM}})$
$\mathcal{A}_{3,2}^{\mathsf{O}*,\mathsf{S}_1}(\mathsf{contract}^*, \mathsf{st}) \rightarrow att^*, \mathsf{consent}^*$
if $\mathsf{Verify}(\mathsf{contract}^*, att^*, \mathsf{consent}^*)$
$\wedge\ \mathsf{consent}^* \notin \mathsf{given}$:
  return 1
return 0

$\mathsf{OLaunch}(sid, att)$

if $sid \in \mathsf{cst}$
  abort
$r \leftarrow\!\!\$\ \{0,1\}^\lambda$
$\mathsf{com} \leftarrow \mathsf{Com}(H(\mathsf{pw}^*), att; r)$
$\mathsf{query} \leftarrow \mathsf{com}$
$\mathsf{state} \leftarrow (r, \mathsf{com})$
$\mathsf{cst}[sid].att \leftarrow att$
$\mathsf{cst}[sid].\mathsf{state} \leftarrow \mathsf{state}$
return query

$\mathsf{S}'_2(x)$

$\pi \leftarrow\!\!\$\ \Pi_{\mathsf{MULTEQ}}.\mathsf{Sim}(x)$
$\pi \rightarrow (\alpha, \mathsf{ch}, \gamma)$
if $(x, \alpha) \in \mathsf{LH}$
  abort
$\mathsf{LH}[(x, \alpha)] \leftarrow \mathsf{ch}$
return π

$\mathsf{OCommit}(sid, \mathsf{resp})$

if $sid \notin \mathsf{cst}$
  abort
$\mathsf{cst}[sid].\mathsf{state} \rightarrow r, \mathsf{com}$
$\mathsf{cst}[sid].att \rightarrow att$
$\mathsf{resp} \rightarrow \sigma$
remove sid from cst
$\mathsf{contract}^* \rightarrow \mathsf{pk}_{\mathsf{IdM}}, \mathsf{com}_{\mathsf{ID}}$
return false **if** $\Sigma.\mathsf{Verify}(\mathsf{pk}_{\mathsf{IdM}}, \sigma, \mathsf{com}) \neq 1$
$\boxed{\pi \leftarrow \mathsf{S}'_2((\mathsf{com}_{\mathsf{ID}}, \mathsf{com}, att))}$
$\mathsf{consent} \leftarrow (\sigma, \mathsf{com}, \pi)$
$\mathsf{given} \leftarrow \mathsf{given} \cup \mathsf{consent}$
return $\mathsf{consent}$

$\mathsf{OOrder}(att, \mathsf{order})$

$\mathsf{order} \rightarrow \mathsf{com}, r, \sigma$
$s^*_{\mathsf{A}}, \mathsf{contract}^* \rightarrow h_{\mathsf{ID}}, r_{\mathsf{ID}}, \mathsf{pk}_{\mathsf{IdM}}, \mathsf{com}_{\mathsf{ID}}$
return false **if** $\mathsf{com} \neq \mathsf{Com}(h_{\mathsf{ID}}, att, r)$
return false **if** $\Sigma.\mathsf{Verify}(\mathsf{pk}_{\mathsf{IdM}}, \sigma, \mathsf{com}) \neq 1$
$\boxed{\pi \leftarrow \mathsf{S}'_2((\mathsf{com}_{\mathsf{ID}}, \mathsf{com}, att))}$
$\mathsf{consent} \leftarrow (\sigma, \mathsf{com}, \pi)$
return $\mathsf{consent}$

$\mathsf{S}_1(x, \alpha)$

if $(x, \alpha) \notin \mathsf{LH}$
  $\mathsf{LH}[(x, \alpha)] \leftarrow\!\!\$\ \mathbb{Z}_q$
return $\mathsf{LH}[(x, \alpha)]$

Fig. 11. $\mathsf{CUF}_{\mathsf{IdM}}$ G5

$\mathsf{G6}(\mathcal{A}_3, \mathcal{D})$

$\mathsf{given} \leftarrow \emptyset$
$\mathsf{cst} \leftarrow \{\}$ // challenge state
$\mathsf{LH} \leftarrow \{\}$ // lazy sampling oracle
$\mathsf{pk}_{\mathsf{IdM}}, \mathsf{st}, \mathsf{login}^*, \mathsf{ID}^* \leftarrow \mathcal{A}_{3,1}(1^\lambda)$
$\mathsf{pw}^* \leftarrow\!\!\$\ \mathcal{D}$
$s_{\mathsf{A}}^*, \mathsf{contract}^* \leftarrow \mathsf{Enroll}(\mathsf{ID}^*, \mathsf{pw}^*, \mathsf{pk}_{\mathsf{IdM}})$
$\mathcal{A}_{3,2}^{\mathsf{O}*,\mathsf{S}_1}(\mathsf{contract}^*, \mathsf{st}) \rightarrow att^*, \mathsf{consent}^*$
if $\mathsf{Verify}(\mathsf{contract}^*, att^*, \mathsf{consent}^*)$
$\wedge\ \mathsf{consent}^* \notin \mathsf{given}$:
  return 1
return 0

$\mathsf{OLaunch}(sid, att)$

if $sid \in \mathsf{cst}$
  abort
$\boxed{\mathsf{com} \leftarrow\!\!\$\ \mathbb{G}}$
$\mathsf{query} \leftarrow \mathsf{com}$
$\boxed{\mathsf{state} \leftarrow \mathsf{com}}$
$\mathsf{cst}[sid].att \leftarrow att$
$\mathsf{cst}[sid].\mathsf{state} \leftarrow \mathsf{state}$
return query

$\mathsf{S}_2'(x)$

$\pi \leftarrow\!\!\$\ \mathit{\Pi}_{\mathsf{MULTEQ}}.\mathsf{Sim}(x)$
$\pi \rightarrow (\alpha, \mathsf{ch}, \gamma)$
if $(x, \alpha) \in \mathsf{LH}$
  abort
$\mathsf{LH}[(x, \alpha)] \leftarrow \mathsf{ch}$
return π

$\mathsf{OCommit}(sid, \mathsf{resp})$

if $sid \notin \mathsf{cst}$
  abort
$\boxed{\mathsf{cst}[sid].\mathsf{state} \rightarrow \mathsf{com}}$
$\mathsf{cst}[sid].att \rightarrow att$
$\mathsf{resp} \rightarrow \sigma$
remove sid from cst
$\mathsf{contract}^* \rightarrow \mathsf{pk}_{\mathsf{IdM}}, \mathsf{com}_{\mathsf{ID}}$
return false **if** $\Sigma.\mathsf{Verify}(\mathsf{pk}_{\mathsf{IdM}}, \sigma, \mathsf{com}) \neq 1$
$\pi \leftarrow \mathsf{S}_2'((\mathsf{com}_{\mathsf{ID}}, \mathsf{com}, att))$
$\mathsf{consent} \leftarrow (\sigma, \mathsf{com}, \pi)$
$\mathsf{given} \leftarrow \mathsf{given} \cup \mathsf{consent}$
return $\mathsf{consent}$

$\mathsf{OOrder}(att, \mathsf{order})$

$\mathsf{order} \rightarrow \mathsf{com}, r, \sigma$
$s_{\mathsf{A}}^*, \mathsf{contract}^* \rightarrow h_{\mathsf{ID}}, r_{\mathsf{ID}}, \mathsf{pk}_{\mathsf{IdM}}, \mathsf{com}_{\mathsf{ID}}$
return false **if** $\mathsf{com} \neq \mathsf{Com}(h_{\mathsf{ID}}, att, r)$
return false **if** $\Sigma.\mathsf{Verify}(\mathsf{pk}_{\mathsf{IdM}}, \sigma, \mathsf{com}) \neq 1$
$\pi \leftarrow \mathsf{S}_2'((\mathsf{com}_{\mathsf{ID}}, \mathsf{com}, att))$
$\mathsf{consent} \leftarrow (\sigma, \mathsf{com}, \pi)$
return $\mathsf{consent}$

$\mathsf{S}_1(x, \alpha)$

if $(x, \alpha) \notin \mathsf{LH}$
  $\mathsf{LH}[(x, \alpha)] \leftarrow\!\!\$\ \mathbb{Z}_q$
return $\mathsf{LH}[(x, \alpha)]$

Fig. 12. $\mathsf{CUF}_{\mathsf{IdM}}$ G6

```
G7(𝒜_3, 𝒟)
given ← ∅
cst ← {}    // challenge state
LH ← {}     // lazy sampling oracle
pk_IdM, st, login*, ID* ← 𝒜_{3,1}(1^λ)
pw* ←$ 𝒟
s*_A, contract* ← Enroll(ID*, pw*, pk_IdM)
𝒜_{3,2}^{O*,S_1}(contract*, st) → att*, consent*
if Verify(contract*, att*, consent*)
 ∧ consent* ∉ given :
   return 1
return 0
```

```
OLaunch(sid, att)
if sid ∈ cst
   abort
com ←$ 𝔾
query ← com
state ← com
cst[sid].att ← att
cst[sid].state ← state
return query
```

```
S'_2(x)
π ←$ Π_MULTEQ.Sim(x)
π → (α, ch, γ)
if (x, α) ∈ LH
   abort
LH[(x, α)] ← ch
return π
```

```
OCommit(sid, resp)
if sid ∉ cst
   abort
cst[sid].state → com
cst[sid].att → att
resp → σ
remove sid from cst
contract* → pk_IdM, com_ID
return false if Σ.Verify(pk_IdM, σ, com) ≠ 1
π ← S'_2((com_ID, com, att))
consent ← (σ, com, π)
given ← given ∪ consent
return consent
```

```
OOrder(att, order)
order → com, r, s
contract* → pk_IdM, com_ID
[ return false if OTest(com − Com(0, att; r)) ≠ 1 ]
return false if Σ.Verify(pk_IdM, s, com) ≠ 1
σ ← S'_2((com_ID, com, att))
consent ← (σ, com, σ)
return consent
```

```
S_1(x, α)
if (x, α) ∉ LH
   LH[(x, α)] ←$ ℤ_q
return LH[(x, α)]
```

```
OTest(w)
return w =? Com(H(pw*), 0; 0)
```

Fig. 13. $\mathsf{CUF}_{\mathsf{IdM}}$ G7

```
G8(A_3, D)
  given ← ∅
  cst ← {}    // challenge state
  LH ← {}     // lazy sampling oracle
  pk_IdM, st, login*, ID* ← A_{3,1}(1^λ)
  pw* ←$ D
  [ h*, r* ←$ Z_q ]
  [ com_ID ←$ Com(h*, 0; r*) ]
  [ contract* ← ID*, com_ID, pk_IdM ]
  A_{3,2}^{O*,S_1}(contract*, st) → att*, consent*
  if Verify(contract*, att*, consent*)
   ∧ consent* ∉ given :
    return 1
  return 0

OLaunch(sid, att)
  if sid ∈ cst
    abort
  com ←$ G
  query ← com
  state ← com
  cst[sid].att ← att
  cst[sid].state ← state
  return query

S'_2(x)
  π ←$ Π_MULTEQ.Sim(x)
  π → (α, ch, γ)
  if (x, α) ∈ LH
    abort
  LH[(x, α)] ← ch
  return π

OCommit(sid, resp)
  if sid ∉ cst
    abort
  cst[sid].state → com
  cst[sid].att → att
  resp → σ
  remove sid from cst
  contract* → pk_IdM, com_ID
  return false if Σ.Verify(pk_IdM, σ, com) ≠ 1
  π ← S'_2((com_ID, com, att))
  consent ← (σ, com, π)
  given ← given ∪ consent
  return consent

OOrder(att, order)
  order → com, r, σ
  contract* → pk_IdM, com_ID
  return false if OTest(com − Com(0, att; r)) ≠ 1
  return false if Σ.Verify(pk_IdM, σ, com) ≠ 1
  π ← S'_2((com_ID, com, att))
  consent ← (σ, com, π)
  return consent

S_1(x, α)
  if (x, α) ∉ LH
    LH[(x, α)] ←$ Z_q
  return LH[(x, α)]

OTest(w)
  return w =? Com(H(pw*), 0; 0)
```

Fig. 14. $\mathsf{CUF}_{\mathsf{IdM}}$ G8

$\mathsf{G9}(\mathcal{A}_3, \mathcal{D})$

$\mathsf{given} \leftarrow \emptyset$
$\mathsf{cst} \leftarrow \{\}$ // challenge state
$\mathsf{LH} \leftarrow \{\}$ // lazy sampling oracle
$\mathsf{pk_{IdM}}, \mathsf{st}, \mathsf{login}^*, \mathsf{ID}^* \leftarrow \mathcal{A}_{3,1}(1^\lambda)$
$\mathsf{pw}^* \leftarrow\$ \mathcal{D}$
$h^*, r^* \leftarrow\$ \mathbb{Z}_q$
$\mathsf{com_{ID}} \leftarrow\$ \mathsf{Com}(h^*, 0; r^*)$
$\mathsf{contract}^* \leftarrow \mathsf{ID}^*, \mathsf{com_{ID}}, \mathsf{pk_{IdM}}$
$\mathcal{A}_{3,2}^{\mathsf{O}*,\mathsf{S}_1}(\mathsf{contract}^*, \mathsf{st}) \rightarrow att^*, \mathsf{consent}^*$
if $\mathsf{Verify}(\mathsf{contract}^*, att^*, \mathsf{consent}^*)$
$\wedge\ \mathsf{consent}^* \notin \mathsf{given}$:
  return 1
return 0

$\mathsf{OLaunch}(sid, att)$

if $sid \in \mathsf{cst}$
  abort
$\mathsf{com} \leftarrow\$ \mathbb{G}$
$\mathsf{query} \leftarrow \mathsf{com}$
$\mathsf{state} \leftarrow \mathsf{com}$
$\mathsf{cst}[sid].att \leftarrow att$
$\mathsf{cst}[sid].\mathsf{state} \leftarrow \mathsf{state}$
return query

$\mathsf{S}_2'(x)$

$\pi \leftarrow\$ \Pi_{\mathsf{MULTEQ}}.\mathsf{Sim}(x)$
$\pi \rightarrow (\alpha, \mathsf{ch}, \gamma)$
if $(x, \alpha) \in \mathsf{LH}$
  abort
$\mathsf{LH}[(x, \alpha)] \leftarrow \mathsf{ch}$
return π

$\mathsf{OCommit}(sid, \mathsf{resp})$

if $sid \notin \mathsf{cst}$
  abort
$\mathsf{cst}[sid].\mathsf{state} \rightarrow \mathsf{com}$
$\mathsf{cst}[sid].att \rightarrow att$
$\mathsf{resp} \rightarrow \sigma$
remove sid from cst
$\mathsf{contract}^* \rightarrow \mathsf{pk_{IdM}}, \mathsf{com_{ID}}$
return false **if** $\Sigma.\mathsf{Verify}(\mathsf{pk_{IdM}}, \sigma, \mathsf{com}) \neq 1$
$\pi \leftarrow \mathsf{S}_2'((\mathsf{com_{ID}}, \mathsf{com}, att))$
$\mathsf{consent} \leftarrow (\sigma, \mathsf{com}, \pi)$
$\mathsf{given} \leftarrow \mathsf{given} \cup \mathsf{consent}$
return $\mathsf{consent}$

$\mathsf{OOrder}(att, \mathsf{order})$

$\mathsf{order} \rightarrow \mathsf{com}, r, \sigma$
$\mathsf{contract}^* \rightarrow \mathsf{pk_{IdM}}, \mathsf{com_{ID}}$
return false **if** $\mathsf{OTest}(\mathsf{com} - \mathsf{Com}(0, att; r)) \neq 1$
return false **if** $\Sigma.\mathsf{Verify}(\mathsf{pk_{IdM}}, \sigma, \mathsf{com}) \neq 1$
$\pi \leftarrow \mathsf{S}_2'((\mathsf{com_{ID}}, \mathsf{com}, att))$
$\mathsf{consent} \leftarrow (\sigma, \mathsf{com}, \pi)$
return $\mathsf{consent}$

$\mathsf{S}_1(x, \alpha)$

if $(x, \alpha) \notin \mathsf{LH}$
  $\mathsf{LH}[(x, \alpha)] \leftarrow\$ \mathbb{Z}_q$
return $\mathsf{LH}[(x, \alpha)]$

$\mathsf{OTest}(w)$

[**return** false]

Fig. 15. $\mathsf{CUF_{IdM}}$ G9

G10 (Figure 16): Since the OTest check in OOrder will always fail, we replace OOrder with returning false:

$$\mathsf{Adv}_{\mathcal{A}_3}^{\mathsf{G9}} = \mathsf{Adv}_{\mathcal{A}_3}^{\mathsf{G10}}$$

G11 (Figure 17): We simulate the $\mathsf{OLaunch}$ and the $\mathsf{OCommit}$ oracle using a single oracle OSim that returns simulated proofs for a given commitment with an input (com, att). We simplify the game by removing the unused variables and signatures. We also explicit the random coins ρ of the adversary. Now we have an adversary that tries to forge a valid proof by observing simulated proofs.

$$\mathsf{Adv}_{\mathcal{A}_3}^{\mathsf{G10}} = \mathsf{Adv}_{\mathcal{A}_{11}}^{\mathsf{G11}}$$

G12 (Figure 18): We use the weak simulation extractability of our NIZK, as discussed in subsection 4.2, to obtain the extraction of a witness for the newly generated proof, thanks to an extractor $\mathcal{E}$.

$$\mathsf{Adv}_{\mathcal{A}_{11}}^{\mathsf{G11}} \leq \frac{q_{S_1} + q_{S_2}}{q} + \sqrt{(q_{S_1} + q_{S_2}) \cdot \mathsf{Adv}_{\mathcal{A}_{11}}^{\mathsf{G12}}}$$

G13 (Figure 19): By now including nearly everything inside the adversary, we obtain an new adversary $\mathcal{B}$ who can open any random commitment $\mathsf{com}_{\mathsf{ID}}^*$.

$$\mathsf{Adv}_{\mathcal{A}_{11}}^{\mathsf{G12}} = \mathsf{Adv}_{\mathcal{B}}^{\mathsf{G13}}$$

G14 (Figure 20): We introduce a new condition for success: that $h_{\mathsf{ID}} \neq h^*$. The failing event $h_{\mathsf{ID}} = h^*$ happens with probability $\frac{1}{q}$, due to that Com being perfectly hiding. Hence,

$$\mathsf{Adv}_{\mathcal{B}}^{\mathsf{G13}} \leq \mathsf{Adv}_{\mathcal{B}}^{\mathsf{G14}} + \frac{1}{q}$$

Wrapping up: The game $\mathsf{G14}$ is the binding security game for the commitment. Hence, $\mathsf{Adv}_{\mathcal{B}}^{\mathsf{G14}}$ is the advantage in the binding game. The number of queries to S_1 corresponds to the queries q_H to the random oracle. The number of queries to S_2 (or S_2') is the number of queries to either $\mathsf{OCommit}$ or OOrder.

```
G10(A_3, D)
  given ← ∅
  cst ← {}   // challenge state
  LH ← {}   // lazy sampling oracle
  pk_IdM, st, login*, ID* ← A_{3,1}(1^λ)
  pw* ←$ D
  h*, r* ←$ Z_q
  com_ID ←$ Com(h*, 0; r*)
  contract* ← ID*, com_ID, pk_IdM
  A_{3,2}^{O*,S_1}(contract*, st) → att*, consent*
  if Verify(contract*, att*, consent*)
   ∧ consent* ∉ given :
    return 1
  return 0

OLaunch(sid, att)
  if sid ∈ cst
    abort
  com ←$ G
  query ← com
  state ← com
  cst[sid].att ← att
  cst[sid].state ← state
  return query

S'_2(x)
  π ←$ Π_MULTEQ.Sim(x)
  π → (α, ch, γ)
  if (x, α) ∈ LH
    abort
  LH[(x, α)] ← ch
  return π

OCommit(sid, resp)
  if sid ∉ cst
    abort
  cst[sid].state → com
  cst[sid].att → att
  resp → σ
  remove sid from cst
  contract* → pk_IdM, com_ID
  return false if Σ.Verify(pk_IdM, σ, com) ≠ 1
  π ← S'_2((com_ID, com, att))
  consent ← (σ, com, π)
  given ← given ∪ consent
  return consent

OOrder(att, order)
  [return false]

S_1(x, α)
  if (x, α) ∉ LH
    LH[(x, α)] ←$ Z_q
  return LH[(x, α)]

OTest(w)
  return false
```

Fig. 16. $\mathsf{CUF}_{\mathsf{IdM}}$ G10

$\mathsf{G11}(\mathcal{A}_{11}, \mathcal{D})$

$\mathsf{given} \leftarrow \emptyset$
$\mathsf{LH} \leftarrow \{\}$ // lazy sampling oracle
$h^*, r^* \leftarrow\$ \mathbb{Z}_q$
$\mathsf{com_{ID}} \leftarrow\$ \mathsf{Com}(h^*, 0; r^*)$
pick ρ
$\mathcal{A}_{11}^{\mathsf{O}*, \mathsf{S}_1}(\mathsf{com_{ID}}; \rho) \rightarrow \pi, att^*, \mathsf{com}^*$
if $\neg \mathit{\Pi}_{\mathsf{MULTEQ}}.\mathsf{Verify}(\pi, \mathsf{com}^*_{\mathsf{ID}}, \mathsf{com}^*, att^*)$
$\lor (\mathsf{com}^*, \pi) \in \mathsf{given}$:
  return 0
return 1

$\mathsf{S}'_2(x)$

$\pi \leftarrow\$ \mathit{\Pi}_{\mathsf{MULTEQ}}.\mathsf{Sim}(x)$
$\pi \rightarrow (\alpha, \mathsf{ch}, \gamma)$
if $(x, \alpha) \in \mathsf{LH}$
  abort
$\mathsf{LH}[(x, \alpha)] \leftarrow \mathsf{ch}$
return π

$\mathsf{OSim}(\mathsf{com}, att)$

$\pi \leftarrow \mathsf{S}'_2((\mathsf{com}^*_{\mathsf{ID}}, \mathsf{com}, att))$
$\mathsf{given} \leftarrow \mathsf{given} \cup (\mathsf{com}, \pi)$
return π

$\mathsf{S}_1(x, \alpha)$

if $(x, \alpha) \notin \mathsf{LH}$
  $\mathsf{LH}[(x, \alpha)] \leftarrow\$ \mathbb{Z}_q$
return $\mathsf{LH}[(x, \alpha)]$

Fig. 17. $\mathsf{CUF_{IdM}}$ $\mathsf{G11}$

A.2 $\mathsf{CUF_{Agent}}$ Security: Proof of Theorem 2

Proof. For the games we construct during the proof, we denote the *i-th* game with $\mathsf{G}i$. We set $\mathsf{G0} = \mathsf{CUF_{Agent}}$.

G1 (Figure 21): We expand each subprocedure.

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{G0}} = \mathsf{Adv}_{\mathcal{A}}^{\mathsf{G1}}$$

G2 (Figure 22):

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{G1}} = \mathsf{Adv}_{\mathcal{A}}^{\mathsf{G2}}$$

G3 (Figure 23): The $\Sigma.\mathsf{Sign}$ operations could be outsourced to a signing oracle OSign, which is the only place where we need $\mathsf{sk_{IdM}}$. A valid consent in $\mathsf{G2}$ must include a valid signature σ on some $(\mathsf{com}^*, \mathsf{ID}^*)$ pair. We reduce to a game $\mathsf{G3}$ where this signature is *not* a forgery, by using the unforgeability of the signature (i.e. by defining an adversary $\mathcal{B}$ who would make a forgery on Σ). Clearly, the

$\mathsf{G12}(\mathcal{A}_{11}, \mathcal{D})$

$\mathsf{given} \leftarrow \emptyset$
$\mathsf{LH} \leftarrow \{\}$ // lazy sampling oracle
$h^*, r^* \leftarrow\$ \mathbb{Z}_q$
$\mathsf{com}_{\mathsf{ID}} \leftarrow\$ \mathsf{Com}(h^*, 0; r^*)$
pick ρ
$\mathcal{A}_{11}^{\mathsf{O}_*, \mathsf{S}_1}(\mathsf{com}_{\mathsf{ID}}; \rho) \rightarrow \pi, att^*, \mathsf{com}^*$
if $\neg \Pi_{\mathsf{MULTEQ}}.\mathsf{Verify}(\pi, \mathsf{com}^*_{\mathsf{ID}}, \mathsf{com}^*, att^*)$
$\quad \vee (\mathsf{com}^*, \pi) \in \mathsf{given}$:
$\quad$ **return** 0
$\mathcal{E}(\mathsf{com}^*_{\mathsf{ID}}, \mathsf{com}^*, att^*, \pi; \rho, \mathsf{LH}, \mathsf{given}) \rightarrow h_{\mathsf{ID}}, r_{\mathsf{ID}}, r$
if $\mathcal{R}((\mathsf{com}^*_{\mathsf{ID}}, \mathsf{com}^*, att^*), (h_{\mathsf{ID}}, r_{\mathsf{ID}}, r))$:
$\quad$ **return** 1
return 0

$\mathsf{S}'_2(x)$

$\pi \leftarrow\$ \Pi_{\mathsf{MULTEQ}}.\mathsf{Sim}(x)$
$\pi \rightarrow (\alpha, \mathsf{ch}, \gamma)$
if $(x, \alpha) \in \mathsf{LH}$
$\quad$ **abort**
$\mathsf{LH}[(x, \alpha)] \leftarrow \mathsf{ch}$
return π

$\mathsf{OSim}(\mathsf{com}, att)$

$\pi \leftarrow \mathsf{S}'_2((\mathsf{com}^*_{\mathsf{ID}}, \mathsf{com}, att))$
$\mathsf{given} \leftarrow \mathsf{given} \cup (\mathsf{com}, \pi)$
return π

$\mathsf{S}_1(x, \alpha)$

if $(x, \alpha) \notin \mathsf{LH}$
$\quad \mathsf{LH}[(x, \alpha)] \leftarrow\$ \mathbb{Z}_q$
return $\mathsf{LH}[(x, \alpha)]$

Fig. 18. $\mathsf{CUF}_{\mathsf{IdM}}$ G12

$\mathsf{G13}(\mathcal{B}, \mathcal{D})$

$h^*, r^* \leftarrow\$ \mathbb{Z}_q$
$\mathsf{com}_{\mathsf{ID}} \leftarrow\$ \mathsf{Com}(h^*, 0; r^*)$
$\mathcal{B}(\mathsf{com}_{\mathsf{ID}}) \rightarrow h_{\mathsf{ID}}, r_{\mathsf{ID}}$
if $\mathsf{Com}(h_{\mathsf{ID}}, 0; r_{\mathsf{ID}}) = \mathsf{com}^*_{\mathsf{ID}}$:
$\quad$ **return** 1
return 0

Fig. 19. $\mathsf{CUF}_{\mathsf{IdM}}$ G13

$\mathsf{G14}(\mathcal{B}, \mathcal{D})$

```
h*, r* <-$ Z_q
com_ID <-$ Com(h*, 0; r*)
B(com_ID) -> h_ID, r_ID
if Com(h_ID, 0; r_ID) = com*_ID ∧ h_ID ≠ h* :
  return 1
return 0
```

Fig. 20. $\mathsf{CUF}_{\mathsf{IdM}}$ G14

signed pair cannot be one of those signed in OCorruptQuery as ID^* would not be in db otherwise and thus would make the game abort. Hence, it must come from OLaunch, which was for some attribute *att*. But to make the game succeed, we must have $att \neq att^*$.

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{G2}} \leq \mathsf{Adv}_{\mathcal{A}}^{\mathsf{G3}} + \mathsf{Adv}_{\mathcal{B}}^{\mathsf{UF}}$$

G4 (Figure 24): We make $\mathcal{A}_4$ simulate everything in the game except the random oracle, the computation of com^*, and the verification of π.

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{G3}} = \mathsf{Adv}_{\mathcal{A}_4}^{\mathsf{G4}}$$

G5 (Figure 25): We use the weak simulation extractability of our NIZK, as discussed in subsection 4.2, to obtain the extraction of a witness for the newly generated proof, thanks to an extractor $\mathcal{E}$.

$$\mathsf{Adv}_{\mathcal{A}_4}^{\mathsf{G4}} \leq \frac{q_{S_1}}{q} + \sqrt{q_{S_1} \cdot \mathsf{Adv}_{\mathcal{A}_4}^{\mathsf{G5}}}$$

G6 (Figure 25): The game G5 boils down to the binding security game for the commitment.

$$\mathsf{Adv}_{\mathcal{A}_4}^{\mathsf{G5}} = \mathsf{Adv}_{\mathcal{C}}^{\mathsf{G6}}$$

Wrapping up: Clearly, $\mathsf{Adv}_{\mathcal{C}}^{\mathsf{G6}}$ is the advantage in the binding game. The number of queries to S_1 corresponds to the queries q_H to the random oracle.

```
G1(𝒜)
corrupted ← ∅
ordered ← ∅
sk_IdM, pk_IdM ← IdM.Setup(1^λ)
𝒜^oracles(pk_IdM) → ID*, att*, consent*
abort if ID* ∉ db
db[ID*] → contract*
if Verify(contract*, att*, consent*)
 ∧ (ID*, att*) ∉ ordered :
   return 1
return 0

OEnroll(ID, pw)
if ID ∈ db ∨ ID ∈ corrupted
   abort
(s_A, contract) ← Enroll(ID, pw, pk_IdM)
db[ID] ← (contract, pw)
return s_A, contract
```

```
OLaunch(ID, att)
if ID ∉ db
   abort
db[ID] → pw
┌ Launch
│ r ←$ {0,1}*
│ com ← Com(H(pw), att; r)
┌ IdM
│ σ ← Σ.Sign(sk_IdM, com, ID)
┌ Commit
│ order ← (com, r, σ)
ordered ← ordered ∪ (ID, att)
return com, σ, order

OCorruptQuery(ID, query)
if ID ∉ corrupted
   abort
┌ IdM
│ σ ← Σ.Sign(sk_IdM, com, ID)
return resp

OCorruptEnroll(ID)
if ID ∈ db
   abort
corrupted ← corrupted ∪ ID
return ⊥
```

Fig. 21. $\mathsf{CUF}_{\mathsf{Agent}}$ G1

```
G2(A)
corrupted ← ∅
ordered ← ∅
sk_IdM, pk_IdM ← IdM.Setup(1^λ)
A^oracles(pk_IdM) → ID*, att*, consent*
abort if ID* ∉ db
db[ID*] → contract*
if Verify(contract*, att*, consent*)
 ∧ (ID*, att*) ∉ ordered :
  return 1
return 0
```

```
OEnroll(ID, pw)
if ID ∈ db ∨ ID ∈ corrupted
  abort
(s_A, contract) ← Enroll(ID, pw, pk_IdM)
db[ID] ← (contract, pw)
return s_A, contract
```

```
OLaunch(ID, att)
if ID ∉ db
  abort
db[ID] → pw
r ←$ {0,1}*
com ← Com(H(pw), att; r)
σ ← Σ.Sign(sk_IdM, com, ID)
order ← (com, r, s)
ordered ← ordered ∪ (ID, att)
return com, σ, order
```

```
OCorruptQuery(ID, query)
if ID ∉ corrupted
  abort
σ ← Σ.Sign(sk_IdM, com, ID)
return resp
```

```
OCorruptEnroll(ID)
if ID ∈ db
  abort
corrupted ← corrupted ∪ ID
return ⊥
```

Fig. 22. $\mathsf{CUF}_{\mathsf{Agent}}$ G2

```
G3(𝒜)
  corrupted ← ∅
  ordered ← ∅
  [signed ← ∅]
  sk_IdM, pk_IdM ← IdM.Setup(1^λ)
  𝒜^oracles(pk_IdM) → ID*, att*, consent*
  abort if ID* ∉ db
  db[ID*] → (., com*_ID, .)
  [consent* → (σ*, com*, π*)]
  if [Π_MULTEQ.Verify(π*, com*_ID, com*, att*)]
    ∧ (ID*, att*) ∉ ordered
    [∧(ID*, com*) ∈ signed] :
      return 1
  return 0

OEnroll(ID, pw)
  if ID ∈ db ∨ ID ∈ corrupted
    abort
  (s_A, contract) ← Enroll(ID, pw, pk_IdM)
  db[ID] ← (contract, pw)
  return s_A, contract

OLaunch(ID, att)
  if ID ∉ db
    abort
  db[ID] → pw
  r ←$ {0, 1}*
  com ← Com(H(pw), att; r)
  σ ← Σ.Sign(sk_IdM, com, ID)
  order ← (com, r, s)
  ordered ← ordered ∪ (ID, att)
  [signed ← signed ∪ (ID, com)]
  return com, σ, order

OCorruptQuery(ID, query)
  if ID ∉ corrupted
    abort
  σ ← Σ.Sign(sk_IdM, com, ID)
  return resp

OCorruptEnroll(ID)
  if ID ∈ db
    abort
  corrupted ← corrupted ∪ ID
  return ⊥
```

Fig. 23. $\mathsf{CUF}_{\mathsf{Agent}}$ G3

```
G4(𝒜_4)
  𝒜_4^{S_1}() → h_ID, com_ID, att, r, att*, π
  com ← Com(h_ID, att; r)
  if Π_MULTEQ.Verify(π, com_ID, com, att*)
    ∧ att ≠ att* :
      return 1
  return 0

S_1(x, α)
  if (x, α) ∉ LH
    LH[(x, α)] ←$ ℤ_q
  return LH[(x, α)]
```

Fig. 24. $\mathsf{CUF}_{\mathsf{Agent}}$ G4

$\mathsf{G5}(\mathcal{A}_4)$

$\boxed{\text{pick } \rho}$

$\mathcal{A}_4^{\mathsf{S}_1}(\boxed{\rho}) \rightarrow h_{\mathsf{ID}}, \mathsf{com}_{\mathsf{ID}}, att, r, att^*, \pi$

$\mathsf{com} \leftarrow \mathsf{Com}(h_{\mathsf{ID}}, att; r)$

$\boxed{\mathcal{E}(\rho, \mathsf{LH}) \rightarrow h^*_{\mathsf{ID}}, r^*_{\mathsf{ID}}, r^*}$

$\textbf{if } \boxed{\mathcal{R}(\mathsf{com}_{\mathsf{ID}}, \mathsf{com}, att^*, h^*_{\mathsf{ID}}, r^*_{\mathsf{ID}}, r^*)}$

$\quad \wedge\, att \neq att^* :$

$\quad \textbf{return } 1$

$\textbf{return } 0$

$\mathsf{S}_1(x, \alpha)$

$\textbf{if } (x, \alpha) \notin \mathsf{LH}$

$\quad \mathsf{LH}[(x, \alpha)] \leftarrow\!\!\$\ \mathbb{Z}_q$

$\textbf{return } \mathsf{LH}[(x, \alpha)]$

Fig. 25. $\mathsf{CUF}_{\mathsf{Agent}}$ G5

$\mathsf{G6}(\mathcal{C})$

$\mathcal{C}() \rightarrow h_{\mathsf{ID}}, att, r, h^*_{\mathsf{ID}}, att^*, r^*$

$\mathsf{com} \leftarrow \mathsf{Com}(h_{\mathsf{ID}}, att; r)$

$\boxed{\mathsf{com}^* \leftarrow \mathsf{Com}(h^*_{\mathsf{ID}}, att^*; r^*)}$

$\textbf{if } \boxed{\mathsf{com} = \mathsf{com}^*} \wedge att \neq att^* :$

$\quad \textbf{return } 1$

$\textbf{return } 0$

Fig. 26. $\mathsf{CUF}_{\mathsf{Agent}}$ G6

B PBS based on blind RSA

In this section, we explain the PBS based on RSA from [JKR13]. Let N, e, d be RSA modulus, exponent and private key respectively. We assume there exists shares d_1, d_2 of an RSA private key d. Such that $d = d_1 + d_2$. This can be done by a joint protocol between Client and Agent or by a dealer. G is a random map for mapping low entropy pw to a group element. H is a random oracle.

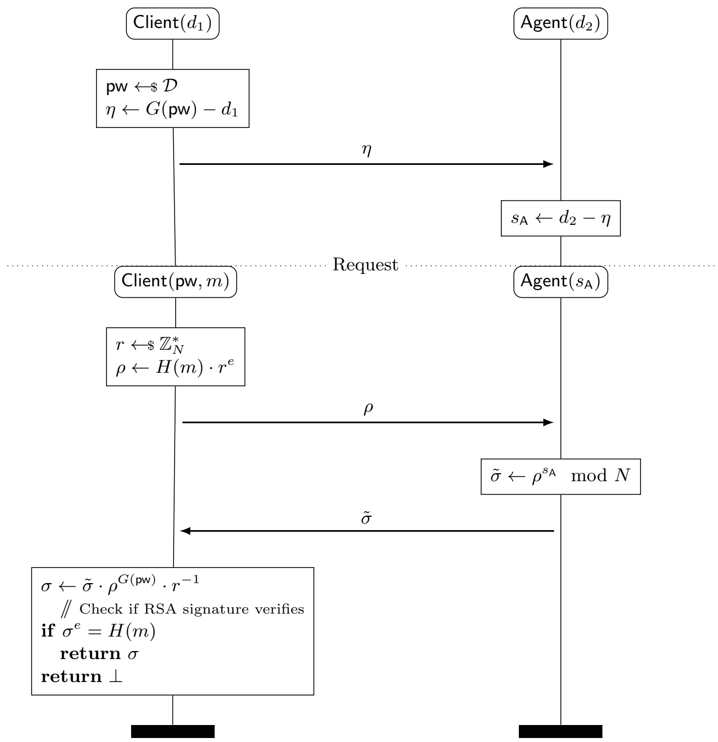

Fig. 27. Password Based Signature Based on RSA

Outsider attack on PBS based on blind RSA. An adversary that can send arbitrary messages to the Agent can mount an offline dictionary attack on the password. First the adversary picks a random r and m. Computes $\rho \leftarrow H(m) \cdot r^e$. Obtains the corresponding $\tilde{\sigma}$ from the agent. After one such query, the following equation can be checked offline to find the password:

$$(\tilde{\sigma} \cdot \rho^{G(\mathsf{pw})} \cdot r^{-1})^e \stackrel{?}{=} H(m)$$

Once the password is found, the adversary can interact with the Agent to forge signatures.

C PBS based on CL Signatures

In [JKR13], the authors also propose a PBS protocol based on CL Signatures [CL01]. Let e be a type 3 pairing group ($\mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_3$) with all groups have prime order p. Let g_1 and Z be generators of $\mathbb{G}_1$ and g_2 be the generator of $\mathbb{G}_2$. Let G be a hash function $G : \{0,1\}^* \rightarrow \mathbb{Z}_p \times \mathbb{Z}_p$. See Fig. 28 for the protocol flow.

Outsider attack on PBS based on CL signatures An adversary that can send arbitrary messages to the Agent can mount an offline dictionary attack on the password. After capturing pk from agent's last message. First the adversary picks a random r and m. Computes $\rho \leftarrow g_1^m \cdot Z^r$. Obtains the corresponding A, C and D from the agent. After one such query, the CL Signature verification equation can be checked offline to find the password. Note that at this point, the adversary is already in possession of a forged signature on message m.

D Server Aided Digital Signatures (SADS)

Notation for the SADS protocol [HWF05] below:

- $\mathsf{RSA.Gen}$: RSA Key Generator
- H: Random Oracle
- $\mathsf{MAC}_{key}(m)$: Message authentication code with key key for message m.
- $\mathsf{Enc}_{key}(m)$: Symmetric encryption with key key for message m
- $\mathsf{Dec}_{key}(c)$: Symmetric decryption with key key for ciphertext c
- $\mathsf{PKE.Gen}()$: Key generator of a PKE.
- $\mathsf{PKE.Enc}_{pk}(m)$: PKE encryption with key pk for message m
- $\mathsf{PKE.Dec}_{sk}(c)$: PKE decryption with key sk for message c
- $\mathsf{Sig.Gen}()$: Key generator of a signature scheme.
- $\mathsf{Sig.Sign}_{sk}(m)$: Signing algorithm of a signature scheme with secret key sk for message m.
- $\mathsf{Sig.Ver}_{pk}(m, sig)$: Verification algorithm of a signature scheme with public key pk on a message m with signature sig.

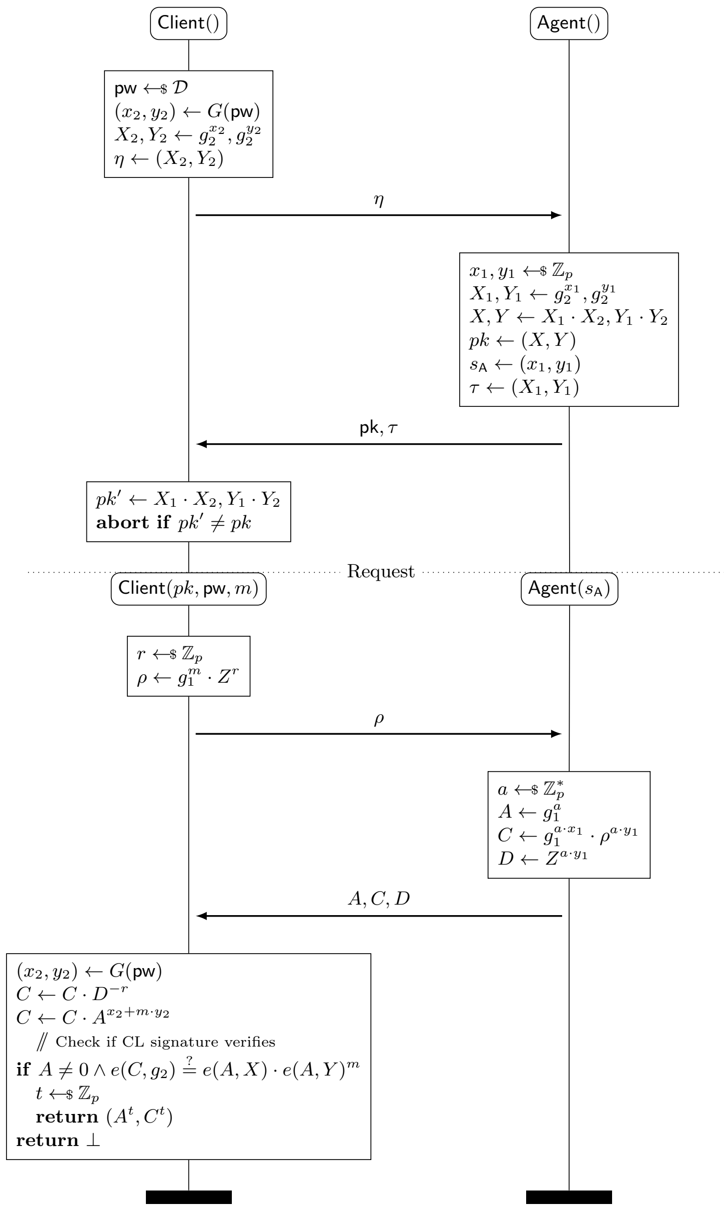

Fig. 28. Password Based Signature Based on CL Signatures

$\mathsf{KeyGen}(\mathsf{pw})$

$\delta_1, \delta_2 \leftarrow\!\!\$\ \mathbb{Z}_q$

$key \leftarrow H(H(\mathsf{pw})^{\delta_1+\delta_2} \mod p)$

$a_1 \leftarrow H(H(\mathsf{pw})^{\delta_1} \mod p)$

$a_2 \leftarrow H(H(\mathsf{pw})^{\delta_2} \mod p)$

$e, d, N \leftarrow\!\!\$\ \mathsf{RSA.Gen}(1^\lambda)$

$d_1 \leftarrow\!\!\$\ \mathbb{Z}_{\phi(N)}$

$d_2 = d - d_1 \mod \phi(N)$

$A \leftarrow \mathsf{Enc}_{key}(d_1||N||\mathsf{MAC}_{key}(d_1||N))$

$ssk_1, spk_1 \leftarrow \mathsf{Sig.Gen}(\lambda)$

$sk_1, pk_1 \leftarrow \mathsf{PKE.Gen}(\lambda)$

$sk_2, pk_2 \leftarrow \mathsf{PKE.Gen}(\lambda)$

return to Server 1$(A, \delta_1, a_2, ssk_1, sk_1)$

return to Server 2$(\delta_2, d_2, a_1, sk_2)$

return to All$(p, N, e, spk_1, pk_1, pk_2)$

Fig. 29. KeyGen for SADS

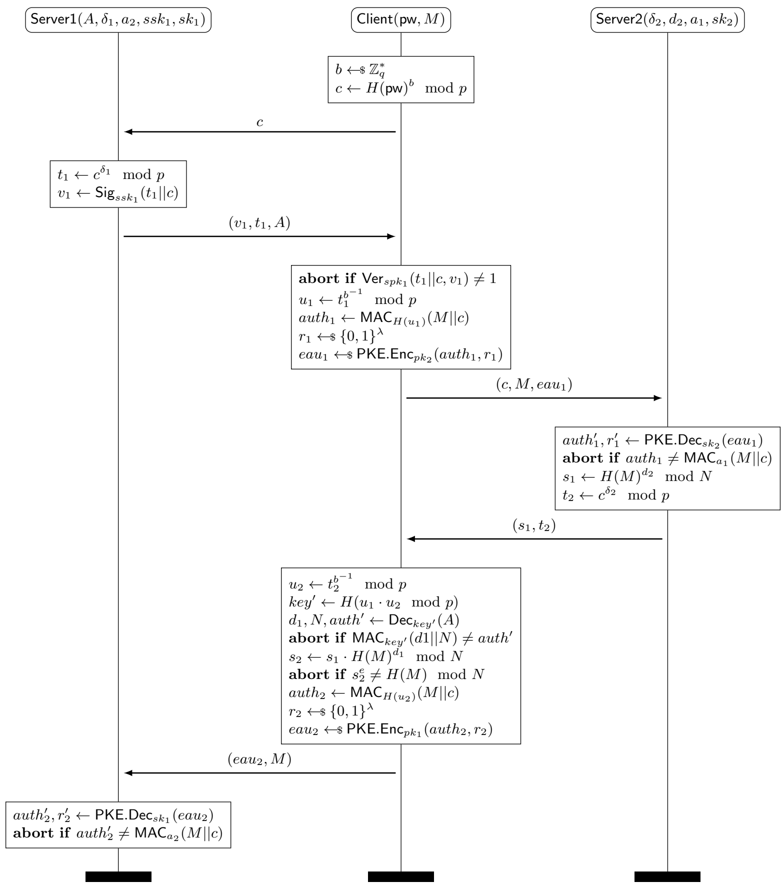

Fig. 30. SADS Protocol Construction