Cryptographically Secure Digital Consent

F. Betül Durak
Microsoft Research
betul.durak@microsoft.com

Abdullah Talayhan*
Bilkent University
talayhan@bilkent.edu.tr

Serge Vaudenay
EPFL
serge.vaudenay@epfl.ch

Abstract

In the digital age, maintaining trust and security in third-party services is crucial, especially when it comes to consent for online actions executed by third parties. This work studies the concept of cryptographically secure digital consent.

We provide a protocol which involves a client (the user) and an agent (which executes the action upon receiving consent) with the help of an identity manager (typically: an SSO service and/or a secure device). The client is required to keep no more than a consent password in addition to their existing authentication credentials. The design addresses several security challenges, including non-repudiability of consent. Security is maintained even if either the identity manager or the agent is compromised.

Our protocol supports various applications, is compatible with existing identity managers, and is very simple.

Our notion of an identity manager is broad, encompassing combinations of different authentication factors such as a password, a smartphone, a security device, biometrics, or national digital identity. We demonstrate applications for signing PDF documents, e-banking, and key recovery. Finally, we implement the PDF signing application, which is robust in the sense that a user can continue to use the signing service even after their devices are broken, lost, or stolen. Hence, an agent can sign on behalf of a user but remains liable for signatures released without consent, which removes a common trust assumption in existing solutions.

1 Introduction

In this paper, we study a family of applications where a service provider (hereafter called an *agent*) serves its users by running their tasks after they consent. The user is identified in the consent with some identity, so we may require the additional help of an independent identity authentication service. Our aim is to allow *existing* identity authentication services (like well deployed SSO or email providers) to provide undeniable consents, and also to allow methods beyond password-based ones to promote wider adoption, easier usability, and enhanced security, as we will explain. We formalize a framework that mimics real-world consent procedures between a user (called a client), an identity manager, and an agent. This leads us to the concept of cryptographically secure digital consent. Our framework with digital consent is designed to guarantee the security of the user in security-critical applications.

Leveraging widely used online identity providers such as Google or Microsoft is not a new concept. For example, Amazon relies on these identity providers to manage its users. **Our contribution** is an unforgeable proof, held by the agent, that the client consented to the action in question, for an eventual future dispute. We first primarily focus on the application of digital document signing, where a user asks a service provider which keeps a signing key to add a digital signature inside a PDF document. We want the service provider to be liable for documents which have been signed without consent. As another application, we consider a service provider which is a safe repository for the valuable digital secrets of the user, and the action to withdraw or replace a valuable secret would require an explicit consent. Finally, we also consider e-banking application, where the service provider is a bank and the user asks to perform some actions like payments or transfers, but the bank must keep a log with consents. We believe this application will be important with the emerging technologies of agentic systems.

The requirements that we put on our consent notions are natural: first of all, the consent must be **secure** (i.e. non-repudiable). It shall require the user to hold **no specific devices or keys** (that can be damaged, stolen, or forgotten) and require **no effort** on the user side. If such a scheme had required effort, users may have wanted to delegate to another agent and the consent problem would remain. Concretely, we do not require the user to additionally store more than a consent password, if they wish to secure the consent by such mechanism. Our consent notion relies on **existing user authentication means** and shall require **no change** on deployed systems. We can use a TEE in a smartphone as an

*Most of this work was done while the author was a PhD student at EPFL.

authentication means and also leave the option of an alternate mechanism such as SSO as a backup option, in the case of an unavailable device.

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. 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 9.

We assume that the agent is *entitled* to perform actions on behalf of the user, as long as it is bound to collect explicit consents as defined by a contract. The enrollment procedure defines the "contract", i.e. what should be a valid consent. It sets what (combination of) authentications we should use to have a valid consent.

We additionally have to work with several constraints to make the proposed protocol deployable in the real world. First, we must not make any change to the existing IdMs, e.g. compatibility 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, e.g. no changes to certain cryptographic operations such as digital signature verification. Finally, we should not rely on long-term storage on the client side. [1]

Straw-person application. We designed a straw-person application for digital document signing where the user delegates their signing key to the cloud or server (which we refer to as Agent). Whenever a document needs to be signed, the user logs in and requests the Agent to sign the document on their behalf. The Agent 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 three phases: (1) enrollment to the Agent, 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, (2) consent generation, which needs authentication (with another password) to the IdM and a signed OIDC token which serves as an *order*, and (3) consent verification in the case of a dispute, to have the Agent accountable for signing without consent. 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 [DocuSign(2024)].

Security. First and foremost, we need to prevent offline dictionary attacks on the password during the enrollment phase so that neither IdM nor any other client can run it. Second, our protocol must prevent forging the digital consent without the client's involvement 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 design will prevent it by the consent verification, which can be done by (1) an after-action dispute protocol; (2) a trusted back-end; or (3) a specific application.

Our approach. In our design, we aim to rely on a well-established identity provider for the following reasons: (1) we want to separate the identity provider from the service provider to prevent forging requests on behalf of the users while keeping their secrets; (2) IdMs provide user authentication for decades and have established widely known fraudulent account detection and prevention techniques in place; and (3) the deployments of strong authentication methods centered around IdMs strengthened with TOTP, Authenticator, or in general multi-factor authentication. We make the choice of IdM as flexible as possible, we refer to Subsection 9.2 for a discussion about the choice of IdM. Involvement of IdM as another entity makes the consent design different from well-known primitives such as password-based signatures (PBS) or server-aided digital signatures (SADS). PBS involves a client interacting with the service provider server which also behaves as an IdM. They cannot protect against offline dictionary attacks against the server. SADS aims to design protocols in a multi-party server setting in order to solve offline dictionary attacks. Next, we discuss the related work with these different primitives in different applications.

We show how to use existing authentication infrastructures such as OIDC (single-sign-on), CTAP (devices like Yubikey), OIDC4VC (government digital identity), and DKIM (email) as an IdM.

Our contribution. We formalize the abstract digital consent framework and its formal security model. We construct an efficient protocol that can be developed with multiple options; formally prove its security, and implement it. We discuss potential applications. Our analysis in a dynamic corruption model identifies a new type of threats: attacks by users on their future self. The notations and primitives used in the paper is given in Section 3. Related works are summarized in Section 2.

2 Related Works

2.1 Related Primitives

Password-based signatures (PBS). PBS schemes are similar to digital consent: it is a signature scheme which relies on only a password. With consent, we achieve the flexibility of

[1] Except a password which is the most common authentication method used today.

bringing different authentication methods, as opposed to only passwords, as we separate the roles of servers.

In PBS, the client (in possession of a password) and the server jointly compute a signature σ on a message m which would verify under the client's public key. Gjøsteen and Thuen [Gjøsteen and Thuen(2012)] formalized the notion of PBS and provided two constructions based on blind signatures RSA and CL. Sangeetha et al. [Sangeetha Jose and Rangan(2013)] provided a PBS scheme using the BLS signature scheme. They further construct a strongly secure variant of PBS in which smaller entropy passwords can be used without losing security. The requirement of any 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. These constructions [Gjøsteen and Thuen(2012)] and [Sangeetha Jose and Rangan(2013)] are prone to offline dictionary attacks by the server 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 server (outsider attack). See Appendix D for the details.

Server-Aided Digital Signatures (SADS). Earlier, He et al. [He et al.(2005)] introduced server-aided digital signatures (SADS) which utilize a two-server mechanism. The utilization of two non-colluding servers solves the problem of offline dictionary attacks in [Gjøsteen and Thuen(2012)], assuming that at least one server remains non corrupted. SADS use an oblivious PRF (OPRF) with two servers to retrieve a signing key. Hence, it is a key recovery protocol based on two non-colluding dedicated servers. Acar et al. [Acar et al.(2013)] separate the OPRF server and the storage. Likewise, Camenisch et al. [Camenisch et al.(2014)] use several servers. Camenisch et al. [Camenisch et al.(2016)] take a similar approach to SADS where one of the servers is modeled as a user device. In those approaches, there is no flexibility with the authentication method. Furthermore, building up two dedicated servers for the same application and assuming that the servers would not collude is a strong trust assumption. By design, the servers cannot be independent. In our paper, we rather rely on servers which have been independently designed for different applications: identity management and agent.

Password-based credentials (PBC). The PBC notion, introduced by [Zhang et al.(2020)] and strengthened against server compromise by [Dayanikli and Lehmann(2024)] uses a similar terminology for different properties. The client and the server (agent) are the only two participants. After enrollment, the client must keep a high-entropy private credential which allows them to sign with the help of a password. So, the client needs a reliable storage to keep the credential safe. Zhang et al. also apply PBC to federated identity, but their tokens are designated-verifier, whereas a consent must be publicly verifiable.

2.2 Related Applications

Proxy signatures (PS). One of the main applications of digital consent is the signing service where Agent signs documents on behalf of the client. One way for this application to be implemented is to design it with PS [Derler et al.(2014)], [Hanser and Slamanig(2013)], [Fuchsbauer and Pointcheval(2009)]. PS was first formalized by Boldyreva et al. [Boldyreva et al.(2003), Boldyreva et al.(2012)]. PS 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 Almashaqbeh and Nitulescu [Almashaqbeh and Nitulescu(2023)] 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 instead of *any* message. In PS, the delegator needs to keep a high-entropy key to sign a warrant (a consent) 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 unframeability is the lack of recoverability: if the delegator 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. We can have fine grained message-based consent and accommodate any signature scheme, without assuming any key storage on the client side. Finally, we could use PBS instead of PS but it would not be secure against a corrupted server for this application.

Key recovery (KR). Another application of digital consent is the key recovery (or credential recovery), which allows a client to recover their keys without depending on a personal device. KR might be addressed with the notion of credential retrieval which was studied by Boyen [Boyen(2009)]. It allows a client to store their credentials on a server and to retrieve them using a password. Credential retrieval also utilizes OPRFs (as SADS). It 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 (who knows k) 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

this participant is honest when the server (or Agent as we call it) is not.

Client authentication in blockchain. In blockchain technologies, a client casts a transaction order on a smart contract. There is a client authentication which goes beyond regular access control problems, because the authentication process is publicly made. The authors of zkLogin [Baldimtsi et al.(2024)] study the problem of helping a client to make a transaction on a blockchain (or a smart contract running on the blockchain), with the sole help of an identity manager and a few servers. Essentially, making a transaction is proving that a client who successfully identified authorized the transaction to be cast. In zkLogin, this proof is computed by a server and verified by a smart contract. One difference with the PDF signing application is that now, the client cannot miss that the transaction was cast as it appears on the blockchain. With our digital consent formalism, we can have the transaction made by the agent once the consent is given. Moreover, our consent is easily verifiable by a smart contract and can be used like zkLogin [Baldimtsi et al.(2024)]. Their version also needs two servers but uses an expensive ZK-SNARK to provide additional privacy guarantees.

3 Preliminaries and Notation

General Notation. We denote the security parameter by λ (or 1^λ in unary). We write $x \leftarrow\!\$\ S$ for sampling x uniformly at random from a set S, and $x \leftarrow \mathsf{Alg}(\cdot)$ for the output of a deterministic or randomized algorithm. PPT stands for probabilistic polynomial time. $\{0,1\}^\lambda$ denotes the set of λ-bit binary strings. We use $\mathbb{Z}_q$ for the integers modulo q, and $\mathbb{G}$ for a cyclic group of prime order q.

Entities. Our framework involves four entities:

- Client: the client, i.e. the user's device (browser or app) acting on behalf of the user.
- IdM: the Identity Manager, an independent identity authentication service (e.g. an OIDC provider).
- Agent: the Agent (service provider) that executes actions on behalf of the Client.
- Judge: a Judge who verifies the validity of a consent.

Protocol Variables. We use the following variables throughout the paper:

- ID: the identity of the client as known to IdM (e.g. an email address).
- ref: a reference used by Agent to index its database db.
- pw: the consent password chosen by the client.
- *att*: the attribute or action description that the client consents to.
- contract: the contract, publicly defining how to verify a consent for a given client.
- s_A: the agent's secret, stored by Agent alongside contract.
- consent: the consent proof, a publicly verifiable certificate that the client authorized *att*.
- query, resp, order: the query sent to IdM, the response from IdM, and the order sent to Agent, respectively.

Cryptographic Primitives.

- $\Sigma = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$: a digital signature scheme. $(\mathsf{sk},\mathsf{pk}) \leftarrow \mathsf{Gen}(1^\lambda)$ generates a key pair; $\sigma \leftarrow \mathsf{Sign}(\mathsf{sk}, m)$ produces a signature; $\mathsf{Verify}(\mathsf{pk}, \sigma, m) \in \{0,1\}$ verifies it. We require existential unforgeability under chosen-message attacks (EUF-CMA).
- Com: an additively homomorphic commitment scheme. $\mathsf{com} \leftarrow \mathsf{Com}(x_1, x_2; r)$ commits to messages x_1, x_2 with randomness r.
- H: a cryptographic hash function, modeled as a random oracle with output in $\mathbb{Z}_q$.
- H_p: a password hash function, modeled as a random oracle with output in $\mathbb{Z}_q$.
- $\Pi_{\mathsf{MULTEQ}} = (\mathsf{PoK}, \mathsf{Verify})$: a non-interactive zero-knowledge (NIZK) proof system, compiled via Fiat-Shamir from a Σ-protocol. It proves knowledge of a witness for a commitment relation (detailed in Subsection A.3).

Security Game. We denote by CUF the Consent Unforgeability game (Figure 2). The advantage of an adversary $\mathcal{A}$ with password distribution $\mathcal{D}$ is $\mathsf{Adv}^{\mathsf{CUF}}_{\mathcal{A}}(\lambda, \mathcal{D})$.

4 The Consent Protocol Framework

4.1 Entities and Interfaces

We consider a simplified model with the following four entities: Client is the device working for its authentic user who is giving the consent. We do not distinguish the Client and the user for simplicity. IdM is the identity provider that Client can interact with to receive access tokens for their identities.[2] Agent is the entity that acts on behalf of a Client with a given consent. Judge is the entity that verifies the consent.[3] We define the interfaces as follows and depict the flow of the procedures in Figure 1.

[2] 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 9.2.

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

- $(\mathsf{sk}_{\mathsf{IdM}}, \mathsf{pk}_{\mathsf{IdM}}) \leftarrow \mathsf{IdM.Setup}(1^\lambda)$: IdM generates their secret/public keys. It is run by the IdM and re-run every time the keys are rotated.
- $(s_\mathsf{A}, \mathsf{contract}, \mathsf{salt}) \leftarrow \mathsf{Client.Enroll}(\mathsf{ID}, \mathsf{pw}, \mathsf{pk}_{\mathsf{IdM}})$: enroll the Client with password pw to an Agent. Generates the agent secret s_A, contract, and a salt. Send them to the Agent. The contract element publicly defines how a consent can be verified. It is binding.
- $(\mathsf{query}, \mathsf{state}) \leftarrow \mathsf{Client.Launch}(\mathsf{pw}, \mathsf{salt}, \mathit{att})$: Client initiates the consent protocol by sending a query to IdM and keeping the state. Client wants to consent to some action which is non-ambiguously defined by *att*.
- $\mathsf{resp} \leftarrow \mathsf{IdM.Query}(\mathsf{ID}, \mathsf{sk}_{\mathsf{IdM}}, \mathsf{query})$: IdM receives the query from the Client and sends back resp.
- $\mathsf{order} \leftarrow \mathsf{Client.Commit}(\mathsf{state}, \mathsf{resp})$: 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*.
- $\mathsf{consent} \leftarrow \mathsf{Agent.Receive}(s_\mathsf{A}, \mathsf{contract}, \mathit{att}, \mathsf{order})$: Agent receives the order and outputs a consent with $(\mathsf{contract}, \mathit{att})$.
- $0/1 \leftarrow \mathsf{Verify}(\mathsf{contract}, \mathit{att}, \mathsf{consent})$: The verification procedure takes as input the contract and a consent issued on *att*. It returns 1 if the consent is valid, 0 otherwise.

Figure 1 shows the interface between the IdM, Client, and Agent. Client selects a reference ref and a password pw then enrolls to Agent. The value ref is used as a reference to a database by Agent. It can be a login name or an index but it must be unique. ID is the identity under which the Client is known to IdM (e.g. an email address). This could be identical to ref but doesn't need to. We thus keep the distinction between ID and ref.

In the enrollment phase, s_A, salt, and contract are sent to the Agent to store. salt can be retrieved from Agent without access control so not considered private. During the consent phase, Client is required to authenticate to IdM.

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

4.2 Assumptions and Adversarial Model

We assume secure communication between participants. More precisely, the Client $\leftrightarrow$ Agent link is confidential (but the adversary may try to impersonate Client). Typically, we use unilateral TLS connection, which implies trust in the browser and in the PKI to authenticate the certificate of Agent.

Communication between Client and IdM has another type of security assumption. 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 9.2. More precisely, we assume that an adversary cannot interfere with communications between Client and IdM (but may see them). For example, we can have a TLS connection to an OIDC IdM[4] through which Client authenticates, may it be with an out-of-band channel and/or a secure personal device (for 2-factor authentication) or with just a password.

We assume that Judge can trust that contract is valid and deduce from it what is a valid consent by Client to perform an action. The action should be performed by the agent only when the agent is entitled to do so. Our framework defines what makes a consent valid to perform a specific action relative to a contract, but assessing whether a contract is a valid entitlement is an independent problem and specific to the use case. For instance, if contracts are posted on a public ledger, we can have a "soft entitlement" by *transparency* based on public reputation and also on verification by the users themselves. If Agent makes a legal identification of the client when registering the contract, we can have a "hard entitlement" by a *legally binding agreement*. Somehow, we need a legally binding infrastructure to bootstrap the entitlement of contract. We discuss it in Section 11.

Similarly, we assume that Judge can trust *att*: *att* has a non-ambiguous semantic interpretation that faithfully reflects the user interpretation. This is a similar problem as for messages in a digital signature.

Our protocol to consent to some action *att* could be replayed. Hence, *att* is assumed to specify if it relates to an action which must be executed only once. To make a consent for an action single-use, we need to include a timestamp, a transaction number, or a nonce in *att*.

We model the combination user/app/OS/device as a single participant Client. Hence, we assume that the user can trust their equipment to insert the right *att* in the protocol input and not to disclose pw.

For each participant, we assume that the storage does not leak if not corrupted. Actually, the adversary can take control of any participant either temporally or permanently. Crucially, *our corruption model is dynamic*. Moving a participant from a corrupt state to an honest state (i.e., healing) implies wiping its memory to restart from a clean state.

We assume that the adversary has the means to run an offline dictionary attack when a password test information leaks. We will relax this assumption in Section 6 (and only there) by using a slow password hash function.

Finally, we assume that during the lifetime of a contract, either IdM remains uncorrupted or Agent remains uncorrupted. This is justified by the fact that both participants are independent, and we can assume that contracts would be amended once the corruption of either becomes known. However, attacks are possible during the gap between the corruption of

[4] for example, https://accounts.google.com/o/oauth2/v2/auth

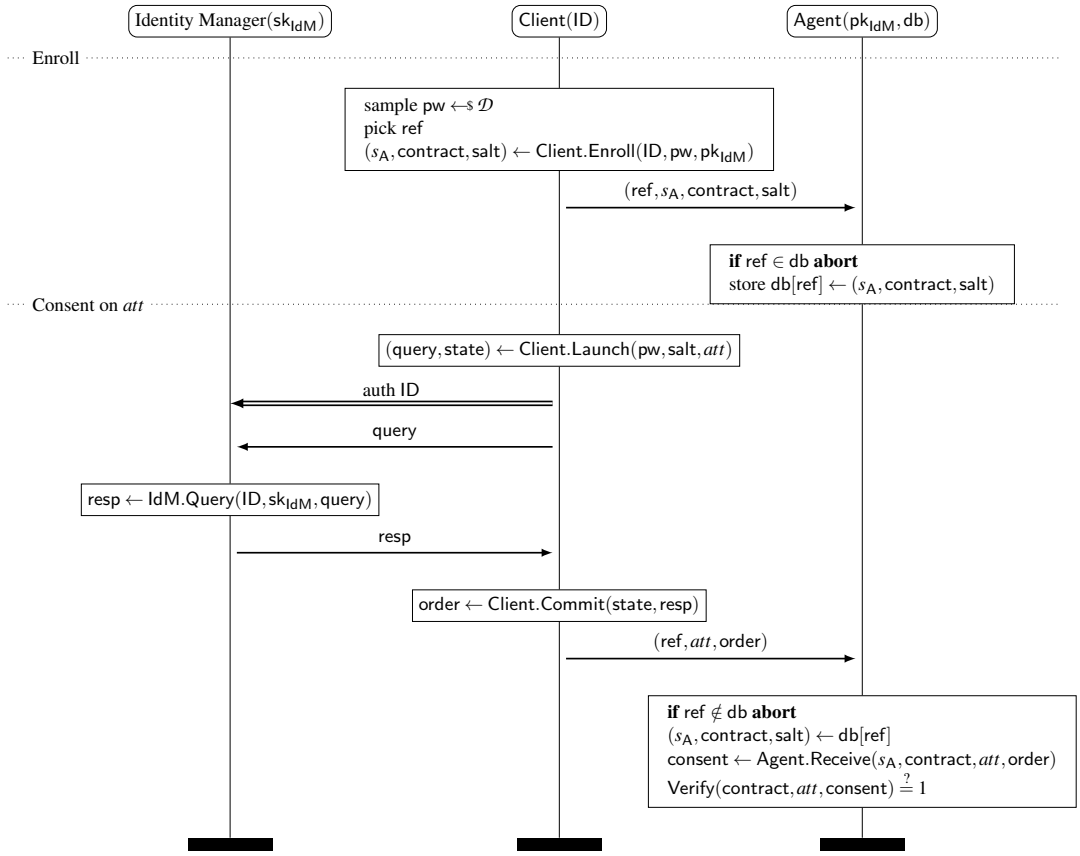

Figure 1: Consent Protocol Interface

the second participant and the contract amendment.

4.3 Unforgeability

Figure 2 models Consent UnForgeability (CUF). The adversary interacts with several oracles which model what honest participants do with adversarially provided inputs. The honest participants can be IdM, Agent, or the target victim ID^*. Any other participant is malicious and run by the adversary.

The adversary has the power to *adaptively* corrupt all parties at any time, including ID^*. Our goal is to protect an honest client ID^* from having a valid $\mathsf{consent}^*$, relative to some $\mathsf{contract}^*$ that they agreed on, to some action att^* which was not intended by them to consent, shown to an honest judge.

To formalize corruption, we use a set Corrupted specifying which participants among IdM, ID^*, and Agent are corrupted. Jumping ahead, what is important is the status of Corrupted from when a challenge contract $\mathsf{contract}^*$ is enrolled until the end of the game. We will require that ID^* is honest during this entire period, as well as either IdM or Agent.

To model *dynamic corruption*, Corrupted is populated based on corruption. It is originally empty but can change during the game. An honest participant P can be corrupted using the $\mathsf{OCorrupt}(P)$ oracle which reveals its memory mem_P and changes its corruption status. A corrupted participant can be healed with the $\mathsf{OHeal}(P)$ oracle, which erases mem_P and changes its corruption status: it resets P. In a *static corruption* model, Corrupted is set once for all as an input of the game.

An honest IdM is modeled by two oracles: OIdMInit, OIdMQuery. The latter models a query made by a corrupted client. The memory $\mathsf{mem}_{\mathsf{IdM}}$ contains the sensitive $\mathsf{sk}_{\mathsf{IdM}}$.

An honest Agent is modeled by the oracles OAgentEnroll and OAgentOrder. The memory $\mathsf{mem}_{\mathsf{Agent}}$ contains the dictionary db of contracts and associated secrets s_{A} and salt. The dictionary is keyed by a unique identifier ref of the client.

The honest victim ID^* is modeled by the oracles OClientEnroll, OClientLaunch, and OClientCommit. One particularity is that we assume an authenticated channel between ID^* and IdM, so OClientLaunch also runs IdM receiving the query if IdM is honest. Otherwise, a malicious resp must be sent to OClientCommit. However, query is always returned (i.e. we do not assume encryption with IdM). Similarly, we assume an encrypted channel between ID^* and Agent (but no authentication). So these two oracles also simulate Agent receiving the order if Agent is honest. The memory $\mathsf{mem}_{\mathsf{ID}^*}$ contains the existing passwords pw.

We also assume that when ID^* and Agent are honest, Agent will not let ID^* register a ref which is already used. This

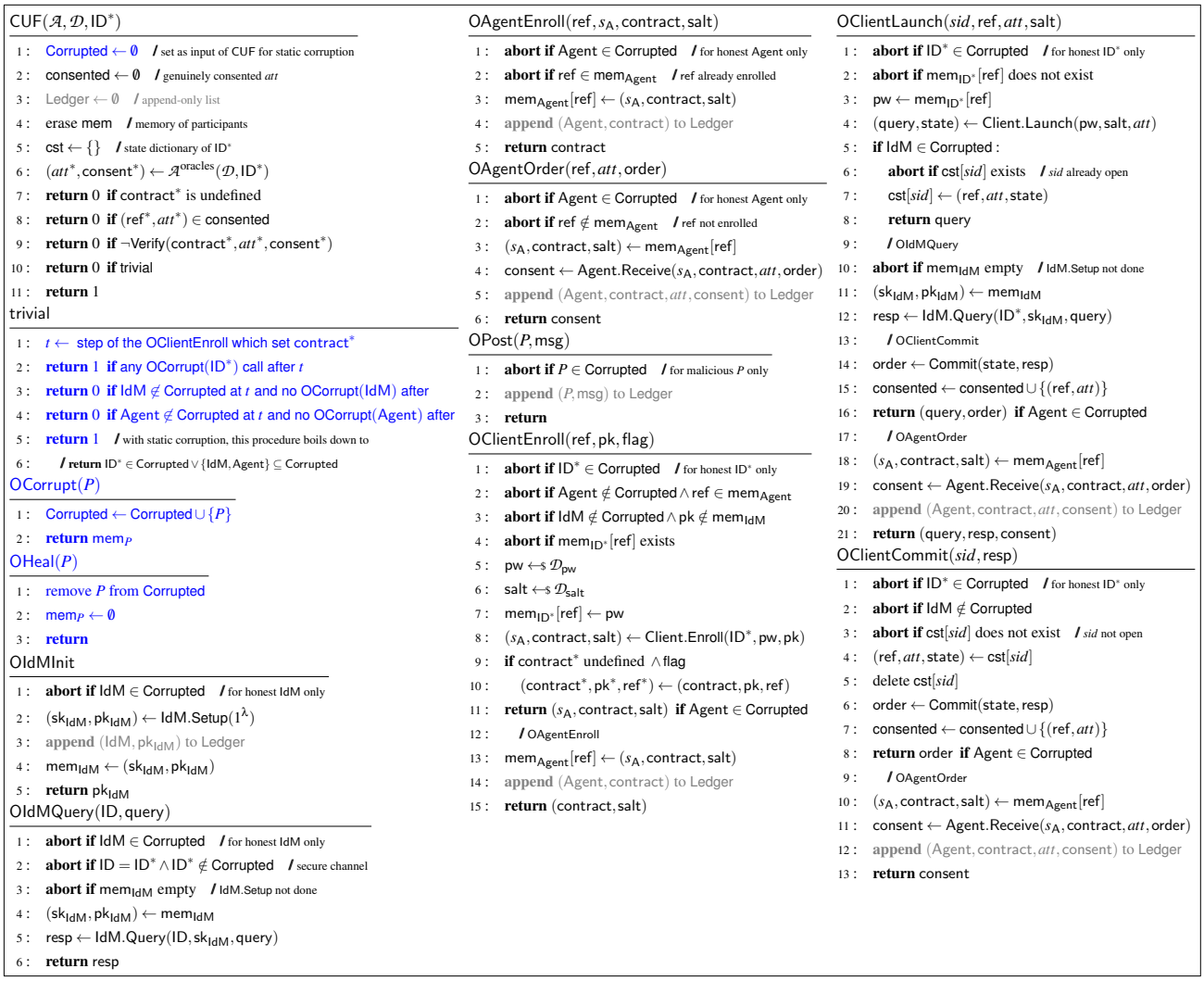

Figure 2: CUF Game. Code in blue is specific to the dynamic corruption model.

is why OClientEnroll rejects when Agent $\notin$ Corrupted and ref $\in \mathsf{mem}_{\mathsf{Agent}}$. Similarly, we assume that an honest ID^* will always get the correct $\mathsf{pk}_{\mathsf{IdM}}$ if IdM is honest, thanks to the secure channel assumption between them. This is why the oracle rejects when IdM $\notin$ Corrupted and pk $\notin \mathsf{mem}_{\mathsf{IdM}}$.

OClientEnroll takes as input flag. It indicates when the adversary requires the creation of the target $\mathsf{contract}^*$.

Our game uses an append-only ledger which keeps a log of important events, such as a new $\mathsf{pk}_{\mathsf{IdM}}$, a new contract, or a new consent. Any participant P can post messages msg in the ledger in the form (P,msg). If P is not corrupted, only the appropriate oracle which models P can do it. Otherwise, the adversary can post using the OPost oracle. The ledger only plays a role in the discussion of Section 11.

The game also takes as input the distribution $\mathcal{D} = (\mathcal{D}_{\mathsf{pw}},\mathcal{D}_{\mathsf{salt}})$ of pw and salt which play a role in security.

The trivial predicate (which is used in Line 10 of CUF) says if an attack is outside of our model. Essentially, this is where our assumption that we cannot corrupt both IdM and Agent during the lifetime of $\mathsf{contract}^*$ appears. When considering dynamic corruption, trivial inspects the history of oracle calls in the game together with $\mathsf{contract}^*$, att^*, and $\mathsf{consent}^*$. At any step, the predicate can determine the current Corrupted set. It checks which OClientEnroll oracle call (if any) defines the challenge ref^*, $\mathsf{contract}^*$, and pk^*. The predicate trivial checks the following properties (if one is not satisfied, the predicate is true).

- There is one non-aborting OClientEnroll oracle call with input flag raised. It defines $\mathsf{contract}^*$ which we call the challenge. This step starts the "challenge period" which continues until the end of the game.
- There is no OCorrupt(ID^*) oracle call during the challenge period.

- Either IdM is honest when the challenge is set and there is no OCorrupt(IdM) during the challenge period (we call it the malicious Agent case), or Agent is honest when the challenge is set and there is no OCorrupt(Agent) call during the challenge period (we call it the malicious IdM case).

The first condition makes sure that contract* is honestly registered. The second makes sure that the corresponding pw does not leak by corrupting ID* (otherwise, forging a consent using the oracles becomes trivial). The last condition is necessary and sufficient to prove the security of our protocol. Indeed, a corrupted Agent can recover pw by offline dictionary attack when the entropy of $\mathcal{D}_{\mathsf{pw}}$ is small[5], and a corrupted IdM reveals $\mathsf{sk}_{\mathsf{IdM}}$. With both pw and $\mathsf{sk}_{\mathsf{IdM}}$, forging a consent is trivial during the challenge period.

We also define a *static corruption* model by removing the corruption/heal oracles OCorrupt/OHeal and setting Corrupted in advance. In the static corruption model, the trivial predicate boils down to checking that Corrupted is allowed in the model (i.e., ID* is not in and IdM and Agent are not simultaneously in).

We note that in the malicious agent case defined by trivial, it must be the case that the used pk in the challenge was honestly set during an OIdMInit oracle call and that IdM remained uncorrupted after that.

Definition 1 (Consent Unforgeability) *Let $\mathcal{D}$ be a distribution for* (pw, salt) *and* ID* *be the target victim client. In the* CUF *game defined in Figure 2, we define the advantage of $\mathcal{A}$ as follows:*

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

A protocol is $\mathcal{D}$-CUF-secure if for any ID*, $\mathsf{Adv}^{\mathsf{CUF}}_{\mathcal{A}}(\lambda, \mathcal{D}, \mathsf{ID}^*)$ *is negligible.*

Non-repudiation. Consider a case where a triplet (contract, *att*, consent) is valid but ID*, who registered contract under reference ref, claims that they did not consent on *att*. Assuming that IdM and Agent were not both corrupted during the lifetime of contract, unforgeability implies that the attack must be trivial, which under these assumptions means that ID* must be corrupted. So, there is no plausible repudiation.

5 Our Protocol Design

We present our construction next. It relies on a commitment scheme and a zero-knowledge protocol which are introduced in Appendix A in Appendix.

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. We only assume that IdM.Setup generates a signature key pair and that IdM.Query returns a signature of ID and query (possibly with extra information) after authenticating Client. In practice, query can be embedded into a nonce which is signed. We only require that the domain of this nonce is large enough to embed a collision-resistant hash (e.g. a 256-bit string).

In Figure 3, Client.Enroll computes the salted hash of pw and commits on that with randomness r_{ID}. The hash and r_{ID} form the agent's secret s_{A}. The contract is specified as ID (as known by the IdM), the commitment on the salted hash of the password, and the public key of the IdM.

Essentially, Client commits to pw and salt with r_{ID} and the commitment is put inside contract. The value s_{A} includes the opening information r_{ID} for the commitment. To consent to *att*, Client commits to the salted hash of pw and to *att* (with another opening randomness r) and the commitment is sent as a nonce to 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 contract and the signed commitment commit to the same secret pw. The final consent consists of the signed commitment and this proof. The opening, r, is then destroyed.

Zero-knowledge is needed to avoid offline dictionary attacks on pw once consent is published. The commitment is additive homomorphic to enable zero-knowledge proofs.

We do not require Client to keep salt safe. We assume that salt can be easily obtained from Agent without security. Hence, the adversary knows salt and can even change it in OClientLaunch. The role of salt is to defeat the attack from self with dynamic corruption as detailed in Section 7. In the idmless variant, it also avoids multi-target attacks.

We recall that it is essential that the communication between Client and Agent is confidential, as it would reveal $H_p(\mathsf{pw}, \mathsf{salt})$ otherwise. This typically implies a TLS connection with a trusted PKI. Instead of a hash, we could use a PRF with a key held by Agent to relax this assumption. But it would require an additional round-trip communication in order to evaluate them in an oblivious manner. Furthermore, to relax the confidentiality assumption, the value taking the role of h_{ID} would need to change for each consent (as $h_{\mathsf{ID}} \leftarrow \mathsf{PRF}(\mathsf{pw}, att)$), which would make the computation of the zero-knowledge proof π more expensive.

The client runs two algorithms in the Consent phase in Figure 3: Client.Launch, and Client.Commit. Client.Launch 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 Client.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

[5] Except in the variant of Section 6.

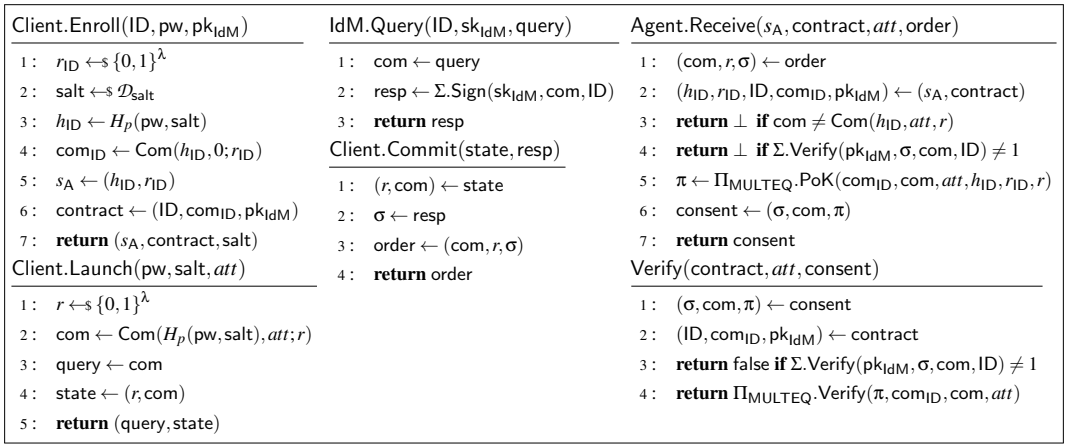

Figure 3: Consent Protocol

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 [Baldimtsi et al.(2024)].

Another way to see this approach is to consent on the creation of a sub-contract contract′ to consent on all att_i. In this sub-contract, there is no pw′, s'_A is the consent within contract, IdM′ is run by Client, and resp = order. The consent on att_i becomes the order_i concatenated with s_A.

Post-Quantum Considerations. The underlying primitives that we use need to be post-quantum resistant in order for our protocol to resist to quantum attacks. For that the signature algorithm by IdM must be quantum-resistant. Similarly, the commitment scheme and the proof-of-knowledge should be replaced with PQ-safe variants. We need an additively homomorphic commitment, so one of the BDLOP [Baum et al.(2018)] or combination of Ajtai and BDLOP, i.e. ABDLOP as described in [Lyubashevsky et al.(2022a)], would work, depending on whether we encode $H_p(\mathsf{pw}, \mathsf{salt})$ and *att* with small coefficients or not. We need PQ binding and hiding for commitments and PQ soundness and zero-knowledge for the proof. These properties can be borrowed from sophisticated proofs based on lattices [Lyubashevsky et al.(2022b), Lyubashevsky et al.(2022a)]. The adaptation of our protocol to a PQ-safe frame is left to future work.

Hash Domain Separation. Our protocol uses a hash function at two places which must be domain-separated: to hash a password $H_p(\mathsf{pw}, \mathsf{salt})$ and to compute a challenge $H(\mathsf{input})$ in the non-interactive zero-knowledge proof. In practice, we can either use different hash functions (as we write H_p and H) or use the same one H with explicit domain separation, e.g. $H(\text{“pw:”} \| \mathsf{pw}, \mathsf{salt})$ and $H(\text{“zk:”} \| \mathsf{input})$. Using a (slow) password hash function H_p is not necessary as clients whose passwords have been cracked by offline dictionary attacks from Agent (who is the only participant who can do that) are still protected by IdM. In Section 6, using a slow hash function H_p becomes critical as there is no IdM any more.

Embedding query **into a nonce.** The query to be signed is the result of a hiding and binding commitment. If the nonce space is larger than the commit value space, a simple injective encoding can be used. Otherwise, the commitment must be followed by a collision-resistant hash, which is equivalent to changing the commitment scheme while preserving the hiding and binding property.

6 Variant without IdM

As described on Figure 4, we can build a protocol without IdM which is secure against a malicious Agent. To replace the additional protection by IdM, the idea is to use a password hash function H_p such as Argon2 which protects against offline dictionary attacks. This password hash returns the seed of a pseudorandom generator which is used to issue a signature key pair. The verification key is given to the Agent. Then, the client can sign *att*.

We assume that the Agent cannot run an offline dictionary attack due to the complexity of H_p. However, when Agent is honest, there may be organizations with more computational power. For that reason, we still exclude to expose the verification key to the other participants by committing to it by $\mathsf{com}_{\mathsf{ID}} = \mathsf{Com}(\mathsf{pk}_{\mathsf{ID}}; r_{\mathsf{ID}})$. So, a consent is a proof of knowledge of a valid signature which verifies under a hidden verification key which is in a commitment.

One H_p computation is costly but it is only needed once for the user.[6] Typically, a user would use a personal device which would cache $\mathsf{sk}_{\mathsf{ID}}$ in order not to redo the H_p evaluation too often. The H_p computation would be needed once a while

[6] RFC 9106 recommends parameters so that one Argon2 computation takes between half a second to a couple of seconds, depending on the platform.

to reset a device or setup a new one. However, an adversary would need many H_p computations to break the scheme.

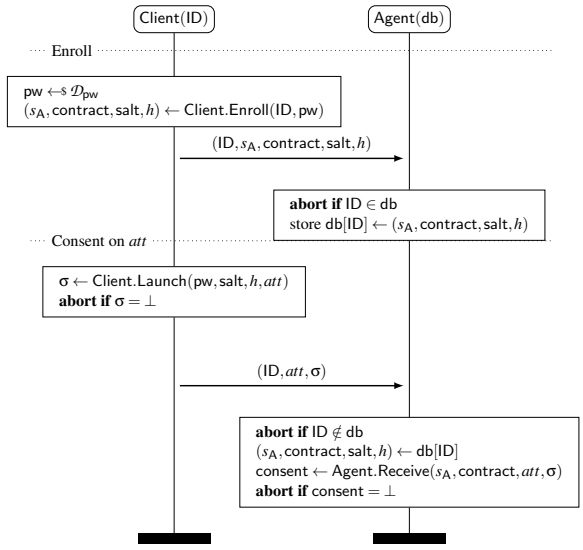

Figure 4: Consent Protocol without IdM

Using the Schnorr signature, the signature is a pair $\sigma = (R, s)$ with $R = r \cdot G$ and $s = r + x \cdot H(\text{“sig:”}, R, att)$. The consent includes R and a proof-of-knowledge of (s, r_{ID}) such that $\mathsf{Com}(s \cdot G, r_{ID}) = \mathsf{Com}(R; 0) + H(\text{“sig:”}, R, att) \cdot \mathsf{com}_{ID}$ using the homomorphic property of Com. The algorithm Verify(contract, att, consent) extracts com_{ID} from contract and R from consent then verifies the proof in consent.

For the homomorphic commitment to the group element pk_{ID}, we can set $r_{ID} = (\rho_1, \rho_2)$ and $\mathsf{com}_{ID} = (\mathsf{pk}_{ID} + \rho_1 \cdot G, \rho_1 \cdot G' + \rho_2 \cdot G)$ which is perfectly hiding and computationally binding. Here, $\mathsf{pk}_{ID} + \rho_1 \cdot G$ is a blinding of pk_{ID} and $\rho_1 \cdot G' + \rho_2 \cdot G$ is a Pedersen commitment on the mask ρ_1.

The full protocol is specified on Figure 5.

We describe two simplified security games on Figure 6. It is based on CUF with Corrupted $= \{\mathsf{Agent}\}$ and Corrupted $= \{\mathsf{IdM}\}$. We remove IdM in the game, as well as OCorrupt and the Corrupted set which is no longer needed. We can now use ID $=$ ref. We however enrich the game to capture multitarget attacks, instead of a pre-determined target ID^*.

Resistance to multi-target attacks. We avoid (too slow) offline dictionary attack to perform linearly better by having a large set of possible targets. For this, we must prevent the adversary to force the client to set its salt to a unique one.[7] We do so by adding a redundancy h to salt so that it can be verified by Client. The client who has lost their salt retrieves salt and h and verifies them.

[7] A malicious Agent could indeed give a unique salt to every client when they have lost it, ask them to consent on a dummy att, and collect signatures. Then, a multi-target offline dictionary attack would eventually find one password pw and for a consent of the respective client and allow to forge new consents.

Honest Agent **model.** In a model where Agent is honest, we protect against a more powerful offline adversary which can run dictionary attacks. In this model, the adversary can only access to contract (so to com_{ID}) and consent (which is zero-knowledge) with a chosen att. That is, OClientEnroll only returns $\mathsf{contract}_{ID}$ and OClientLaunch only returns consent. There, the hiding property of the commitment prevents to get any information on the password. In practice, we also need to assume that Client authenticates to Agent through an asymmetric PAKE before withdrawing salt and h.

7 Attack by Self in the Dynamic Model

Our security analysis in the dynamic corruption model succeeds to identify a counter-intuitive threat: a corrupted user may try to attack their future honest self. There are situations where this scenario could make sense. For instance, an abusive spouse with access to the credentials of their partner anticipate the need to consent on some action which will be done after they separate, even if their partner reset their account with their agent.

In the context of our digital consent protocol, a corrupted user with long-term identity ID registered on IdM may try to play with the OIdMQuery oracle to collect data. More precisely, with several (pw, salt) trials from a dictionary and a target att, the adversary simulates Client.Launch to get a query to be submitted to OIdMQuery, after which Client.Commit is simulated. This provides one (h_{ID}, order) pair per trial. This is a preliminary attack phase. After healing the user and enrolling with OClientEnroll with flag on, the attack can start. Assuming Agent is corrupted, a consent on att may be forged. Indeed, OClientEnroll returns contract and s_A, which includes h^*_{ID}. If h^*_{ID} corresponds to one collected pairs, the adversary can simulate Agent.Receive to get a valid consent for att. The probability of success is the probability that the (pw, salt) selected by the honest user at enrollment belongs to the initial dictionary. When there is no salt, this probability can be large in practice. However, the attack is defeated by using a large enough salt which cannot be predicted in this attack. In pseudocode, the attack works as follows:

Input: $(\mathcal{D}, \mathsf{ID}^*)$
1: select a target att
2: prepare a dictionary for $\mathcal{D}$
3: $D \leftarrow \emptyset$
4: $\mathsf{pk}_{IdM} \leftarrow$ OIdMInit
5: OCorrupt(ID^*)
6: **for** each (pw, salt) ∈ Dictionary **do**
7: (query, state) ← Client.Launch(pw, salt, att)
8: resp ← OIdMQuery(ID^*, query)
9: order ← Client.Commit(state, resp)
10: $h_{ID} \leftarrow H_p$(pw, salt)
11: $D[h_{ID}] \leftarrow$ order
12: **end for**

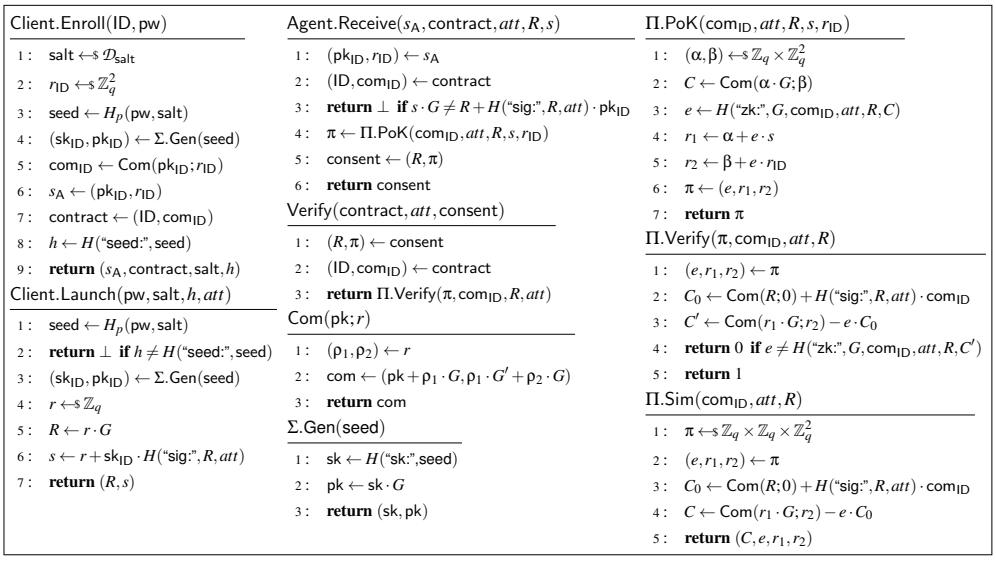

Figure 5: Consent Protocol without IdM

13: OHeal(ID*)
14: select ref arbitrarily
15: OCorrupt(Agent)
16: $(s_{\mathsf{A}}, \mathsf{contract}^*, \mathsf{salt}^*) \leftarrow$ OClientEnroll(ref, $\mathsf{pk}_{\mathsf{IdM}}$, true)
17: $(h^*_{\mathsf{ID}}, r^*_{\mathsf{ID}}) \leftarrow s_{\mathsf{A}}$
18: abort **if** $D[h^*_{\mathsf{ID}}]$ does not exist
19: order ← $D[h^*_{\mathsf{ID}}]$
20: consent ← Agent.Receive(s_{A}, contract*, att, order)
21: **return** (att, consent)

The attack can extend to no corrupted agent. However, the adversary must guess which pair to use. This attack complements the one with a malicious IdM trying to guess (pw, salt) in an online dictionary attack, but with a corrupted ID* instead of a corrupted IdM.

Our above attack scenario can break protocols such as zkLogin with no dictionary and 100% success rate, because there is no pw and no salt to guess during the preliminary phase. We can have a corrupted user authenticating with IdM to get a JWT, then heal and register on the zkLogin platform. Then, a corrupted zkLogin platform can use the JWT to forge a consent. There are two obstacle to use this theoretical attack in practice. First of all, the JWT has often a short validity period. So what justifies an action done at some time t must correspond to a consent based on a JWT which was valid at time t. For some applications, the Agent can just do an action at the time IdM is queried and forge consent later on, or just falsify the time when the action is done. In the case of zkLogin, the action is done by a smart contract on a blockchain, so the time when it is done cannot be falsified, and consent must be provided at this time. That is, the application is of Type III, following the terminology in Section 9. However, the security now relies on an additional security assumption: that IdM has a clock which is synchronized with the blockchain.

8 Security of Our Protocols

We provide here the security results.

Theorem 1 *Let* Com *be an additively homomorphic, computationally hiding and binding commitment scheme,* Σ *be a signature scheme which is secure against existential forgeries,* H *be a hash function (random oracle) with output domain* $\mathbb{Z}_q$, E_{pw}, E_{salt}, *and* E_{Com} *be the min-entropy of* pw, salt, *and* $\mathsf{Com}(H(\mathsf{pw}, \mathsf{salt}), 0; 0)|\mathsf{salt}$ *respectively, and* Π_{MULTEQ} *be our NIZK proof in the random oracle model. For any* PPT *adversary* $\mathcal{A}$ *playing the* CUF *game, there exist* PPT *adversaries* $\mathcal{B}, \mathcal{B}', \mathcal{C}, \mathcal{C}', \mathcal{D}, \mathcal{E}$ *such that*

$$\begin{aligned}
\mathsf{Adv}^{\mathsf{CUF}}_{\mathcal{A}}(\lambda, \mathcal{D}) &\leq 2\frac{(q_{\mathsf{H}} + q_{\mathsf{Launch}}) \cdot q_{\mathsf{Launch}}}{q^2} \\
&+ 2\frac{q_{\mathsf{H}}}{q} + \frac{q_{\mathsf{Order}}}{2^{E_{\mathsf{Com}}}} + \frac{q_{\mathsf{H}}}{2^{E_{\mathsf{pw}} + E_{\mathsf{salt}}}} \\
&+ (q_{\mathsf{Launch}} + q_{\mathsf{Enroll}}) \cdot \mathsf{Adv}^{\mathsf{HIDE}}_{\mathcal{B}'} + q_{\mathsf{Init}} \cdot \mathsf{Adv}^{\mathsf{UF}}_{\mathcal{C}} \\
&+ \sqrt{q_{\mathsf{H}} \cdot \left(\frac{1}{q} + \mathsf{Adv}^{\mathsf{BIND}}_{\mathcal{B}} + \mathsf{Adv}^{\mathsf{HIDE}}_{\mathcal{C}'}\right)} \\
&+ \sqrt{q_{\mathsf{H}} \cdot \left(2\mathsf{Adv}^{\mathsf{BIND}}_{\mathcal{D}} + \sqrt{q_{\mathsf{Query}} \cdot \left(\mathsf{Adv}^{\mathsf{BIND}}_{\mathcal{E}} + \frac{1}{q}\right)}\right)}
\end{aligned}$$

where q_{H}, q_{Enroll}, q_{Launch}, q_{Order}, q_{Init}, *and* q_{Query} *are the number of queries to the random oracle* H, OClientEnroll, OClientLaunch, OAgentOrder, OIdMInit, *and* OIdMQuery, *respectively.* $\mathsf{Adv}^{\mathsf{HIDE}}$ *and* $\mathsf{Adv}^{\mathsf{BIND}}$ *are advantages to break*

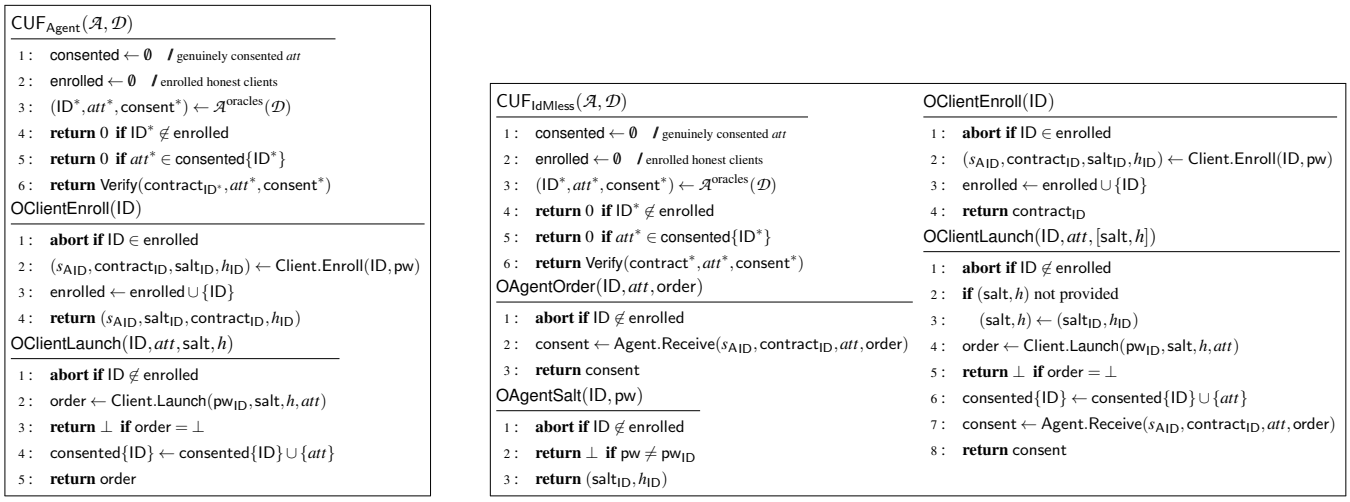

Figure 6: $\mathsf{CUF}_{\mathsf{Agent}}$ Game (with a corrupted Agent) and $\mathsf{CUF}_{\mathsf{IdMless}}$ Game (with an honest Agent) without IdM

the hiding and binding properties of Com. $\mathsf{Adv}^{\mathsf{UF}}$ *is the advantage to forge signatures* Σ.

We can take $q_{\mathsf{Enroll}} + q_{\mathsf{Launch}} + q_{\mathsf{Order}} + q_{\mathsf{Query}}$ as an *online complexity* and q_{H} as something bounded by the (offline) time complexity. Assuming a conservative $q \sim 2^{2\lambda}$ and $E_{\mathsf{salt}} \geq \lambda$, the dominant terms are $\sqrt{q_{\mathsf{H}} \cdot \mathsf{Adv}^{\mathsf{Com}}}$ and $\sqrt[4]{q_{\mathsf{H}}^2 \cdot q_{\mathsf{Query}} \cdot \mathsf{Adv}^{\mathsf{BIND}}}$, where $\mathsf{Adv}^{\mathsf{Com}}$ is about the security of the commitment scheme. These bounds are not tight. They come from the weak simulation extractability with $d = 2$, as defined in Def. 2. We could have better bounds using an online extractable NIZK, but with a higher complexity.

The term $q_{\mathsf{Order}} 2^{-E_{\mathsf{Com}}}$ models a malicious IdM making an online dictionary attack by guessing $\mathsf{com}_0 = \mathsf{Com}(H_p(\mathsf{pw},\mathsf{salt}),0;0)$ with salt known and $\mathsf{pw} \leftarrow\!\$\ \mathcal{D}_{\mathsf{pw}}$, then sending an order with $\mathsf{com} = \mathsf{com}_0 + \mathsf{Com}(0, att)$ to Agent to forge a consent for att. In practice, E_{Com} is identical to E_{pw}, as discussed in the proof (in the Game $\mathsf{G5}_{\mathsf{IdM}}$).

The terms $q_H 2^{-E_{\mathsf{pw}} - E_{\mathsf{salt}}}$ and in q_{Query} model the attack on Section 7. As we can see, salt plays an important role in this term. However, in a static corruption model, these terms disappear.

The term in $\sqrt{q_{\mathsf{Query}} \cdot \left(\mathsf{Adv}^{\mathsf{BIND}}_{\mathcal{E}} + \frac{1}{q}\right)}$ actually measures the advantage of an adversary who first commits with q_{Query} values com, gets as a challenge a random $h \leftarrow\!\$\ \mathbb{G}$, and finally opens one of the commitments to h. (See the game $\mathsf{G9}_{\mathsf{Agent}}$ in the proof.) The bound that we used to relate to the binding property is certainly not tight.

Theorem 2 *With the same assumptions as in Theorem 1 but in a static corruption model, the bound becomes*

$$\mathsf{Adv}^{\mathsf{CUF}}_{\mathcal{A}}(\lambda,\mathcal{D}) \leq \frac{(q_{\mathsf{H}} + q_{\mathsf{Launch}}) \cdot q_{\mathsf{Launch}}}{q^2} + \frac{q_{\mathsf{H}}}{q} + \frac{q_{\mathsf{Order}}}{2^{E_{\mathsf{Com}}}} + (q_{\mathsf{Launch}} + q_{\mathsf{Enroll}}) \cdot \mathsf{Adv}^{\mathsf{HIDE}}_{\mathcal{B}'} + \sqrt{q_{\mathsf{H}} \cdot \left(\frac{1}{q} + \mathsf{Adv}^{\mathsf{BIND}}_{\mathcal{B}} + \mathsf{Adv}^{\mathsf{HIDE}}_{C'}\right)}$$

when IdM *is corrupted and*

$$\mathsf{Adv}^{\mathsf{CUF}}_{\mathcal{A}}(\lambda,\mathcal{D}) \leq \frac{(q_{\mathsf{H}} + q_{\mathsf{Launch}}) \cdot q_{\mathsf{Launch}}}{q^2} + \frac{q_{\mathsf{H}}}{q} + (q_{\mathsf{Launch}} + q_{\mathsf{Enroll}}) \cdot \mathsf{Adv}^{\mathsf{HIDE}}_{\mathcal{B}'} + q_{\mathsf{Init}} \cdot \mathsf{Adv}^{\mathsf{UF}}_{C} + \sqrt{q_{\mathsf{H}} \cdot 2\mathsf{Adv}^{\mathsf{BIND}}_{\mathcal{D}}}$$

when Agent *is corrupted.*

Proof 1 (Sketch) *The full proof is provided in Appendix B. We first simplify the game to have a single* ref *in oracles and neither* ID* *nor* Agent *oracles before the challenge phase. Then, we simulate the proof* π *in the* Agent *algorithms in oracles, by programming the random oracle. We then split the game analysis into two cases.*

In the honest Agent *case, due to our simulation, the randomness r (used to make* com *in* query*) isn't needed; we can replace* com *by a random commitment - no use of* h_{ID}. *The* h_{ID} *is still used in the verification by* Agent *but we can reduce it to a guess game for* $\mathsf{Com}(h_{\mathsf{ID}},0;0)$. *When we don't guess it, we can remove the verification by* Agent. *Then, we use the extractor of the proof (aka forking lemma) to obtain a witness for the proof* π *in the final consent. Finally, we use the extracted witness to break the binding property of* Com.

In the honest IdM *case, we first use the unforgeability of the signature to reduce to a case where the final consent is made by one of the oracles. Then, we use the extractor to get a witness of the proof* π *in the final consent. After eliminating trivial cases, we end up in a game where a corrupted* ID^* *asks* IdM *to sign several commitments, then heals, honestly registers a password, and succeeds to open one of the previous commitments to* h^*_{ID}.

Theorem 3 *We consider the* IdM*less protocol in Figure 5 in the random oracle model. For any* PPT *adversary* $\mathcal{A}$ *playing the* $\mathsf{CUF}_{\mathsf{Agent}}$ *and* $\mathsf{CUF}_{\mathsf{IdM}less}$ *games on Figure 6, there exist* PPT *adversaries* $\mathcal{B}, \mathcal{C}, \mathcal{D}$ *such that*

$$\mathsf{Adv}^{\mathsf{CUF}_{\mathsf{Agent}}}_{\mathcal{A}}(\lambda, \mathcal{D}) \leq$$
$$\frac{q_{\mathsf{Enroll}}(q_{\mathsf{Enroll}}-1)}{2\cdot 2^{E_{\mathsf{salt}}}} + \frac{q_{H_p}(q_{H_p}-1)}{2\cdot 2^{2\lambda}} + \frac{q_H(q_H-1)}{2q} +$$
$$q_{\mathsf{Launch}}\cdot\left(\frac{1}{q}+\frac{1}{2^{2\lambda}}\right) + \frac{q_{H_p}+q_{\mathsf{Launch}}}{2^{E_{\mathsf{pw}}}} + \mathsf{Adv}^{\mathsf{mKR}}_{\mathcal{B}}(q_{\mathsf{Enroll}}) +$$
$$\frac{q_H}{q} + \sqrt{q_H\cdot\left(\mathsf{Adv}^{\mathsf{BIND}}_{C} + \mathsf{Adv}^{\mathsf{mUF}}_{\mathcal{D}}(q_{\mathsf{Enroll}})\right)}$$

and

$$\mathsf{Adv}^{\mathsf{CUF}_{\mathsf{IdM}less}}_{\mathcal{A}}(\lambda, \mathcal{D}) \leq \frac{(q_H+q_{\mathsf{Launch}}+q_{\mathsf{Order}})^2}{q^2}$$
$$+ q_{\mathsf{Enroll}}\cdot\mathsf{Adv}^{\mathsf{HIDE}}_{\mathcal{B}} + \frac{q_H+q_{\mathsf{Launch}}+q_{\mathsf{Order}}}{q}$$
$$+ \sqrt{(q_H+q_{\mathsf{Launch}}+q_{\mathsf{Order}})\cdot\left(\mathsf{Adv}^{\mathsf{HIDE}}_{C} + \mathsf{Adv}^{\mathsf{BIND}}_{\mathcal{D}}\right)}$$
$$+ \frac{q_{H_p}(q_{H_p}-1)}{2\cdot 2^{2\lambda}} + \frac{q_H(q_H-1)}{2q} + 2\frac{q_{H_p}\cdot q_{\mathsf{Enroll}}+q_{\mathsf{Launch}}}{2^{E_{\mathsf{salt}}}}$$
$$+ \frac{q_{\mathsf{Enroll}}\cdot q_H}{2^{2\lambda}} + \frac{q_{\mathsf{Order}}}{q} + \frac{q_{\mathsf{Salt}}}{2^{E_{\mathsf{pw}}}} + \frac{q_H\cdot q_{\mathsf{Enroll}}}{2^{2\lambda}} + \frac{q_{\mathsf{Order}}}{q}$$

where q_{H_p}, q_{H}, q_{Enroll}, q_{Order}, q_{Salt}, *and* q_{Launch} *are the number of queries to the random oracles* H_p *and* H, *and to the oracles* OClientEnroll, OAgentOrder, OAgentSalt, *and* OClientLaunch, *respectively.* $\mathsf{Adv}^{\mathsf{mKR}}(q_{\mathsf{Enroll}})$ *and* $\mathsf{Adv}^{\mathsf{mUF}}(q_{\mathsf{Enroll}})$ *are the advantages of adversaries playing the key recovery and the forgery games against the Schnorr signature with* q_{Enroll} *possible targets (see Figure 7).*

As we can see, the $\mathsf{CUF}_{\mathsf{Agent}}$ security result has an extra (offline) term $q_{H_p}2^{-E_{\mathsf{pw}}}$, but the delay function makes sure that q_{H_p} remains small in practice. Quite remarkably, this term is not multiplied by q_{Enroll}. This is why we claim resistance to multi-target attacks. Note that the hiding property plays no role in this security model.

We can see in the $\mathsf{CUF}_{\mathsf{IdM}less}$ result that a new $q_{\mathsf{Salt}}2^{-E_{\mathsf{pw}}}$ term appears. It models on online dictionary attack on OAgentSalt. There is also a $q_{H_p}q_{\mathsf{Enroll}}2^{-E_{\mathsf{salt}}}$ term.

Proof 2 (sketch) *The full proof is provided in Appendix C.*

Essentially, the first task for the $\mathsf{CUF}_{\mathsf{Agent}}$ *model is to reduce to generating some random seeds for each client instead of going through passwords. This must be done without exposing to multi-target attacks. Then, we can reduce to a single-target attack by guessing which enrollment is on the final target. We select the seed hash without adding it in the random oracle table, which introduces an* $\mathsf{Adv}^{\mathsf{UF}}$ *offset by having the adversary who finds the seed. We use the Forking Lemma to extract* s *and* r_{ID}. *We obtain a game in which either we break the binding property or the Schnorr signature.*

For the $\mathsf{CUF}_{\mathsf{IdM}less}$ *model, we first simulate the proofs* π *by programming the random oracle* H, *we reduce to a game where no* π *is forged besides the ones made by this technique by observing that an extractor would break the commitment scheme, we use the hardness to guess* salt *or* seed *to reduce the game of making* OAgentSalt *return something to an online password guessing game, and after a few similar game hops, we reduce to the game of guessing* pk, *as it is hidden to the adversary.*

On Privacy. At the moment a client sends query to IdM, the purpose of the query is perfectly hidden so that there cannot be any selective censorship by IdM. However, our purpose is to use IdM as an identity provider and to bind a consent to the identity ID so that Judge can assess that the client with identity ID consented to *att* based on $(\mathsf{contract}, att, \mathsf{consent})$. Depending on the application, consent and/or contract may be disclosed or not. If disclosed, IdM can link the client to the *att* action.

More precisely, consent allows to link query to ID but not to the *att* action (if $\mathsf{com}_{\mathsf{ID}}$ in contract is unknown). contract allows to link ID to a set of queries but not to any *att* action (if the π in consent are unknown). contract and consent together allow to link query to ID and also to test if consent relates to a specific *att* action.

Privacy can become more delicate if IdM signs other private attributes related to ID as it can be the case with OIDC as we discuss in Section 10).

In contrast, zkLogin has a different purpose, where even the registered identifier ID of the client to IdM must be hidden and the final consent must be public. This implies that the signature σ must not be part of consent. Instead, zkLogin uses a specific identifier zkaddr for their application, which is derived deterministically from ID and a secret salt and a zero-knowledge proof of knowledge of σ with binding to zkaddr.

9 Applications and Variants

9.1 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 (for instance, the document signing application). The visibility may come from the use case or from an obligation to post the consent on a ledger. The client can easily complain to a Judge for the action to have been done by Agent without any valid consent.

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* by design (as being a smart contract or a TEE) to execute the action only when a valid 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. The distinction with Type II is that the validation of the action happens outside of Agent instead of inside a trusted backend. It is done by an independent service that executes 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, with the techniques from Section 11. It is harder in Type III as the new contract must propagate in the verifying application.

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.

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.[8] 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, Agent signs *att* together with the client ID to notify that it is a signature on behalf of the ID. It may also add a certificate cert showing its entitlement to sign on behalf of ID. This certificate can have the form $\mathsf{cert} = (\mathsf{contract}, \mathsf{consent}_0, \sigma_0)$ where $\mathsf{consent}_0$ is the "signature" of the contract by ID as discussed in Section 11 (i.e., on $att_0 = (\mathsf{sign}, \mathsf{contract})$), and σ_0 is a signature by Agent on contract.

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 [Baldimtsi et al.(2024)] 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. In zkLogin, Client sets a short-term signature key pair and *att* is the declaration of their new public key for this term. The smart contract executes *att* by registering it. Later on, the smart contract accepts transactions from Client which are verified with this key during this term.

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. Client uses IdM and Agent to consent to retrieve the secret. Upon verification of the consent, the trusted backend releases the secret.

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

Some applications need extra protections to avoid Agent to maliciously replay order to get consents to execute *att* several times. This is specially the case when an action cannot be disputed. In the case of transactions in crypto-currencies, having *att* set to the declaration of a short-term public key can be replaid without consequences. However, for bank orders, *att* should include a nonce (e.g. a sequence number, a precise timestamp, or random) and for PDF signatures, *att* should include a date to be included in the signed PDF.

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

9.2 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. In the SSO design, a service can rely on a common and independent identity provider. In the OpenID Connect (OIDC) protocol, a service is a relying party RP. It requests an access token which is signed by the OpenID provider OP to confirm the identity ID of the user. The signature is made on a JSON Web Token JWT. It includes a nonce chosen by RP and ID, which matches our protocol.

Following the OIDC terminology, our Agent is RP, our Client is the user/browser, and our IdM is OP. In OIDC, RP is registered to OP and must sign the authentication request which includes the nonce. This modifies the flow of our protocol as follows: the Client sends a query to Agent, who sets it as the nonce, and signs the authentication request. The Agent returns the request to the Client as a redirect. The Client checks that query was well included in the request and correctly signed by Agent, and redirects the request to IdM. The IdM then authenticates Client. Depending on implementations, IdM may not send back the JWT to Client but rather send it to the Agent directly, in which case the Client only needs to send *att* and *r* to the Agent.

Email provider. Instead of using OIDC to sign the query of the client with ID, we can use the DKIM protocol in email providers. Essentially, the client can send an email to the server including com and the encryption of (ref, att, r). The ciphertext may be added in an ad-hoc email field. The email provider of the client will sign the email fields which are selected by the client and add the signature in the email. Selected fields typically consists of the body (which includes com), the From field (ID is just the email address of the client), and the Date field (for timestamp). Then, the consent includes the selected fields in the email.

It was discussed that DKIM should not make emails undeniable and a way to achieve this was to often rotate $\mathsf{pk}_{\mathsf{IdM}}$ and to publish expired $\mathsf{sk}_{\mathsf{IdM}}$. As far as we know, this has not been implemented but if it does, we should be aware that the publication of $\mathsf{sk}_{\mathsf{IdM}}$ renders consents repudiable.

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 keep CUF security when IdM_1 and Agent are both corrupted but IdM_2 is honest.

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 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, like $(\mathsf{IdM}_1 \wedge \mathsf{IdM}_2) \vee \mathsf{IdM}_3$.

Smartphone IdM. A client may use their smartphone as an IdM which provides identity to a single client.[9] This type of IdM may find perfect 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 (such as a smart phone) to which we can send a nonce and which returns a signature of it can be used *in lieu* of our IdM. The CTAP protocol (used for web authentication using FIDO authenticators such as Yubikey) is easier to adapt to our protocol than OIDC. There again, our Agent is the RP and IdM is the authenticator, defined by the CTAP API. In the CTAP terminology, our query is the challenge and the resp is the returned signature. The Client acts in the middle of the authenticator and RP but contrarily to OIDC, there is no authentication of the RP to the authenticator.

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

Government digital identity. The European digital identity standard EUID and its Swiss variant Swiyu are based on OIDC4VC, which is a variant of OIDC. The idea is that the RP provides a request in the form of a QR code to the user. The user scans the QR code which passes the request to their wallet. The wallet signs a key binding JWT and sends it directly to RP. We can use the wallet as a new IdM. When the QR code includes query, the user's app can verify it before passing it to the wallet.

EUID already allows to pass metadata to the wallet which is shown to user for review during authentication and signed by the wallet. In the currently deployed architecture for qualified electronic signatures, the agent holds signing keys for each user and signs PDF documents upon receiving a signed consent by the eID wallet. This consent signs metadata which have been reviewed by the user such as the title of the document, the number of pages, and a short description. In theory, nothing prevents the agent to sign a document corresponding to the same metadata (for instance, a contract with a few terms slightly modified). What we propose is that the signer uses a trusted app around their eID wallet which would show the document and check that query is correctly passed to the metadata description.

[9] We may wonder why not using a trivial consent protocol where the smartphone directly signs a consent. This approach is actually not secure in the sense of CUF (if the smartphone is corrupted, then consents may be forged).

9.3 Password-less variant

Adding pw for Client to enter in addition to the authentication means to IdM brings security against a corrupted IdM but might decrease usability. Assuming IdM is trusted (for instance when we use an AND composite IdM to make it stronger or when we use a trusted hardware as an IdM), we could remove pw. We could keep salt to protect against self-attacks, but in a static corruption model, salt would be dropped too. In that case, $\mathsf{com}_{\mathsf{ID}}$ is no longer needed in contract and s_{A} is void. We keep the commitment com to *att* to hide *att* from an honest-but-curious IdM but we can replace the NIZK proof by $\pi = r$. The CUF game (with the requirement that ID^* and IdM are not corrupted) reduces directly to the unforgeability of σ and the binding security of com:

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

This is tighter than Theorem 1 but the trust assumption on IdM is now critical.

10 Implementation

We implemented 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 implement the CTAP and OIDC4VC variants similarly, using the WebAuthn assertion format with challenge = com and OpenID4VP with an SD-JWT VC and a key binding JWT over nonce = com, respectively. We implement the DKIM IdM variant using Ed25519 signing as well, specifically, we include the `From, To, Body` and `Date` field in the DKIM signature. We also include an encryption of $(\mathsf{ref}, \mathsf{att}, r)$ within the email using RSA2048-OAEP-SHA256. When H is used as a random oracle, we use SHA256. When H is used as a password hash function, we use Argon2id with standard parameters from RFC9106 or SHA256. We include the benchmarks with SHA256 as a password hash function to highlight the fact that most of the work in Client.Enroll and Client.Launch is spent on Argon2id. The benchmarks have been obtained using the Criterion.rs statistical benchmarking suite on a laptop with Apple M4 processor. We report the benchmark results in Table 1. Each row is one IdM variant; the Client columns are given for both password hash functions.

The source code is available to review and evaluate at https://github.com/consentbenchmark/consent_benchmark/.

Note on real world deployments. OpenPubKey [Heilman et al.(2023)] 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 [Baldimtsi et al.(2024)] and Aptos Keyless Accounts[10] 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 has 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.

Table 1: Benchmark Results (with *att* of 256-bits) on Apple M4 processor.

	Client.Enroll		Client.Launch		IdM	Agent	Verify
	SHA256	Argon2id	SHA256	Argon2id			
OIDC	35.50 μs	71.13 ms	35.64 μs	71.15 ms	10.48 μs	137.40 μs	140.23 μs
CTAP	35.50 μs	71.13 ms	35.64 μs	71.15 ms	8.98 μs	136.74 μs	139.26 μs
OIDC4VC	35.50 μs	71.13 ms	35.64 μs	71.15 ms	11.61 μs	162.03 μs	164.66 μs
DKIM	35.50 μs	71.13 ms	123.23 μs	66.48 ms	14.37 μs	835.75 μs	151.19 μs
NoIdM	83.31 μs	73.04 ms	44.42 μs	72.76 ms	–	151.63 μs	162.49 μs

Privacy extension. In applications requiring to publish the consent (such as sending the consent to a smart contract in order to unlock a service), we must hide some OIDC attributes in the JWT which are put by IdM. This can be done as in zkLogin. [11] However, unlike zkLogin, we do not need unlinkability of the show phase since the consent is typically a one-off credential. Hence, we can reveal the signature and prove a selective opening of the attributes in a given hash. If IdM implements SD-JWT, this privacy extension reduces to opening the hash commitments of the selected attributes, which is extremely cheap. However, SD-JWT is not implemented yet and we rather need a ZK-SNARK. With zkLogin [Baldimtsi et al.(2024)], the proof generation is done in about 30sec. With CRESCENT [Paquin et al.(2024)], the proof generation is split in a preparation phase of 20sec related to credentials and a selective show phase of 30msec. In the application of consent, credentials typically have a one-time usage so we cannot count on amortization. We can reduce the complexity by 33% with Snark-JWT-Verify by disclosing the hash of the token and the signature, as it reduces the size of the circuit. Finally, the Frigo-shelat technique [Frigo and abhi shelat(2024)] based on adhoc optimizations should reduce further to 1.2sec on a mobile device. Although the cost of ZK-SNARKs has dramatically reduced, the complexity is higher compared to our solution and more importantly, the security can be more fragile than imagined [Khovratovich et al.(2025)]. We highlight that our design and security proofs are intact against the vulnerabilities discussed in [Celi et al.(2026)].

Note that the DKIM variant has less privacy problems as the signed email does not reveal much more than ref and com.

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

[11] We note that zkLogin introduced some vulnerabilities in the ZK proof by proving statements on the result of parsed JWT which might not be well-defined, valid or semantically robust for security [Celi et al.(2026)].

Actually, the client can select fields to be signed.

Note on the key rotation of the IdM. A natural question that may arise is what would happen to the enrollment of the Client once the IdM's keys have been rotated, as IdM signatures would no longer verify with the registered key. One solution would be to update the contract, including the $\mathsf{pk}_{\mathsf{IdM}}$, each time a key is rotated. We discuss it in Section 11.

Since we use an OIDC provider in our implementation, we rather append the long-term URI to the public key in contract. Namely, each OpenID JWT contains an `iss` field which specifies the issuing authority. Using this field, it is possible to access the OpenID configuration of the issuer[12]. The configuration includes a `jwk_uri` field which enables retrieving the corresponding public key under which the JWT token is signed. This configuration either includes the rolled over public keys in the long term or only for a duration that is enough to rotate to a new key. If it is the latter, we would further need an application-level cache to store the past public keys per consent or an append-only public ledger, as discussed in Section 11. We would need a permanent public audit in order to check that Agent does not insert rogue keys.

Of course, relying on a long-term URI in the contract and the OIDC token requires trusting the connection to this URI, hence trusting a PKI, however, this assumption is already made in our setting (Subsection 4.2). Thus, we do not introduce any new trust assumption.

11 Long-Term Contract Management

So far, we assumed that either IdM or Agent is honest forever, starting from the initial creation time of a contract. However, a contract may not last for long, be amended (for instance because $\mathsf{pk}_{\mathsf{IdM}}$ rotated), or be revoked. We describe below how to manage contracts and how to use a *ledger* (blockchain) for long-term security.

Creation/amendment/termination of a contract. To avoid having an adversary to create a contract with an ID that they do not control, we require that upon registration of contract, the client must prove ownership of ID under IdM. This is done by giving a consent to the creation of the contract, which is equivalent to signing the contract. This consent is for $att = (\mathsf{sign}, H(\mathsf{contract}))$. Hence, the contract cannot be created and signed by a malicious client or by Agent alone. It can only be created by the right Client or by a malicious IdM. We require to add an expiration date in contract so that it terminates naturally. However, a contract can be revoked before expiration by giving a consent to $att = (\mathsf{revoke}, H(\mathsf{contract}))$. A contract can be amended (for instance, when $\mathsf{pk}_{\mathsf{IdM}}$ rotates) by adding a reference to the current contract in the new one and by giving a consent to change contract into $\mathsf{contract}_{\mathsf{new}}$. This is done by setting a consent with $att = (\mathsf{amend}, H(\mathsf{contract}_{\mathsf{new}}))$ under the old contract and a consent $att = (\mathsf{sign}, H(\mathsf{contract}_{\mathsf{new}}))$ under the new one. The left problem, so far, is the case of an identity theft by the identity manager: a malicious IdM enrolling to Agent to create a new contract with an ID of one of their client.

Endorsement of a contract. When a contract is created, Agent is assumed to take extra care to avoid that a malicious IdM registers a new contract with a stolen ID. If ID is already registered in another existing contract (because an honest user may need several contracts under the same ID), we require that a new contract with the same ID must be either an amendment to the old contract or an endorsed contract by the existing one. It means that the new contract must come with a consent including $att = (\mathsf{endorse}, H(\mathsf{contract}_{\mathsf{new}}))$ for the existing contract and a reference to the endorsing contract must appear in the new contract. That is, contracts with the same ID must define a rooted tree of contracts which are controlled by the same person.

The remaining case is the one of a new contract with a fresh ID (i.e. the root of the tree). We call it an *orphan* contract. We shall not assume that ID corresponds to a particular client. If there is a need for such a binding, additional measures must be used (for instance, legal means). In any case, we must assume that Judge, verifying the validity of a consent, can verify if the root contract is legit. Then, the chain of endorsements/amendments can verify the current contract.

If for some reasons (loss of a password or sudden change in IdM keys) no new consent can be issued on an existing contract, the agent should take extra care when registering a new (orphan) contract with same ID. Such an event must be documented with evidence for Judge.

Long-term security. We want consents to remain verifiable and non-repudiable a long time after a contract was expired or revoked. In the long term, several security incidents can happen on IdM, Agent, or Client. Hence, dynamic corruption after the enrollment of contract must be addressed. For this, we use an append-only ledger to log every event with a timestamp. Events can be the operations on contracts (creation, amendment, revocation, endorsement), the issuance of consents, rotations of $\mathsf{pk}_{\mathsf{IdM}}$ (when observed), and also security incidents (as soon as they are declared and also at the end of the healing process). A Client reporting the leak of their pw implicitly assumes the revocation of their contract. However, having both IdM and Agent corrupt at the same time only implies that contract is suspended until at least one of them heals. This means that Agent is not supposed to issue any consent for Client during this time. We assume read access to the ledger, at least for the involved participants in the logged event and for Judge, so that logs can be audited. The process of verifying a consent becomes more complicated. A consent is valid if it is issued (according to the ledger) between the creation and the termination of contract and if there is no overlap between the corruption time of IdM and Agent between the

[12] This can be simply done by appending the path to the `iss` url, e.g. `iss//.well-known/openid-configuration`.

issuance of contract and consent. The only vulnerable periods which remain are between ID^* or both IdM and Agent are corrupt and the time it is reported on the ledger. However, this limits collateral in the case these corruptions happen, namely by not canceling previous consents.

Enforcing logs on a ledger is useful for *transparency* if the ledger is public. Auditors can check that the reported $\mathsf{pk}_{\mathsf{IdM}}$ rotations are correct. A client can check their logs, spot impersonation, and raise an alert in the case where both IdM and Agent are corrupt, although it does not void a forged consent. A client can also spot when a malicious Agent delays for too long the publication of an event, e.g. the revocation of a contract. It can also place orphan contracts under public scrutiny, for instance to spot when a known person who is not using the system appears to have a contract under their ID.

Our formal security model for the CUF game already includes append commands on the ledger. However, we need a few more changes to implement the above ideas. In the CUF game, we add as a condition that the $(att^*, \mathsf{consent}^*)$ forgery must appear on the ledger. We also change the definition of the challenge phase in the trivial predicate so that it ends at the time t' this consent appears for the first time on the ledger. When we check "after t", we must also check "until t'". This way, we can allow corruption of ID^* and of both IdM and Agent, but for trivial cases.

Generative AI Usage. We used AI (Claude Opus 4.7) to compile the notations given in Section 3 and to perform grammatical checks. The authors reviewed all AI suggestions and implemented a subset of them.

References

[Acar et al.(2013)] Tolga Acar, Mira Belenkiy, and Alptekin Küpçü. 2013. Single password authentication. *Computer Networks* 57, 13 (2013), 2597–2614. doi:10.1016/j.comnet.2013.05.007

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

[Baldimtsi et al.(2024)] Foteini Baldimtsi, Konstantinos Kryptos Chalkias, Yan Ji, Jonas Lindstrøm, Deepak Maram, Ben Riva, Arnab Roy, Mahdi Sedaghat, and Joy Wang. 2024. zkLogin: Privacy-Preserving Blockchain Authentication with Existing Credentials. arXiv:2401.11735 https://arxiv.org/abs/2401.11735

[Baum et al.(2018)] LCarsten Baum, Ivan Damgård, Vadim Lyubashevsky, Sabine Oechsner, and Chris Peikert. 2018. More Efficient Commitments from Structured Lattice Assumptions. In *Security and Cryptography for Networks - SCN 2018*. Springer International Publishing, Amalfi, Italy, 368–385.

[Boldyreva et al.(2003)] Alexandra Boldyreva, Adriana Palacio, and Bogdan Warinschi. 2003. Secure Proxy Signature Schemes for Delegation of Signing Rights. *IACR Cryptol. ePrint Arch.* 2003 (2003), 96. http://eprint.iacr.org/2003/096

[Boldyreva et al.(2012)] Alexandra Boldyreva, Adriana Palacio, and Bogdan Warinschi. 2012. Secure Proxy Signature Schemes for Delegation of Signing Rights. *J. Cryptol.* 25, 1 (2012), 57–115. doi:10.1007/S00145-010-9082-X

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

[Camenisch et al.(2014)] Jan Camenisch, Anja Lehmann, Anna Lysyanskaya, and Gregory Neven. 2014. 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 (Lecture Notes in Computer Science, Vol. 8617)*, Juan A. Garay and Rosario Gennaro (Eds.). Springer, Santa Barbara, CA, 256–275. https://doi.org/10.1007/978-3-662-44381-1_15

[Camenisch et al.(2016)] Jan Camenisch, Anja Lehmann, Gregory Neven, and Kai Samelin. 2016. 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*. Springer-Verlag, Berlin, Heidelberg, 353–371. https://doi.org/10.1007/978-3-319-44618-9_19

[Camenisch and Lysyanskaya(2001)] Jan Camenisch and Anna Lysyanskaya. 2001. An Efficient System for Non-transferable Anonymous Credentials with Optional Anonymity Revocation. In *Advances in Cryptology — EUROCRYPT 2001*, Birgit Pfitzmann (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 93–118.

[Celi et al.(2026)] Sofia Celi, Hamed Haddadi, and Kyle Den Hartog. 2026. Analysis and Vulnerabilities in zkLogin. Cryptology ePrint Archive, Paper 2026/227. https://eprint.iacr.org/2026/227

[Dayanikli and Lehmann(2024)] Dennis Dayanikli and Anja Lehmann. 2024. Password-Based Credentials with Security Against Server Compromise. In *Computer Security*

– *ESORICS 2023*. Springer Nature Switzerland, Cham, 147–167.

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

[DocuSign(2024)] DocuSign. 2024. Security for DocuSign eSignature. https://www.docusign.com/trust/security/esignature

[Faust et al.(2012)] Sebastian Faust, Markulf Kohlweiss, Giorgia Azzurra Marson, and Daniele Venturi. 2012. On the Non-Malleability of the Fiat-Shamir Transform. In *Progress in Cryptology - INDOCRYPT 2012*. Springer Berlin Heidelberg, Berlin, Heidelberg, 60–79.

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

[Frigo and abhi shelat(2024)] Matteo Frigo and abhi shelat. 2024. Anonymous credentials from ECDSA. Cryptology ePrint Archive, Paper 2024/2010. https://eprint.iacr.org/2024/2010

[Fuchsbauer and Pointcheval(2009)] Georg Fuchsbauer and David Pointcheval. 2009. *Anonymous Consecutive Delegation of Signing Rights: Unifying Group and Proxy Signatures*. Springer Berlin Heidelberg, Berlin, Heidelberg, 95–115.

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

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

[He et al.(2005)] Yong-Zhong He, Chuan-Kun Wu, and Deng-Guo Feng. 2005. Server-Aided Digital Signature Protocol Based on Password. In *Proceedings 39th Annual 2005 International Carnahan Conference on Security Technology*. IEEE, Las Palmas, Spain, 89–92.

[Heilman et al.(2023)] Ethan Heilman, Lucie Mugnier, Athanasios Filippidis, Sharon Goldberg, Sebastien Lipman, Yuval Marcus, Mike Milano, Sidhartha Premkumar, Chad Unrein, and John Merfeld. 2023. OpenPubkey: Augmenting OpenID Connect with User held Signing Keys. Cryptology ePrint Archive, Paper 2023/296. https://eprint.iacr.org/2023/296

[Khovratovich et al.(2025)] Dmitry Khovratovich, Ron D. Rothblum, and Lev Soukhanov. 2025. How to Prove False Statements: Practical Attacks on Fiat-Shamir. Advances in Cryptology - CRYPTO '25. https://eprint.iacr.org/2025/118

[Lyubashevsky et al.(2022a)] Vadim Lyubashevsky, Ngoc Khanh Nguyen, and Maxime Plançon. 2022a. Lattice-Based Zero-Knowledge Proofs and Applications: Shorter, Simpler, and More General. In *Advances in Cryptology, CRYPTO 2022*, Yevgeniy Dodis and Thomas Shrimpton (Eds.). Springer, Santa Barbara, CA, USA, 71–101.

[Lyubashevsky et al.(2022b)] Vadim Lyubashevsky, Ngoc Khanh Nguyen, and Maxime Plancon. 2022b. Lattice-Based Zero-Knowledge Proofs and Applications: Shorter, Simpler, and More General. Cryptology ePrint Archive, Paper 2022/284. https://eprint.iacr.org/2022/284

[Paquin et al.(2024)] Christian Paquin, Guru-Vamsi Policharla, and Greg Zaverucha. 2024. Crescent: Stronger Privacy for Existing Credentials. Cryptology ePrint Archive, Paper 2024/2013. https://eprint.iacr.org/2024/2013

[Pointcheval and Stern(1996)] David Pointcheval and Jacques Stern. 1996. Security Proofs for Signature Schemes. In *Advances in Cryptology - EUROCRYPT '96, International Conference on the Theory and Application of Cryptographic Techniques, Saragossa, Spain, May 12-16, 1996, Proceeding (Lecture Notes in Computer Science, Vol. 1070)*, Ueli M. Maurer (Ed.). Springer, 387–398. doi:10.1007/3-540-68339-9_33

[Sangeetha Jose and Rangan(2013)] Preetha Mathew K. Sangeetha Jose and C. Pandu Rangan. 2013. Strongly Secure Password Based Blind Signature for Real World Applications. *Infocommunications Journal* 5, 3 (2013), 22–29.

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

[Schnorr(1991)] Claus Peter Schnorr. 1991. Efficient Signature Generation by Smart Cards. *Journal of Cryptology* 4, 3 (1991), 161–174.

$\mathsf{mUF}(\mathcal{A},t)$

1 : $\mathsf{signed} \leftarrow \{\}$
2 : **for** $i = 1,\ldots,t$
3 : $\quad(\mathsf{sk}_i,\mathsf{pk}_i) \leftarrow \Sigma.\mathsf{Gen}$
4 : $(i,m,\sigma) \leftarrow \mathcal{A}^{\mathsf{OSign}}(\mathsf{pk}_1,\ldots,\mathsf{pk}_t)$
5 : **abort if** $i \notin \{1,\ldots,t\} \vee m \in \mathsf{signed}\{i\}$
6 : **return** $\Sigma.\mathsf{Verify}(\mathsf{pk}_i,m,\sigma)$

$\mathsf{mKR}(\mathcal{B},t)$

1 : **for** $i = 1,\ldots,t$
2 : $\quad(\mathsf{sk}_i,\mathsf{pk}_i) \leftarrow \Sigma.\mathsf{Gen}$
3 : $(i,\mathsf{sk}) \leftarrow \mathcal{A}^{\mathsf{OSign}}(\mathsf{pk}_1,\ldots,\mathsf{pk}_t)$
4 : **abort if** $i \notin \{1,\ldots,t\}$
5 : **return** $\mathsf{sk}_i = \mathsf{sk}$

$\mathsf{OSign}(i,m)$

1 : **abort if** $i \notin \{1,\ldots,t\}$
2 : insert m in $\mathsf{signed}\{i\}$
3 : **return** $\Sigma.\mathsf{Sign}(\mathsf{sk}_i,m)$

Figure 7: Multi-target security of a signature scheme

[Zhang et al.(2020)] Zhenfeng Zhang, Yuchen Wang, and Kang Yang. 2020. Strong Authentication without Tamper-Resistant Hardware and Application to Federated Identities. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020*. The Internet Society.

Supplementary Materials

A Building Blocks

A.1 Signature Scheme

We define the multi-target security model for a signature scheme on Figure 7. In this model, we create t key pairs and the goal is to break security for at least one of them. We focus on unforgeability ($\mathsf{mUF}(t)$) and key recovery ($\mathsf{mKR}(t)$). The regular security is for $t = 1$. For instance, $\mathsf{UF} = \mathsf{mUF}(1)$.

A.2 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 $m = 2$.

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

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

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

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

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

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

Figure 8: NIZK for Π_{MULTEQ}

Hiding and Binding properties. We denote by $\mathbb{G}$ the output range of Com. For adversaries $\mathcal{A}$ and $\mathcal{B}$, we define the advantages $\mathsf{Adv}^{\mathsf{HIDE}}_{\mathcal{A}}$ and $\mathsf{Adv}^{\mathsf{BIND}}_{\mathcal{B}}$ for the hiding and binding games as follows, with the games on Figure 9.

$$\mathsf{Adv}^{\mathsf{HIDE}}_{\mathcal{A}} = \Pr[\mathsf{HIDE}(\mathcal{A},1) \to 1] - \Pr[\mathsf{HIDE}(\mathcal{A},0) \to 1]$$
$$\mathsf{Adv}^{\mathsf{BIND}}_{\mathcal{B}} = \Pr[\mathsf{BIND}(\mathcal{B}) \to 1]$$

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).

$\mathsf{HIDE}(\mathcal{A},b)$	$\mathsf{BIND}(\mathcal{B})$
$(x,\mathsf{state}) \leftarrow \mathcal{A}_1$	$(x_1,r_1,x_2,x_2) \leftarrow \mathcal{B}$
$c_0 \leftarrow \mathsf{Com}(x)$	**return** 0 **if** $x_1 = x_2$
$c_1 \leftarrow\$ \mathbb{G}$	**return** $\mathsf{Com}(x_1;r_1) = \mathsf{Com}(x_2;r_2)$
$z \leftarrow \mathcal{A}_2(\mathsf{state}, c_b)$	
return z	

Figure 9: Hiding and binding game

A.3 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 [Schnorr(1989), Schnorr(1991)] compiled with Fiat-Shamir [Fiat and Shamir(1986)]. 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 8. 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 [Faust et al.(2012)].

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 $(\alpha,\mathsf{ch},\gamma) \leftarrow \mathsf{Sim}(x)$. 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 [Faust et al.(2012), 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. This happens if (x,α) is already in the LH table. In our protocol, α is uniformly distributed in a set of size q^2. So, the advantage is bounded by $\frac{(q_{S_1}+q_{S_2})q_{S_2}}{q^2}$.

We also use the fact that the NIZK is $(q_S, \frac{q_S}{q})$-weakly simulation extractable for $p = q_S$ and $d = 2$, following [Faust et al.(2012), thm. 3]. More precisely, we use the following definition.

Definition 2 *A NIZK for an NP relation $\mathcal{R}$ is (q_S,ν)-weakly simulation extractable if for all PPT adversary $\mathcal{A}$ limited to q_S queries to S_1 or S_2', there exists a PPT extractor $\mathcal{E}$, a polynomial p, and an integer d such that $p_{acc} \leq \nu + \sqrt[d]{p.p_{ext}}$, where p_{acc} is the probability that $\mathcal{A}$ succeeds to forge a new valid proof (x^*,π^*) and p_{Ext} is the probability that the following experiment returns 1 (i.e. $p_{acc} = \Pr[\mathsf{forge}]$ and $p_{ext} = \Pr[\mathsf{forge} \wedge \mathcal{R}(x^*,w^*)]$):*

1 : *pick* ρ
2 : $(x^*,\pi^*) \leftarrow \mathcal{A}^{S_1,S_2'}(;\rho)$
3 : $\mathcal{L} \leftarrow$ *list of input/output queries to* S_2'
4 : $w^* \leftarrow \mathcal{E}(x^*,\pi^*,\rho,\mathsf{LH},\mathcal{L})$
5 : **if** π^* *was output of* $S_2'(x^*)$
6 : $\quad \mathsf{forge} \leftarrow \mathsf{false}$
7 : **else**
8 : $\quad \mathsf{forge} \leftarrow \mathsf{Verify}(\pi^*,x^*)$
9 : **return** $\mathsf{forge} \wedge \mathcal{R}(x^*,w^*)$

For our NIZK, this means

$$p_{acc} \leq \frac{q_S}{q} + \sqrt{q_S \cdot p_{ext}}$$

B Security of Our Protocol with IdM

We formally prove our security result here for the protocol using IdM. First, we show that CUF reduces to a simpler game CUF^* with a single target $\mathsf{contract}^*$. Then, we reduce to two games with static corruption: $\mathsf{CUF}_{\mathsf{IdM}}$ where only IdM is corrupted, and $\mathsf{CUF}_{\mathsf{Agent}}$ where only Agent is corrupted. Then, we prove the security of our specific protocol in the $\mathsf{CUF}_{\mathsf{IdM}}$ and $\mathsf{CUF}_{\mathsf{Agent}}$ games.

For the games we construct during the proof, we denote the *i-th* game with G*i*. In the games, we highlight what has been changed compared to the previous game. We set $\mathsf{G0} = \mathsf{CUF}$. In the random oracle model, the hash function H which is used in the non-interactive proof and in the password hash is modeled by an oracle S_1 doing lazy sampling.

B.1 Reduction to Static Corruption

We first reduce CUF to a slightly simpler game in a generic manner.

G1 (**Figure 10**). First of all, we remove from the game the actions with Ledger and the OPost oracle because they are not used. The storage of salt in $\mathsf{mem}_{\mathsf{Agent}}$ is not used so removed too.

We replace all H queries (for password hash or ZK) by the lazy sampling oracle S_1. Then, we add an oracle

which will be useful: $\mathsf{OAuthenticate}(\mathsf{pw},\mathsf{salt},att,\rho)$ first runs $\mathsf{Client.Launch}(\mathsf{pw},\mathsf{salt},att;\rho)$ with the adversarially provided randomness ρ to get query and state, then runs $\mathsf{IdM.Query}(\mathsf{ID}^*,\mathsf{sk}_{\mathsf{IdM}},\mathsf{query})$ to get resp, and insert att in consented. Using this oracle, an adversary can perfectly simulate OClientLaunch and OClientCommit when pw is known. The same holds for OAgentOrder when s_{A} is known.

Since the trivial condition implies no $\mathsf{OCorrupt}(\mathsf{ID}^*)$ after $\mathsf{contract}^*$ is set, we can change the condition "$\mathsf{ID}^* \notin$ Corrupted" by the less restrictive one "$\mathsf{contract}^*$ exists". We just need to replace $\mathcal{A}$ by an adversary who simulates $\mathcal{A}$ but filters queries which would have aborted.

A simulation of $\mathcal{A}$ can easily figure out which OClientEnroll query defines the challenge as soon as it is queried (because the simulator can predict which queries will abort the game and which query has flag set; there can be only one non-aborting query with flag set). The simulation can also figure out the corruption status of every participant.

We define a new adversary $\mathcal{A}_1$ which first simulates $\mathcal{A}$ and all client and agent oracles using the above method until the OClientEnroll query defining the challenge is made by $\mathcal{A}$. This defines ref^*. Simulating OClientEnroll implies knowing pw and s_{A}, so being able to simulate OClientLaunch, OClientCommit, and OAgentOrder. The adversary $\mathcal{A}_1$ makes the challenge-defining OClientEnroll call then continues the simulation by simulating only the queries with $\mathsf{ref} \neq \mathsf{ref}^*$ and *sid* coming from a simulation. Corrupting/healing Agent before ref^* is set is pointless. By postponing a residual corruption to after ref^* is set, there is no such query any more. We obtain a game which only plays with IdM and corrupt/healing IdM, then defines the target contract $\mathsf{contract}^*$ in a first phase, then continues with all oracles. It uses a single OClientEnroll oracle call, the memory of ID^* only contains pw^*, and the memory of Agent only contains s^*_{A}.

The trivial verification allows to reduce to a game with no $\mathsf{OCorrupt}(\mathsf{ID}^*)$ any more, and either no IdM corruption after the challenge is set, or no $\mathsf{OCorrupt}(\mathsf{Agent})$ at all.

We have:

$$\mathsf{Adv}^{\mathsf{G0}}_{\mathcal{A}} = \mathsf{Adv}^{\mathsf{G1}}_{\mathcal{A}_1}$$

$\mathsf{G1}_P$ **(Figure 11).** The G1 game includes a trivial predicate which considers two cases: the malicious IdM case and the malicious Agent case. We can define two predicates $\mathsf{trivial}_P$ which would only consider the malicious P case, and obtain two games $\mathsf{G1}_{\mathsf{IdM}}$ and $\mathsf{G1}_{\mathsf{Agent}}$. The $\mathsf{trivial}_{\mathsf{IdM}}$ predicate says if there is any $\mathsf{OCorrupt}(\mathsf{Agent})$ call, which is not allowed in $\mathsf{G1}_{\mathsf{IdM}}$. The $\mathsf{trivial}_{\mathsf{Agent}}$ predicate says if there is any $\mathsf{OCorrupt}(\mathsf{IdM})$ call after the OClientEnroll call defining $\mathsf{contract}^*$, which is not allowed in $\mathsf{G1}_{\mathsf{Agent}}$.

In $\mathsf{G1}_{\mathsf{IdM}}$, we can assume without loss of generality that IdM is corrupted before the OClientEnroll call and that it never heals. So, OIdMQuery and OAuthenticate are no longer used after the challenge is set. Similarly, we can assume in $\mathsf{G1}_{\mathsf{Agent}}$ that Agent is corrupted right after the challenge is set (meaning that s^*_{A} is returned to the adversary) and never heals. So, OAgentOrder is never used. We can thus simplify the corruption management as specified in the new game.

Obviously, an adversary playing the G1 game can also play the two games $\mathsf{G1}_P$, and we are ensured that any winning case in G1 translates in at least one of the two games. We have

$$\mathsf{Adv}^{\mathsf{G1}}_{\mathcal{A}_1} \leq \mathsf{Adv}^{\mathsf{G1}_{\mathsf{IdM}}}_{\mathcal{A}_1} + \mathsf{Adv}^{\mathsf{G1}_{\mathsf{Agent}}}_{\mathcal{A}_1}$$

B.2 Simulation of π

We use the technique in [Faust et al.(2012)] to simulate the proof π in consent.

$\mathsf{G2}_P$ **(Figure 12).** In OAgentOrder, OClientLaunch, and OClientCommit, we expand the Agent algorithm following our protocol. However, we replace $\Pi_{\mathsf{MULTEQ}}.\mathsf{PoK}$, by $\Pi_{\mathsf{MULTEQ}}.\mathsf{Sim}$ using a new oracle S_2 which programs the random oracle. The technique works as follows: The oracle S_2 takes as input a statement x and a witness w, checks the validity of the relation $R(x,w)$, and returns a simulated proof while programming LH. We use the fact that Π_{MULTEQ} is perfect zero-knowledge, as discussed in Subsection A.3. 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:

$$\mathsf{Adv}^{\mathsf{G1}_P}_{\mathcal{A}_1} \leq \mathsf{Adv}^{\mathsf{G2}_P}_{\mathcal{A}_1} + \frac{(q_{\mathsf{S}_1} + q_{\mathsf{S}_2}) \cdot q_{\mathsf{S}_2}}{q^2}$$

We have $q_{\mathsf{S}_1} \leq q_{\mathsf{H}}$ and $q_{\mathsf{S}_2} \leq q_{\mathsf{Launch}}$. Indeed, queries to OClientLaunch and OClientCommit with the same *sid* cannot both use S_2.

$\mathsf{G2}'_P$ **(Figure 13).** We can see in $\mathsf{G2}_P$ that SimAgent is only called by honest oracles who produce a valid witness. Hence, we can change S_2 by S'_2 which needs no witness w as input. Consequently, r_{ID} is no longer used.

We can see that the SimAgent check of $\mathsf{com} = \mathsf{Com}(h_{\mathsf{ID}},att,r)$ is now always satisfied in OClientLaunch and OClientCommit. This is due to that when ID^* and Agent are honest, they must have a coherent view based on the same ref. Hence, we can replace SimAgent by $\mathsf{SimAgent}'$ which skips this verification in the client oracles. We keep SimAgent in OAgentOrder.

We have:

$$\mathsf{Adv}^{\mathsf{G2}'_{\mathsf{IdM}}}_{\mathcal{A}_1} = \mathsf{Adv}^{\mathsf{G3}_{\mathsf{IdM}}}_{\mathcal{A}_1}$$

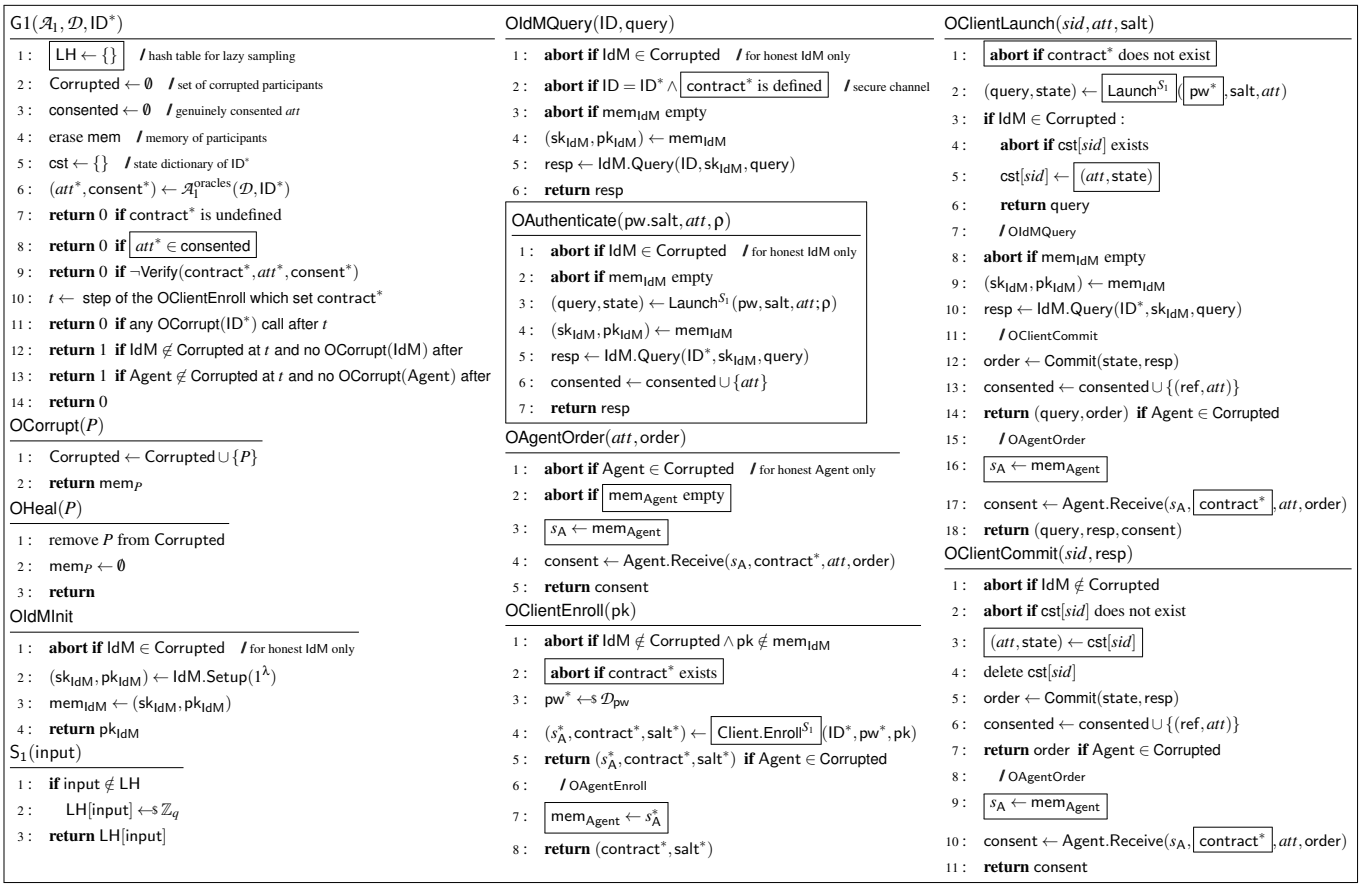

Figure 10: G1 Game.

B.3 The Corrupted IdM Case

$\mathsf{G3}_{\mathsf{IdM}}$ **(Figure 14).** We can simplify the game with $P = \mathsf{IdM}$. We realize that r created in OClientLauch cannot be returned to the adversary and is not used anywhere. Similarly, s_{A} is not revealed and nowhere used. So, we can drop them.

Given that IdM is corrupted, the signature in the consent is not important in this game. Hence, we can replace $\mathcal{A}_1$ by an adversary $\mathcal{A}_3$ which simulates $\mathcal{A}_1$ as well as the oracles OCorrupt, OIdMInit, OIdMQuery, and OAuthenticate. We also drop the restriction on the input pk in OClientEnroll. The adversary can only succeed more in the new game.

We have:

$$\mathsf{Adv}^{\mathsf{G2}'_P}_{\mathcal{A}_1} \leq \mathsf{Adv}^{\mathsf{G3}_P}_{\mathcal{A}_3}$$

$\mathsf{G4}_{\mathsf{IdM}}$ **(Figure 15).** Note that r is not used in OClientLaunch after Com anymore. We use the fact that Com is hiding and replace com with a random group element. We further remove the generation of r. Note that OClientLaunch does not use pw^* anymore.

Similarly, r_{ID} is never used after computing $\mathsf{com}_{\mathsf{ID}}$. Hence, we can also replace $\mathsf{com}_{\mathsf{ID}}$ by a random group element in OClientEnroll.

For every query to OClientEnroll and OClientLaunch, this defines an adversary $\mathcal{B}'$ playing the HIDE game.

We further change OAgentOrder so that it no longer uses h_{ID} by invoking a new oracle $\mathsf{OTest}(w)$ which says whether w is equal to $\mathsf{Com}(h_{\mathsf{ID}}, 0, 0)$ or not. Then, we use the homomorphic property of the commitment to update OAgentOrder.

$$\mathsf{Adv}^{\mathsf{G3}_{\mathsf{IdM}}}_{\mathcal{A}_3} \leq \mathsf{Adv}^{\mathsf{G4}_{\mathsf{IdM}}}_{\mathcal{A}_3} + (q_{\mathsf{Enroll}} + q_{\mathsf{Launch}}) \cdot \mathsf{Adv}^{\mathsf{HIDE}}_{\mathcal{B}'}$$

$\mathsf{G5}_{\mathsf{IdM}}$ **(Figure 16).** In $\mathsf{G4}_{\mathsf{IdM}}$, pw^* is sampled once in OClientEnroll then only used in OTest. We now replace OTest by an oracle always returning 0. The failure event is when an adversary succeeded to guess the value of $\mathsf{Com}(H_p(\mathsf{pw}^*, \mathsf{salt}^*), 0, 0)$ by making q_{Order} guesses, when salt^* is known.

We let E_{Com} be the min-entropy of $\mathsf{Com}(H_p(\mathsf{pw}, \mathsf{salt}), 0; 0)$ with salt fixed and $\mathsf{pw} \leftarrow\!\!\$\ \mathcal{D}_{\mathsf{pw}}$. Clearly, the failure event has

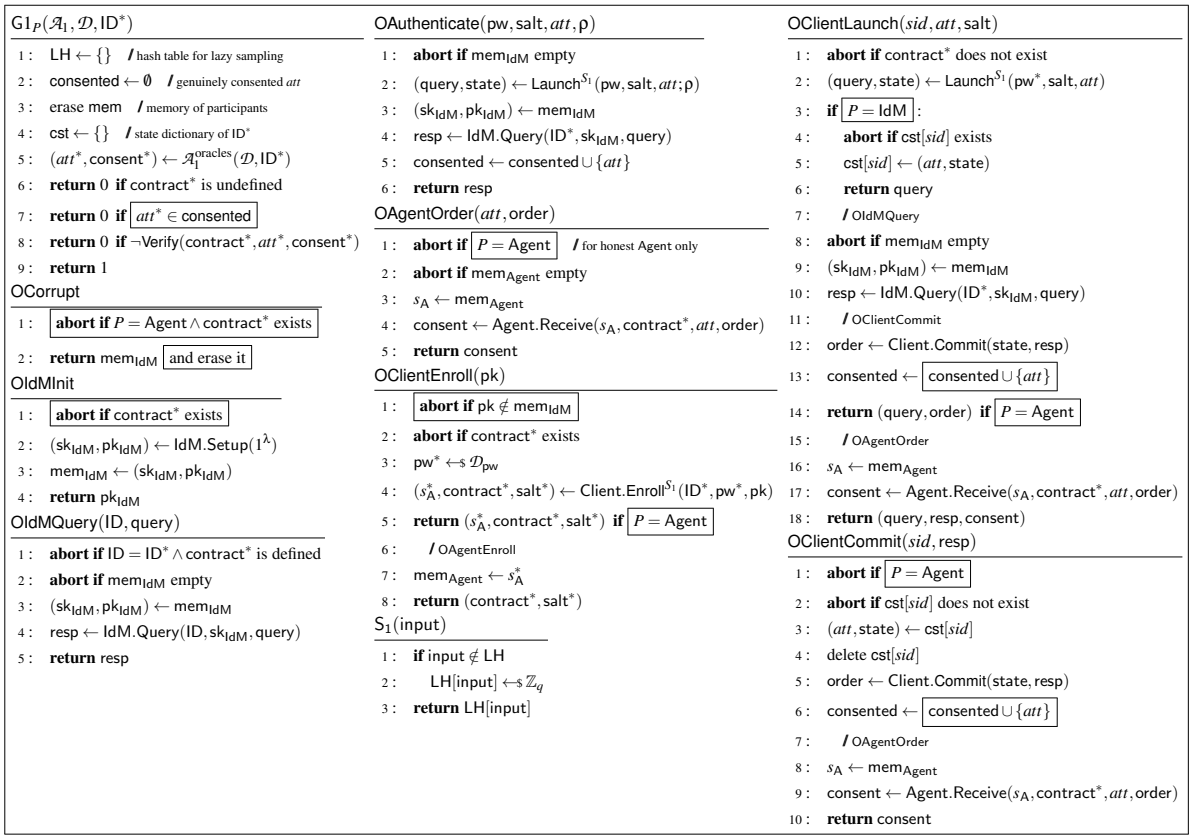

Figure 11: G1$_P$ Game.

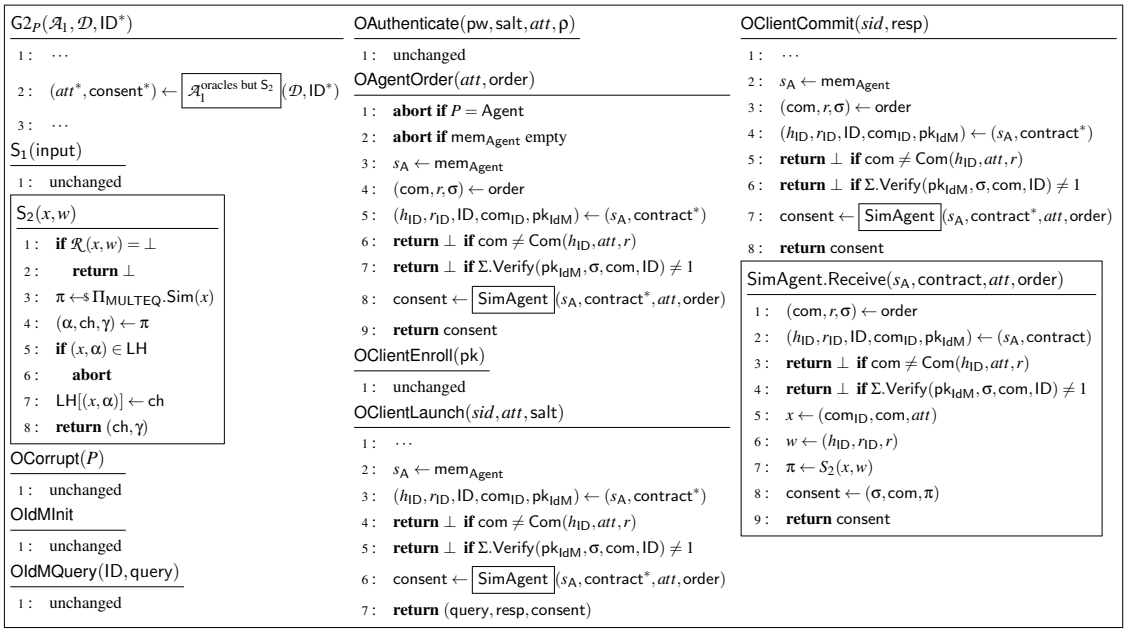

Figure 12: G2$_P$ Game.

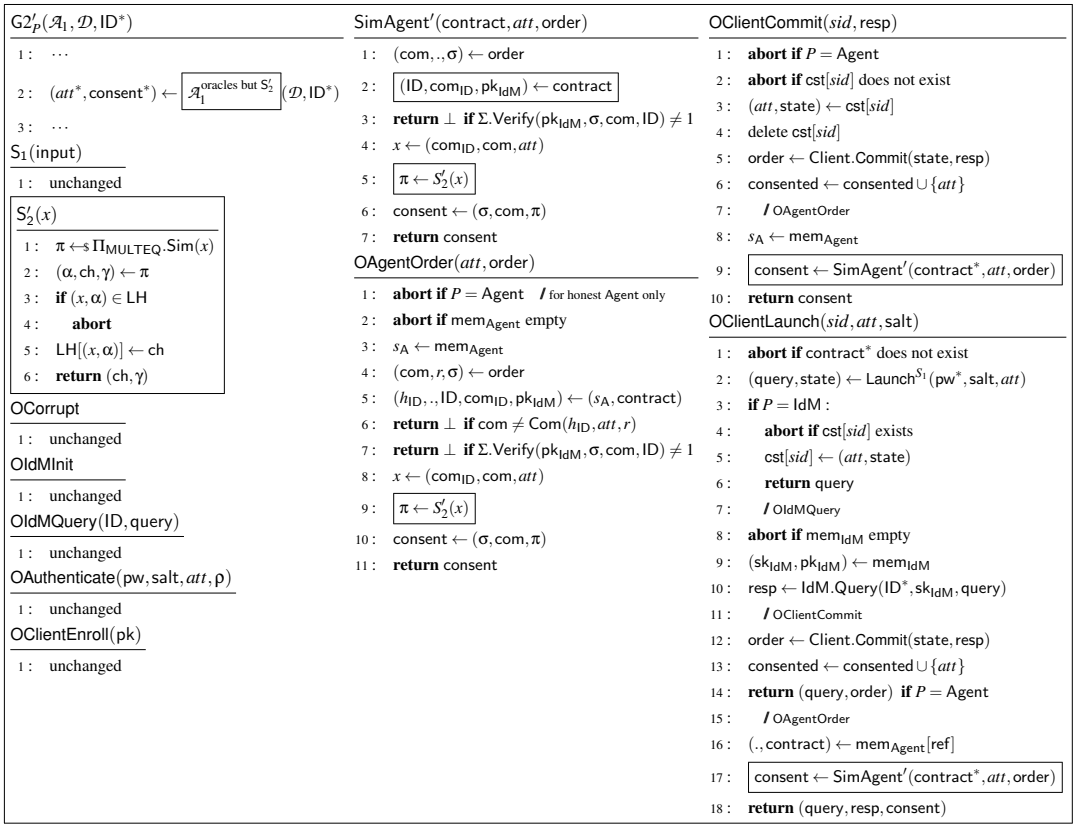

Figure 13: $\mathsf{G2}'_P$ Game.

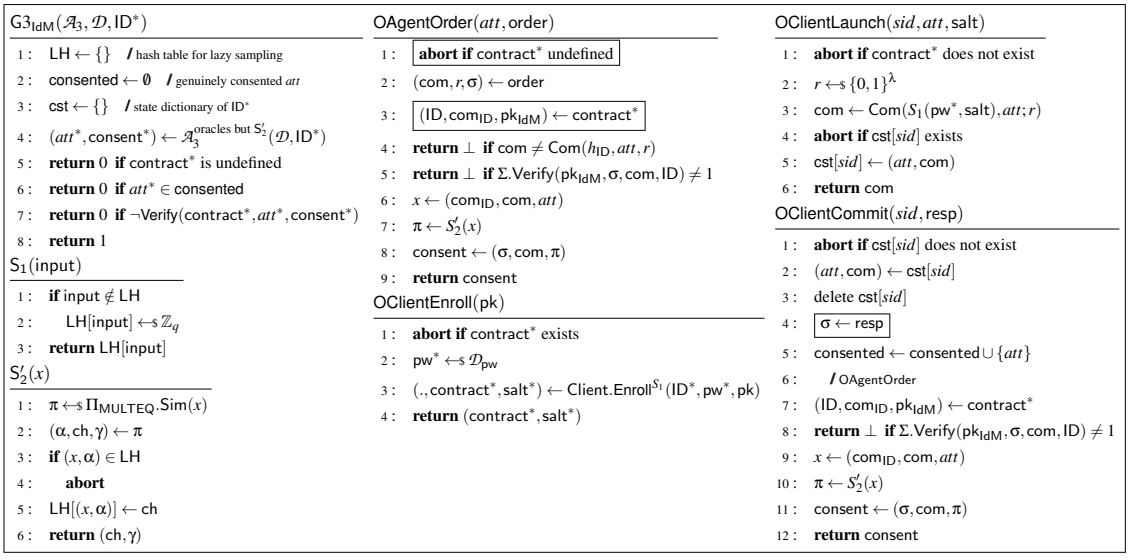

Figure 14: $\mathsf{G3}_{\mathsf{IdM}}$ Game.

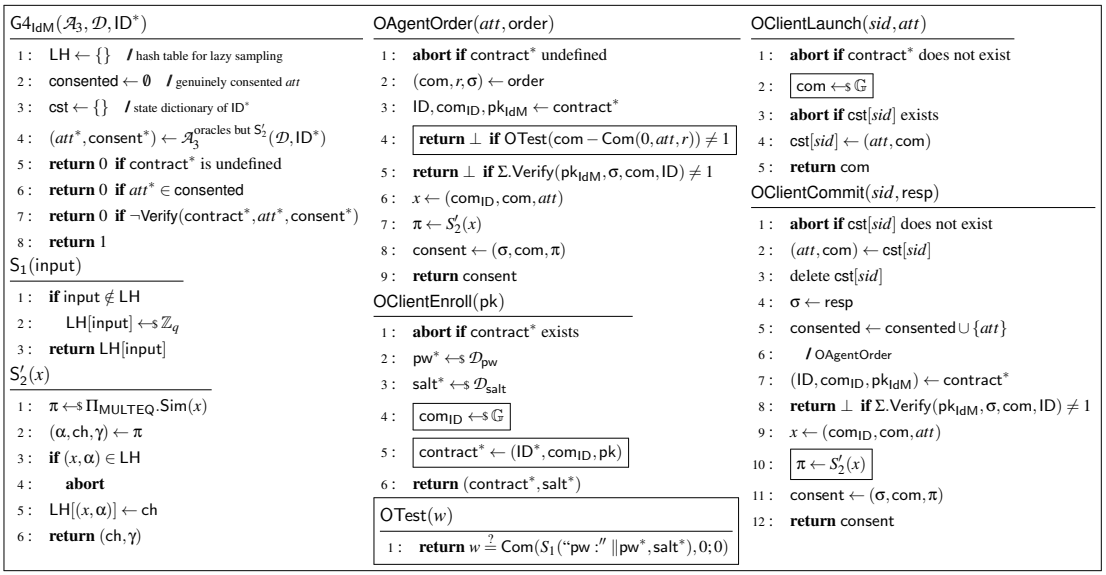

Figure 15: $\mathsf{G4_{IdM}}$ Game.

success probability bounded by $q_{\mathsf{Order}} \cdot 2^{-E_{\mathsf{Com}}}$.

$$\mathsf{Adv}^{\mathsf{G4_{IdM}}}_{\mathcal{A}_3} \leq \mathsf{Adv}^{\mathsf{G5_{IdM}}}_{\mathcal{A}_3} + \frac{q_{\mathsf{Order}}}{2^{E_{\mathsf{Com}}}}$$

As OTest always returns 0, OAgentOrder always aborts. These two oracles can then be ignored in what follows.

To estimate E_{Com}, we proceed as follows. Assuming that the set of all possible pw is enumerable for the adversary, we can consider an adversary computing the table of all $\mathsf{pw} \mapsto \mathsf{Com}(H_p(\mathsf{pw}, \mathsf{salt}^*), 0; 0)$. If this process finds no collision on H_p and no collision on $h \mapsto \mathsf{Com}(h, 0; 0)$, then the table is a one-to-one mapping so $E_{\mathsf{Com}} = E_{\mathsf{pw}}$. Otherwise, the adversary finds a collision on H_p or breaks the binding property of the commitment. Therefore, we can bound

$$\frac{q_{\mathsf{Order}}}{2^{E_{\mathsf{Com}}}} \leq \frac{q_{\mathsf{Order}}}{2^{E_{\mathsf{pw}}}} + \frac{q_H^2}{2q} + \mathsf{Adv}^{\mathsf{BIND}}_{\mathcal{A}'}$$

$\mathsf{G6_{IdM}}$ **(Figure 17).** We introduce a new oracle OSim which forges proofs for any adversarially provided $x = (\mathsf{com_{ID}}, \mathsf{com}, att)$ but inserts att in the set consented. Clearly, we can replace $\mathcal{A}_3$ by an adversary $\mathcal{A}_6$ which simulates $\mathcal{A}_3$, OClientCommit, and OClientLaunch. 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. For clarity, we expanded the Verify algorithm in the game, except the verification of the signature which brings nothing to the security. Finally, the generation of $\mathsf{com_{ID}}$ in OClientEnroll can be done at the beginning of the game, the generation of pw^* is useless, and the oracle can be dropped. Therefore,

$$\mathsf{Adv}^{\mathsf{G5_{IdM}}}_{\mathcal{A}_3} \leq \mathsf{Adv}^{\mathsf{G6_{IdM}}}_{\mathcal{A}_6}$$

$\mathsf{G7_{IdM}}$ **(Figure 18).** We use the weak simulation extractability of our NIZK, as discussed in Subsection A.3, to obtain the extraction of a witness for the newly generated proof, thanks to an extractor $\mathcal{E}$. Therefore,

$$\mathsf{Adv}^{\mathsf{G6_{IdM}}}_{\mathcal{A}_6} \leq \frac{q_{\mathsf{S}_1}}{q} + \sqrt{q_{\mathsf{S}_1} \cdot \mathsf{Adv}^{\mathsf{G7_{IdM}}}_{\mathcal{A}_6, \mathcal{E}}}$$

We have $q_{\mathsf{S}_1} \leq q_{\mathsf{H}}$.

$\mathsf{G8_{IdM}}$ **(Figure 19).** We now pick some random h^* and r^* and we add a new success condition: that $h_{\mathsf{ID}} \neq h^*$. Clearly, the change case corresponds to guessing h^* with probability $\frac{1}{q}$. With the difference lemma, we obtain

$$\mathsf{Adv}^{\mathsf{G7_{IdM}}}_{\mathcal{A}_6, \mathcal{E}} \leq \mathsf{Adv}^{\mathsf{G8_{IdM}}}_{\mathcal{A}_6, \mathcal{E}} + \frac{1}{q}$$

$\mathsf{G9_{IdM}}$ **(Figure 20).** We change the selection of $\mathsf{com_{ID}}$ by setting $\mathsf{com_{ID}} = \mathsf{Com}(h^*, 0; r^*)$. Due to the hiding property, the gap defines an adversary $\mathcal{C}'$ playing the HIDE game. Then, by including nearly everything inside the adversary, we obtain a new adversary $\mathcal{A}_9$ who can open any random commitment $\mathsf{com_{ID}}$. Due to the hiding property, we obtain

$$\mathsf{Adv}^{\mathsf{G8_{IdM}}}_{\mathcal{A}_6, \mathcal{E}} \leq \mathsf{Adv}^{\mathsf{G9_{IdM}}}_{\mathcal{A}_9} + \mathsf{Adv}^{\mathsf{HIDE}}_{\mathcal{C}'}$$

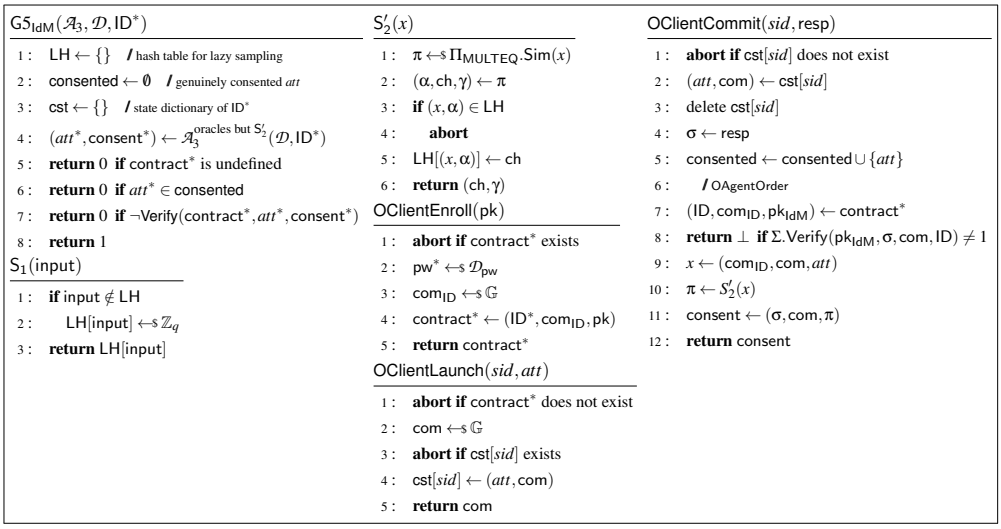

Figure 16: $\mathsf{G5}_{\mathsf{IdM}}$ Game.

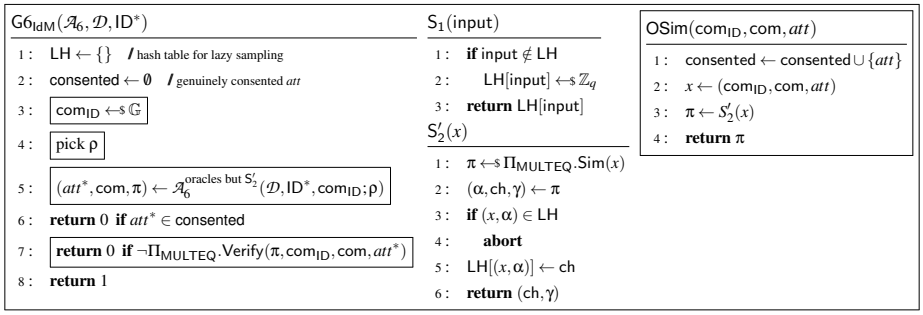

Figure 17: $\mathsf{G6}_{\mathsf{IdM}}$ Game.

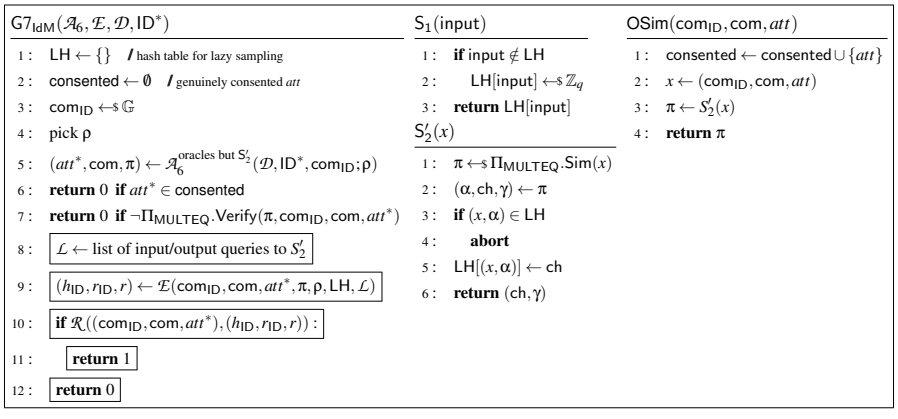

Figure 18: $\mathsf{G7}_{\mathsf{IdM}}$ Game.

$\mathsf{G8}_{\mathsf{IdM}}(\mathcal{A}_6, \mathcal{E}, \mathcal{D}, \mathsf{ID}^*)$

1 : $\mathsf{LH} \leftarrow \{\}$ / hash table for lazy sampling
2 : $\mathsf{consented} \leftarrow \emptyset$ / genuinely consented *att*
3 : $\boxed{h^*, r^* \leftarrow\!\$\, \mathbb{Z}_q}$
4 : $\mathsf{com}_{\mathsf{ID}} \leftarrow\!\$\, \mathbb{G}$
5 : pick ρ
6 : $(att^*, \mathsf{com}, \pi) \leftarrow \mathcal{A}_6^{\text{oracles but } \mathsf{S}_2'}(\mathcal{D}, \mathsf{ID}^*, \mathsf{com}_{\mathsf{ID}}; \rho)$
7 : **return** 0 **if** $att^* \in \mathsf{consented}$
8 : **return** 0 **if** $\neg \Pi_{\mathsf{MULTEQ}}.\mathsf{Verify}(\pi, \mathsf{com}_{\mathsf{ID}}, \mathsf{com}, att^*)$
9 : $\mathcal{L} \leftarrow$ list of input/output queries to S_2'
10 : $(h_{\mathsf{ID}}, r_{\mathsf{ID}}, r) \leftarrow \mathcal{E}(\mathsf{com}_{\mathsf{ID}}, \mathsf{com}, att^*, \pi, \rho, \mathsf{LH}, \mathcal{L})$
11 : $\boxed{\textbf{return } 0 \textbf{ if } h^* = h_{\mathsf{ID}}}$
12 : **if** $\mathcal{R}((\mathsf{com}_{\mathsf{ID}}, \mathsf{com}, att^*), (h_{\mathsf{ID}}, r_{\mathsf{ID}}, r))$:
13 : **return** 1
14 : **return** 0

$\mathsf{S}_1(\mathsf{input})$

1 : **if** input $\notin \mathsf{LH}$
2 : $\mathsf{LH}[\mathsf{input}] \leftarrow\!\$\, \mathbb{Z}_q$
3 : **return** $\mathsf{LH}[\mathsf{input}]$

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

1 : $\pi \leftarrow\!\$\, \Pi_{\mathsf{MULTEQ}}.\mathsf{Sim}(x)$
2 : $(\alpha, \mathsf{ch}, \gamma) \leftarrow \pi$
3 : **if** $(x, \alpha) \in \mathsf{LH}$
4 : **abort**
5 : $\mathsf{LH}[(x, \alpha)] \leftarrow \mathsf{ch}$
6 : **return** (ch, γ)

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

1 : $\mathsf{consented} \leftarrow \mathsf{consented} \cup \{att\}$
2 : $x \leftarrow (\mathsf{com}_{\mathsf{ID}}, \mathsf{com}, att)$
3 : $\pi \leftarrow S_2'(x)$
4 : **return** π

Figure 19: $\mathsf{G8}_{\mathsf{IdM}}$ Game.

$\mathsf{G9}_{\mathsf{IdM}}(\mathcal{A}_8, \mathcal{D}, \mathsf{ID}^*)$

1 : $h^*, r^* \leftarrow\!\$\, \mathbb{Z}_q$
2 : $\mathsf{com}_{\mathsf{ID}} \leftarrow \mathsf{Com}(h^*, 0; r^*)$
3 : $(h_{\mathsf{ID}}, r_{\mathsf{ID}}) \leftarrow \mathcal{A}_8(\mathsf{com}_{\mathsf{ID}})$
4 : **return** 0 **if** $h^* = h_{\mathsf{ID}}$
5 : **return** 0 **if** $\mathsf{Com}(h_{\mathsf{ID}}, 0, r_{\mathsf{ID}}) \neq \mathsf{com}_{\mathsf{ID}}$
6 : **return** 1

Figure 20: $\mathsf{G9}_{\mathsf{IdM}}$ Game.

Wrapping up. The game $\mathsf{G9}_{\mathsf{IdM}}$ can be simulated by an adversary $\mathcal{B}$ playing the binding security game for the commitment. Hence, $\mathsf{Adv}^{\mathsf{G9}_{\mathsf{IdM}}}_{\mathcal{A}_9}$ is the advantage in the binding game.

$$\mathsf{Adv}^{\mathsf{G9}_{\mathsf{IdM}}}_{\mathcal{B}} = \mathsf{Adv}^{\mathsf{BIND}}_{\mathcal{B}}$$

B.4 The Corrupted Agent Case

$\mathsf{G3}_{\mathsf{Agent}}$ **(Figure 21).** We simplify the $\mathsf{G2}'_{\mathsf{Agent}}$ game as we know that $P = \mathsf{Agent}$. We note that $\mathsf{OAgentOrder}$ and S_2' disappear. We have:

$$\mathsf{Adv}^{\mathsf{G2}'_{\mathsf{Agent}}}_{\mathcal{A}_1} = \mathsf{Adv}^{\mathsf{G3}_{\mathsf{Agent}}}_{\mathcal{A}_3}$$

$\mathsf{G4}_{\mathsf{Agent}}$ **(Figure 22).** We simplify a bit the $\mathsf{OCorrupt}/\mathsf{OIdMInit}$ key disclosure/creation. We also mark all $(\mathsf{ID}, \mathsf{query})$ pairs which are signed by the IdM algorithm in a set signed for future use.

We expand Enroll in $\mathsf{OClientEnroll}$. Also for future use, we add as a condition that $H_p(\mathsf{pw}^*, \mathsf{salt}^*)$ was not queried to the random oracle at the time it has been randomly selected. Clearly, the failure case happens with a probability which depends on the min-entropy of pw^* and salt^*.

$$\mathsf{Adv}^{\mathsf{G3}_{\mathsf{Agent}}}_{\mathcal{A}_3} \leq \frac{q_{\mathsf{S}_1}}{2^{E_{\mathsf{pw}}+E_{\mathsf{salt}}}} + \mathsf{Adv}^{\mathsf{G4}_{\mathsf{Agent}}}_{\mathcal{A}_3}$$

$\mathsf{G5}_{\mathsf{Agent}}$ **(Figure 23).** We add as a success condition that $(\mathsf{ID}^*, \mathsf{com}^*)$ belongs to signed and use the difference lemma. The failure event is that the game succeeds to forge a signature. Using a hybrid argument, we can assume that we can guess which $\mathsf{OIdMInit}$ query is making the key which verifies the forgery. This way, we define an adversary $\mathcal{C}$ playing the UF game against the signature scheme Σ. We have:

$$\mathsf{Adv}^{\mathsf{G4}_{\mathsf{Agent}}}_{\mathcal{A}_3} \leq q_{\mathsf{Init}} \cdot \mathsf{Adv}^{\mathsf{UF}}_{\mathcal{C}} + \mathsf{Adv}^{\mathsf{G5}_{\mathsf{Agent}}}_{\mathcal{A}_3}$$

$\mathsf{G6}_{\mathsf{Agent}}$ **(Figure 24).** We simplify $\mathsf{OClientEnroll}$ so that sampling pw^* and salt^* is done by the adversary and that we no longer need the public key of IdM. We let the adversary handle the key generation and signatures. We obtain a new adversary $\mathcal{A}_6$. This eliminates $\mathsf{OIdMInit}$ and $\mathsf{OIdMCorrupt}$ and further simplifies $\mathsf{OIdMQuery}$, $\mathsf{OAuthenticate}$, and $\mathsf{OClientLaunch}$. As $\mathsf{OClientLaunch}$ can be simulated with $\mathsf{OAuthenticate}$, we can remove it. Since adding in Signed some $(\mathsf{ID}, \mathsf{query})$ pairs with $\mathsf{ID} \neq \mathsf{ID}^*$ is useless, we restrict $\mathsf{OIdmQuery}$ to $\mathsf{ID} = \mathsf{ID}^*$ and simplify it. The impact is that all oracles can now be simulated by an extractor who simulates the adversary.

We have

$$\mathsf{Adv}^{\mathsf{G5}_{\mathsf{Agent}}}_{\mathcal{A}_3} = \mathsf{Adv}^{\mathsf{G6}_{\mathsf{Agent}}}_{\mathcal{A}_6}$$

$\mathsf{G7}_{\mathsf{Agent}}$ **(Figure 25).** We use the weak simulation extractability of our NIZK, as discussed in Subsection A.3, to obtain the extraction of a witness for the proof π^* which is returned by

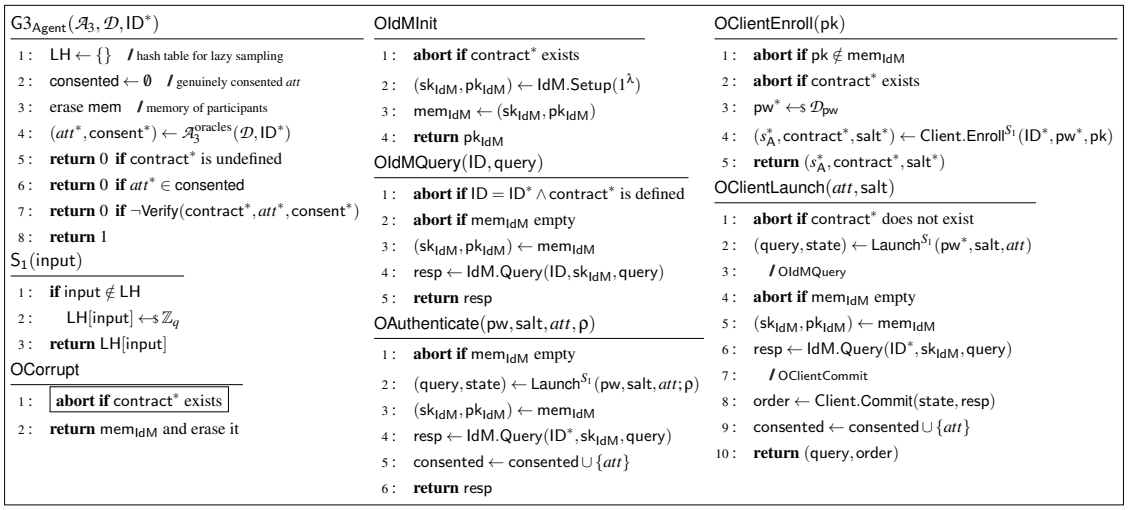

Figure 21: $\mathsf{G3}_{\mathsf{Agent}}$ Game.

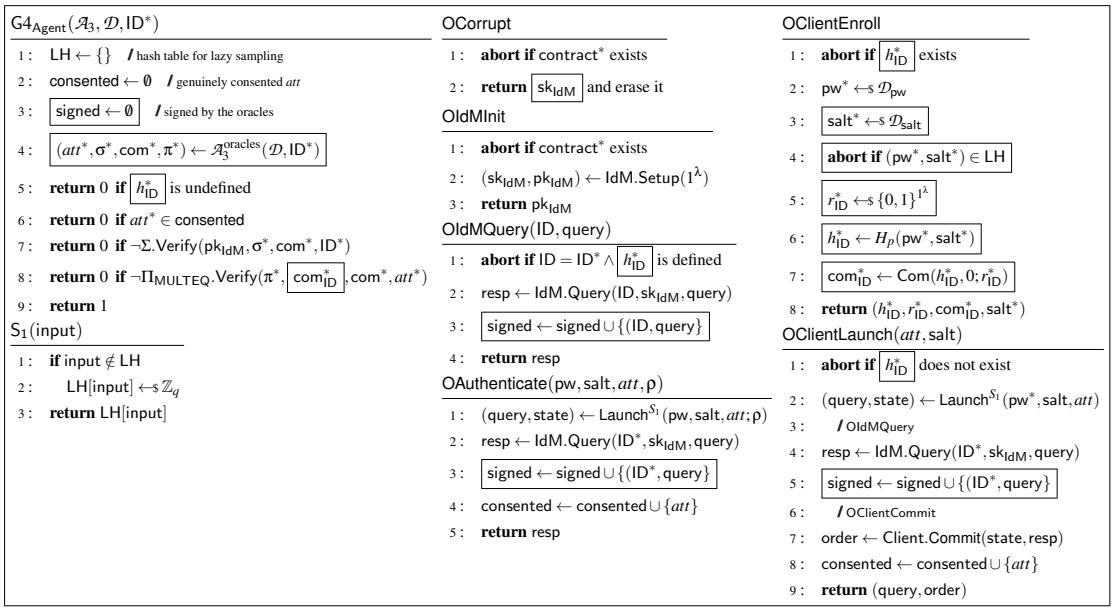

Figure 22: $\mathsf{G4}_{\mathsf{Agent}}$ Game.

$\mathcal{A}_6$, thanks to an extractor $\mathcal{E}$. Therefore,

$$\mathsf{Adv}^{\mathsf{G6}_{\mathsf{Agent}}}_{\mathcal{A}_6} \leq \frac{q_{\mathsf{S}_1}}{q} + \sqrt{q_{\mathsf{S}_1} \cdot \mathsf{Adv}^{\mathsf{G7}_{\mathsf{Agent}}}_{\mathcal{A}_6,\mathcal{E}}}$$

We have $q_{\mathsf{S}_1} \leq q_{\mathsf{H}}$.

$\mathsf{G8}_{\mathsf{Agent}}$ **(Figure 26).** The OAuthenticate oracle is signing and consenting at the same time. Clearly, if the final com* corresponds to a query in one of them, the winning conditions imply that we broke the binding property of the commitment. Hence, we add an additional condition that the signature comes from OIdmQuery, which is equivalent to removing the oracle OAuthenticate. The adversary can simulate it.

Similarly, the $\mathcal{R}$ relation implies that $h_{\mathsf{ID}} = h^*_{\mathsf{ID}}$ or that we break the binding property on $\mathsf{com}^*_{\mathsf{ID}}$. Hence, we can add the condition $h_{\mathsf{ID}} = h^*_{\mathsf{ID}}$ and simplify further.

We can now store query alone in signed as ID^* is not useful any more.

The consented set is no longer useful and can be removed.

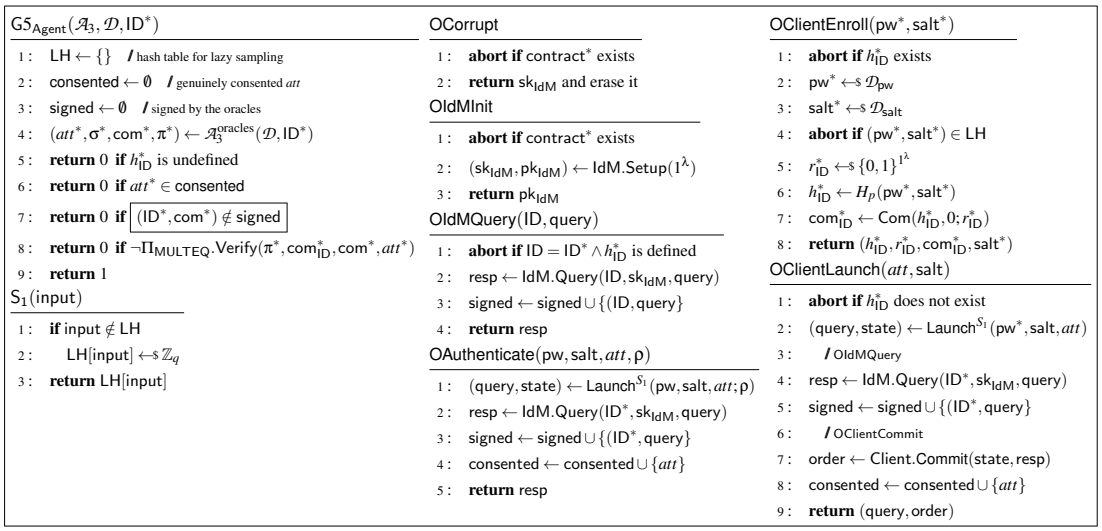

Figure 23: $\mathsf{G5}_{\mathsf{Agent}}$ Game.

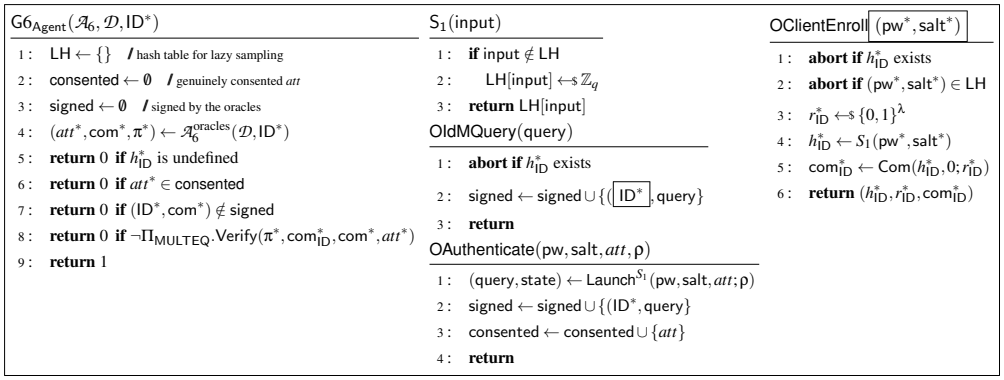

Figure 24: $\mathsf{G6}_{\mathsf{Agent}}$ Game.

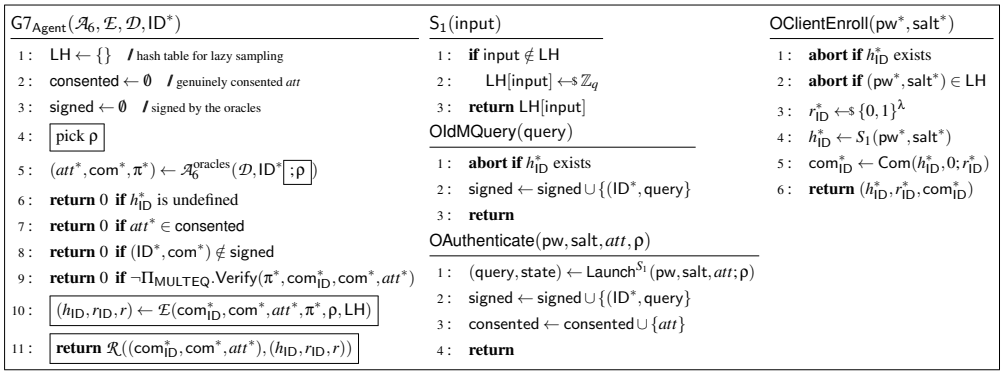

Figure 25: $\mathsf{G7}_{\mathsf{Agent}}$ Game.

We obtain

$$\mathsf{Adv}^{\mathsf{G7}_{\mathsf{Agent}}}_{\mathcal{A}_6,\mathcal{E}} \leq \mathsf{Adv}^{\mathsf{G8}_{\mathsf{Agent}}}_{\mathcal{A}_8} + 2\mathsf{Adv}^{\mathsf{BIND}}_{\mathcal{D}}$$

$\mathsf{G9}_{\mathsf{Agent}}$ **(Figure 27).** We simulate nearly everything in $\mathsf{G8}_{\mathsf{Agent}}$ to obtain an adversary $\mathcal{A}_9$ playing the $\mathsf{G9}_{\mathsf{Agent}}$ game. We have

$$\mathsf{Adv}^{\mathsf{G8}_{\mathsf{Agent}}}_{\mathcal{A}_8} \leq \mathsf{Adv}^{\mathsf{G9}_{\mathsf{Agent}}}_{\mathcal{A}_9}$$

In the game, h^*_{ID} is uniformly selected in OClientEnroll.

We split the randomness which are used in the game $\mathsf{G9}_{\mathsf{Agent}}$ as: ρ which are used for everything except h^*_{ID}, and ρ' which defines h^*_{ID}. Let I be the index of the OldMQuery which signed the commit value which is eventually opened in the end, with $I = \perp$ if the adversary does not win. We have

$$\mathsf{Adv}^{\mathsf{G9}}_{\mathcal{A}_9} = \sum_{i \neq \perp} \Pr[I = i] = \sum_{\rho, i \neq \perp} p_i(\rho) \Pr[\rho]$$

where $p_i(\rho) = \Pr[I = i|\rho]$, with the probability over ρ'.

$\mathsf{G10}_{\mathsf{Agent}}$ **(Figure 28).** We apply the forking lemma method: we define a G10 game where we simulate two executions of the game G9 with the same ρ and independent ρ'_1 and ρ'_2. The game runs the simulation and get their results. If any aborts, G10 aborts as well. Otherwise, we obtain I_1, I_2, h^*_1, h^*_2, att^*_1, att^*_2, r_1, and r_2. G10 succeeds if $I_1 = I_2 \neq \perp$, i.e. if the I-th query to OldMQuery (which must be the same in both executions since the randomness different is not effective until OClientEnroll is queried) defines a unique commit value which opens on h^*_1 and h^*_2.

We have $\Pr[I_1 = I_2 \neq \perp] = \sum p_i(\rho)^2 \Pr[\rho]$. Using the convexity of the square function, we have

$$\sum_{\rho, i \neq \perp} p_i(\rho)^2 \frac{\Pr[\rho]}{q_{\mathsf{Query}}} \geq \left(\sum_{\rho, i \neq \perp} p_i(\rho) \frac{\Pr[\rho]}{q_{\mathsf{Query}}} \right)^2$$

so we have

$$\mathsf{Adv}^{\mathsf{G9}}_{\mathcal{A}_9} \leq \sqrt{q_{\mathsf{Query}} \cdot \mathsf{Adv}^{\mathsf{G10}}_{\mathcal{A}_{10}}}$$

Wrapping up. Clearly, G10 defines an adversary $\mathcal{E}$ who breaks the binding property of the commitment unless G10 selects $h^*_1 = h^*_2$. Hence

$$\mathsf{Adv}^{\mathsf{G10}}_{\mathcal{A}_{10}} \leq \mathsf{Adv}^{\mathsf{BIND}}_{\mathcal{E}} + \frac{1}{q}$$

C Security of the IdMless Variant

C.1 Security in the $\mathsf{CUF}_{\mathsf{Agent}}$ Model

$\mathsf{G1}_{\mathsf{Agent}}$ **(Figure 29).** We expand our protocol algorithms in the IdMless $\mathsf{CUF}_{\mathsf{Agent}}$ game and introduce the random oracles H_p and H.

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

$\mathsf{G2}_{\mathsf{Agent}}$ **(Figure 30).** We generate all $(\mathsf{pw}, \mathsf{salt}, \mathsf{seed}, h)$ at the beginning of the game and generate keys from a cached seed in OClientLaunch. We also change the adversary so that it does not repeat any query to H_p and we mark as a bad event the case where the adversary queries some $(\mathsf{pw}_i, \mathsf{salt}_i)$ to H_p, which is already in LH_p (i.e., that the adversary finds pw_i). Bad events have no impact in $\mathsf{G2}_{\mathsf{Agent}}$ but we will add in the next game as a winning condition that bad events do not occur. Finally, we add as an extra winning condition that no salt shall repeat and that no collision on H_p or H occurs. Advantage gaps correspond to random collision probabilities.

$$\mathsf{Adv}^{\mathsf{G1}_{\mathsf{Agent}}}_{\mathcal{A}} \leq \mathsf{Adv}^{\mathsf{G2}_{\mathsf{Agent}}}_{\mathcal{A}_2} + \frac{q_{\mathsf{Enroll}}(q_{\mathsf{Enroll}} - 1)}{2} 2^{-\mathsf{salt}} + \frac{q_{H_p}(q_{H_p} - 1)}{2} 2^{-2\lambda} + \frac{q_H(q_H - 1)}{2q}$$

$\mathsf{G3}_{\mathsf{Agent}}$ **and** $\mathsf{G3.1}_{\mathsf{Agent}}$ **(Figure 31, Figure 32).** We add as an extra condition that none of the marked bad events occur. We can notice that when no bad event occurs, the input salt and h of OClientLaunch are always $\mathsf{salt}_{\mathsf{ID}}$ and h_{ID}, so can be omitted. Using the difference lemma, the advantage gap between $\mathsf{G2}_{\mathsf{Agent}}$ and $\mathsf{G3}_{\mathsf{Agent}}$ is bounded by the advantage of a new game $\mathsf{G3.1}_{\mathsf{Agent}}$ which succeeds when we reach any bad event.

$$\mathsf{Adv}^{\mathsf{G2}_{\mathsf{Agent}}}_{\mathcal{A}_2} \leq \mathsf{Adv}^{\mathsf{G3}_{\mathsf{Agent}}}_{\mathcal{A}_3} + \mathsf{Adv}^{\mathsf{G3.1}_{\mathsf{Agent}}}_{\mathcal{A}_2}$$

$\mathsf{G3.2}_{\mathsf{Agent}}$ **(Figure 33).** We now give the $\mathsf{seed}_{\mathsf{pw}}$ to the adversary so that the new adversary $\mathcal{A}_{3.2}$ can simulate the signature key operations and H. A bad event may occur in OClientLaunch when seed is not correct but its hash is. The collision-freeness condition on H implies that either h had ("seed:", seed) as a known preimage in LH before the oracle call, or that this is a fresh call to H with a new h, which ends up being equal to h with probability $\frac{1}{q}$. If we treat the latter case as a failing event, the adversary can provide seed as input and the oracle simplifies. Finally, we give all $(\mathsf{salt}_i, \mathsf{seed}_i)$ at the beginning instead of letting the adversary call OClientEnroll, so this oracle can also be simulated.

$$\mathsf{Adv}^{\mathsf{G3.1}_{\mathsf{Agent}}}_{\mathcal{A}} \leq \mathsf{Adv}^{\mathsf{G3.2}_{\mathsf{Agent}}}_{\mathcal{A}_{3.2}} + \frac{q_{\mathsf{Launch}}}{q}$$

$\mathsf{G3.3}_{\mathsf{Agent}}$ **(Figure 34).** If there is no preimage of seed in LH_p when OClientLaunch is invoked, the probability that this call encounters the bad event is $2^{-2\lambda}$. If we exclude this case as a failing event, the collision-freeness condition implies that there is a unique $(\mathsf{pw}', \mathsf{salt}')$ preimage and that the oracle is testing whether $\mathsf{pw}' = \mathsf{pw}_i$ and $\mathsf{salt}' = \mathsf{salt}$. Due to the non-repeating salt assumption, this means that we are testing if pw_i is equal to the unique pw' which is known by the adversary such that $H_p(\mathsf{pw}', \mathsf{salt}) = \mathsf{seed}$, or to the unique pw_j for j such that $\mathsf{salt}_j = \mathsf{salt}$. This can be emulated by using two new oracles OTest_1 and OTest_2. The other bad case in H_p

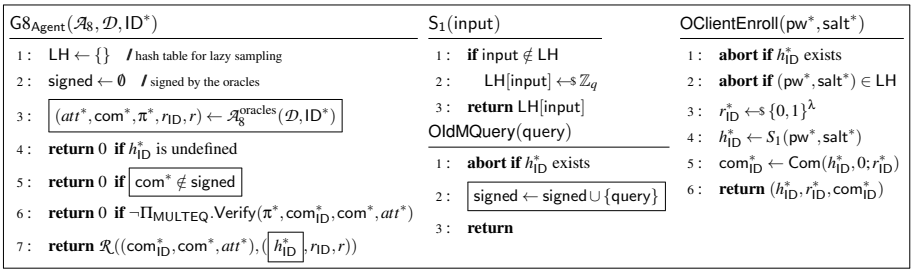

Figure 26: $\mathsf{G8}_{\mathsf{Agent}}$ Game.

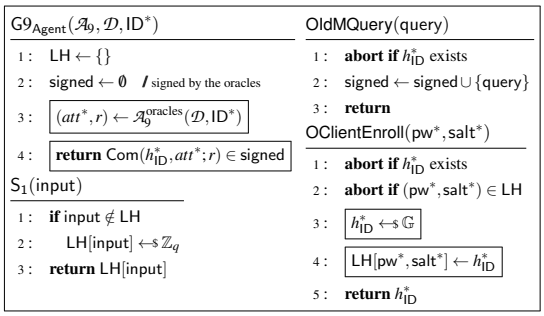

Figure 27: $\mathsf{G9}_{\mathsf{Agent}}$ Game.

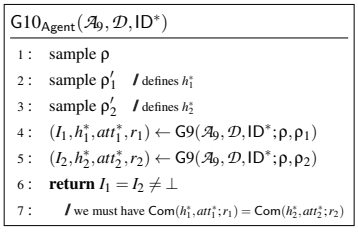

Figure 28: $\mathsf{G10}_{\mathsf{Agent}}$ Game.

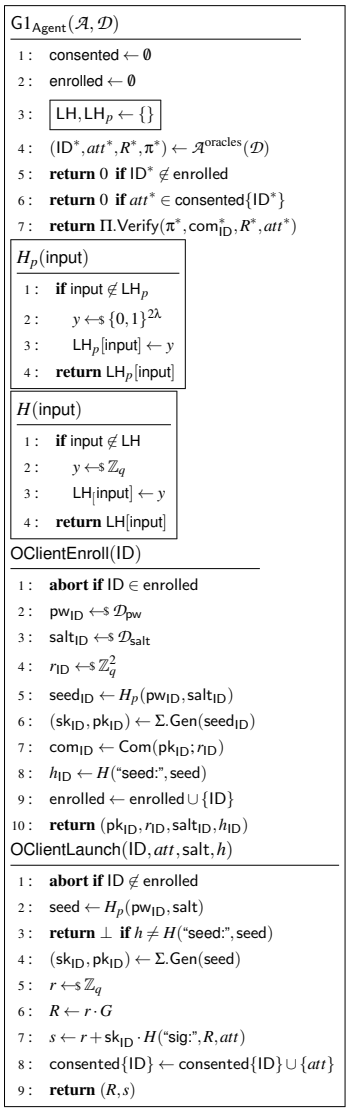

Figure 29: $\mathsf{G1}_{\mathsf{Agent}}$ for the IdMless Variant

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

consented $\leftarrow \emptyset$
enrolled $\leftarrow \emptyset$
$\mathsf{LH}, \mathsf{LH}_p \leftarrow \{\}$
for $i = 1, \ldots, q_{\mathsf{Enroll}}$ **do**
  $\mathsf{pw}_i \leftarrow\!\$\ \mathcal{D}_{\mathsf{pw}}$
  $\mathsf{salt}_i \leftarrow\!\$\ \mathcal{D}_{\mathsf{salt}}$
  $\mathsf{seed}_i \leftarrow\!\$\ \{0,1\}^{2\lambda}$
  $\mathsf{LH}_p[\mathsf{pw}_i, \mathsf{salt}_i] \leftarrow \mathsf{seed}_i$
  $h_i \leftarrow H(\text{“seed:”}, \mathsf{seed})$
abort if $\exists i, j\ \ i \neq j \wedge \mathsf{salt}_i = \mathsf{salt}_j$
$i \leftarrow 0$
$(\mathsf{ID}^*, att^*, R^*, \pi^*) \leftarrow \mathcal{A}_2^{\text{oracles}}(\mathcal{D})$
return 0 **if** $\mathsf{ID}^* \notin$ enrolled
return 0 **if** $att^* \in$ consented$\{\mathsf{ID}^*\}$
return $\Pi.\mathsf{Verify}(\pi^*, \mathsf{com}^*_{\mathsf{ID}}, R^*, att^*)$

$H_p(\mathsf{input})$

if input $\notin \mathsf{LH}_p$
  $y \leftarrow\!\$\ \{0,1\}^{2\lambda}$
  $\mathsf{G2}_{\mathsf{Agent}}$ **abort if** $\exists x\ \ \mathsf{LH}_p[x] = y$
  $\mathsf{LH}_p[\mathsf{input}] \leftarrow y$
else
  // bad event
return $\mathsf{LH}_p[\mathsf{input}]$

$H(\mathsf{input})$

if input $\notin \mathsf{LH}$
  $y \leftarrow\!\$\ \mathbb{Z}_q$
  $\mathsf{G2}_{\mathsf{Agent}}$ **abort if** $\exists x\ \ \mathsf{LH}[x] = y$
  $\mathsf{LH}[\mathsf{input}] \leftarrow y$
return $\mathsf{LH}[\mathsf{input}]$

OClientEnroll(ID)

abort if ID $\in$ enrolled $\vee i \geq q_{\mathsf{Enroll}}$
$i \leftarrow i + 1$
$(\mathsf{pw}_{\mathsf{ID}}, \mathsf{salt}_{\mathsf{ID}}, \mathsf{seed}_{\mathsf{ID}}, h_{\mathsf{ID}}) \leftarrow (\mathsf{pw}_i, \mathsf{salt}_i, \mathsf{seed}_i, h_i)$
$r_{\mathsf{ID}} \leftarrow\!\$\ \mathbb{Z}_q^2$
$(\mathsf{sk}_{\mathsf{ID}}, \mathsf{pk}_{\mathsf{ID}}) \leftarrow \Sigma.\mathsf{Gen}(\mathsf{seed}_{\mathsf{ID}})$
$\mathsf{com}_{\mathsf{ID}} \leftarrow \mathsf{Com}(\mathsf{pk}_{\mathsf{ID}}; r_{\mathsf{ID}})$
enrolled $\leftarrow$ enrolled $\cup \{\mathsf{ID}\}$
return $(\mathsf{pk}_{\mathsf{ID}}, r_{\mathsf{ID}}, \mathsf{salt}_{\mathsf{ID}}, h_{\mathsf{ID}})$

OClientLaunch(ID, att, salt, h)

abort if ID $\notin$ enrolled
if salt $\neq \mathsf{salt}_{\mathsf{ID}}$
  seed $\leftarrow H_p(\mathsf{pw}_{\mathsf{ID}}, \mathsf{salt})$
  return $\perp$ **if** $h \neq H(\text{“seed:”}, \mathsf{seed})$
  $(\mathsf{salt}_{\mathsf{ID}}, \mathsf{seed}_{\mathsf{ID}}, h_{\mathsf{ID}}) \leftarrow (\mathsf{salt}, \mathsf{seed}, h)$ // bad event
$(\mathsf{sk}_{\mathsf{ID}}, \mathsf{pk}_{\mathsf{ID}}) \leftarrow \Sigma.\mathsf{Gen}(\mathsf{seed}_{\mathsf{ID}})$
$r \leftarrow\!\$\ \mathbb{Z}_q$
$R \leftarrow r \cdot G$
$s \leftarrow r + \mathsf{sk}_{\mathsf{ID}} \cdot H(\text{“sig:”}, R, att)$
consented$\{\mathsf{ID}\} \leftarrow$ consented$\{\mathsf{ID}\} \cup \{att\}$
return (R, s)

Figure 30: $\mathsf{G2}_{\mathsf{Agent}}$ for the IdMless Variant

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

consented $\leftarrow \emptyset$
enrolled $\leftarrow \emptyset$
$\mathsf{LH}, \mathsf{LH}_p \leftarrow \{\}$
for $i = 1, \ldots, q_{\mathsf{Enroll}}$ **do**
  $\mathsf{pw}_i \leftarrow\!\$\ \mathcal{D}_{\mathsf{pw}}$
  $\mathsf{salt}_i \leftarrow\!\$\ \mathcal{D}_{\mathsf{salt}}$
  $\mathsf{seed}_i \leftarrow\!\$\ \{0,1\}^{2\lambda}$
  $\mathsf{LH}_p[\mathsf{pw}_i, \mathsf{salt}_i] \leftarrow \mathsf{seed}_i$
  $h_i \leftarrow H(\text{“seed:”}, \mathsf{seed})$
abort if $\exists i, j\ \ i \neq j \wedge \mathsf{salt}_i = \mathsf{salt}_j$
$i \leftarrow 0$
$(\mathsf{ID}^*, att^*, R^*, \pi^*) \leftarrow \mathcal{A}_3^{\text{oracles}}(\mathcal{D})$
return 0 **if** $\mathsf{ID}^* \notin$ enrolled
return 0 **if** $att^* \in$ consented$\{\mathsf{ID}^*\}$
return $\Pi.\mathsf{Verify}(\pi^*, \mathsf{com}^*_{\mathsf{ID}}, R^*, att^*)$

$H_p(\mathsf{input})$

$\mathsf{G3}_{\mathsf{Agent}}$ **abort if** input $\in \mathsf{LH}_p$
$y \leftarrow\!\$\ \{0,1\}^{2\lambda}$
$\mathsf{G3}_{\mathsf{Agent}}$ **abort if** $\exists x\ \ \mathsf{LH}_p[x] = y$
$\mathsf{LH}_p[\mathsf{input}] \leftarrow y$
return $\mathsf{LH}_p[\mathsf{input}]$

$H(\mathsf{input})$

if input $\notin \mathsf{LH}$
  $y \leftarrow\!\$\ \mathbb{Z}_q$
  $\mathsf{G3}_{\mathsf{Agent}}$ **abort if** $\exists x\ \ \mathsf{LH}[x] = y$
  $\mathsf{LH}[\mathsf{input}] \leftarrow y$
return $\mathsf{LH}[\mathsf{input}]$

OClientEnroll(ID)

abort if ID $\in$ enrolled $\vee i \geq q_{\mathsf{Enroll}}$
$i \leftarrow i + 1$
$(\mathsf{pw}_{\mathsf{ID}}, \mathsf{salt}_{\mathsf{ID}}, \mathsf{seed}_{\mathsf{ID}}, h_{\mathsf{ID}}) \leftarrow (\mathsf{pw}_i, \mathsf{salt}_i, \mathsf{seed}_i, h_i)$
$r_{\mathsf{ID}} \leftarrow\!\$\ \mathbb{Z}_q^2$
$(\mathsf{sk}_{\mathsf{ID}}, \mathsf{pk}_{\mathsf{ID}}) \leftarrow \Sigma.\mathsf{Gen}(\mathsf{seed}_{\mathsf{ID}})$
$\mathsf{com}_{\mathsf{ID}} \leftarrow \mathsf{Com}(\mathsf{pk}_{\mathsf{ID}}; r_{\mathsf{ID}})$
enrolled $\leftarrow$ enrolled $\cup \{\mathsf{ID}\}$
return $(\mathsf{pk}_{\mathsf{ID}}, r_{\mathsf{ID}}, \mathsf{salt}_{\mathsf{ID}}, h_{\mathsf{ID}})$

OClientLaunch(ID, att)

abort if ID $\notin$ enrolled
$r \leftarrow\!\$\ \mathbb{Z}_q$
$R \leftarrow r \cdot G$
$s \leftarrow r + \mathsf{sk}_{\mathsf{ID}} \cdot H(\text{“sig:”}, R, att)$
consented$\{\mathsf{ID}\} \leftarrow$ consented$\{\mathsf{ID}\} \cup \{att\}$
return (R, s)

Figure 31: $\mathsf{G3}_{\mathsf{Agent}}$ for the IdMless Variant

$\mathsf{G3.1}_{\mathsf{Agent}}(\mathcal{A}_2, \mathcal{D})$

$\mathsf{enrolled} \leftarrow \emptyset$
$\mathsf{LH}, \mathsf{LH}_p \leftarrow \{\}$
for $i = 1, \ldots, q_{\mathsf{Enroll}}$ **do**
  $\mathsf{pw}_i \leftarrow\$ \mathcal{D}_{\mathsf{pw}}$
  $\mathsf{salt}_i \leftarrow\$ \mathcal{D}_{\mathsf{salt}}$
  $\mathsf{seed}_i \leftarrow\$ \{0,1\}^{2\lambda}$
  $\mathsf{LH}_p[\mathsf{pw}_i, \mathsf{salt}_i] \leftarrow \mathsf{seed}_i$
  $h_i \leftarrow H(\text{“seed:”}, \mathsf{seed})$
abort if $\exists i, j \;\; i \neq j \wedge \mathsf{salt}_i = \mathsf{salt}_j$
$i \leftarrow 0$
$\mathcal{A}_2^{\text{oracles}}(\mathcal{D})$
return 0

$H_p(\mathsf{input})$

return $\mathsf{G3.1}_{\mathsf{Agent}}$ **return** 1 **if** $\mathsf{input} \in \mathsf{LH}_p$
$y \leftarrow\$ \{0,1\}^{2\lambda}$
$\mathsf{G3.1}_{\mathsf{Agent}}$ **abort if** $\exists x \;\; \mathsf{LH}_p[x] = y$
$\mathsf{LH}_p[\mathsf{input}] \leftarrow y$
return $\mathsf{LH}_p[\mathsf{input}]$

$H(\mathsf{input})$

if $\mathsf{input} \notin \mathsf{LH}$
  $y \leftarrow\$ \mathbb{Z}_q$
  $\mathsf{G3.1}_{\mathsf{Agent}}$ **abort if** $\exists x \;\; \mathsf{LH}[x] = y$
  $\mathsf{LH}[\mathsf{input}] \leftarrow y$
return $\mathsf{LH}[\mathsf{input}]$

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

abort if $\mathsf{ID} \in \mathsf{enrolled} \vee i \geq q_{\mathsf{Enroll}}$
$i \leftarrow i + 1$
$(\mathsf{pw}_{\mathsf{ID}}, \mathsf{salt}_{\mathsf{ID}}, \mathsf{seed}_{\mathsf{ID}}, h_{\mathsf{ID}}) \leftarrow (\mathsf{pw}_i, \mathsf{salt}_i, \mathsf{seed}_i, h_i)$
$r_{\mathsf{ID}} \leftarrow\$ \mathbb{Z}_q^2$
$(\mathsf{sk}_{\mathsf{ID}}, \mathsf{pk}_{\mathsf{ID}}) \leftarrow \Sigma.\mathsf{Gen}(\mathsf{seed}_{\mathsf{ID}})$
$\mathsf{com}_{\mathsf{ID}} \leftarrow \mathsf{Com}(\mathsf{pk}_{\mathsf{ID}}; r_{\mathsf{ID}})$
$\mathsf{enrolled} \leftarrow \mathsf{enrolled} \cup \{\mathsf{ID}\}$
return $(\mathsf{pk}_{\mathsf{ID}}, r_{\mathsf{ID}}, \mathsf{salt}_{\mathsf{ID}}, h_{\mathsf{ID}})$

$\mathsf{OClientLaunch}(\mathsf{ID}, att, \mathsf{salt}, h)$

abort if $\mathsf{ID} \notin \mathsf{enrolled}$
if $\mathsf{salt} \neq \mathsf{salt}_{\mathsf{ID}}$
  $\mathsf{seed} \leftarrow H_p(\mathsf{pw}_{\mathsf{ID}}, \mathsf{salt})$
  return $\perp$ **if** $h \neq H(\text{“seed:”}, \mathsf{seed})$
  $\mathsf{G3.1}_{\mathsf{Agent}}$ **return** 1
$r \leftarrow\$ \mathbb{Z}_q$
$R \leftarrow r \cdot G$
$s \leftarrow r + \mathsf{sk}_{\mathsf{ID}} \cdot H(\text{“sig:”}, R, att)$
return (R, s)

Figure 32: $\mathsf{G3.1}_{\mathsf{Agent}}$ for the IdMless Variant

$\mathsf{G3.2}_{\mathsf{Agent}}(\mathcal{A}_{3.2}, \mathcal{D})$

$\mathsf{LH}_p \leftarrow \{\}$
for $i = 1, \ldots, q_{\mathsf{Enroll}}$ **do**
  $\mathsf{pw}_i \leftarrow\$ \mathcal{D}_{\mathsf{pw}}$
  $\mathsf{salt}_i \leftarrow\$ \mathcal{D}_{\mathsf{salt}}$
  $\mathsf{seed}_i \leftarrow\$ \{0,1\}^{2\lambda}$
  $\mathsf{LH}_p[\mathsf{pw}_i, \mathsf{salt}_i] \leftarrow \mathsf{seed}_i$
abort if $\exists i, j \;\; i \neq j \wedge \mathsf{salt}_i = \mathsf{salt}_j$
$\mathcal{A}_{3.2}^{\text{oracles}}(\mathcal{D}, \boxed{(\mathsf{salt}_i, \mathsf{seed}_i)_{i=1,\ldots,q_{\mathsf{Enroll}}}})$
return 0

$H_p(\mathsf{input})$

$\mathsf{G3.2}_{\mathsf{Agent}}$ **return** 1 **if** $\mathsf{input} \in \mathsf{LH}_p$
$y \leftarrow\$ \{0,1\}^{2\lambda}$
$\mathsf{G3.2}_{\mathsf{Agent}}$ **abort if** $\exists x \;\; \mathsf{LH}_p[x] = y$
$\mathsf{LH}_p[\mathsf{input}] \leftarrow y$
return $\mathsf{LH}_p[\mathsf{input}]$

$\mathsf{OClientLaunch}(\boxed{i}, \mathsf{salt}, \boxed{\mathsf{seed}})$

return if $\mathsf{salt} = \boxed{\mathsf{salt}_i}$
$\boxed{\mathsf{G3.2}_{\mathsf{Agent}}$ **return** 1 **if** $\mathsf{seed} = H_p(\mathsf{pw}_i, \mathsf{salt})}$
return

Figure 33: $\mathsf{G3.2}_{\mathsf{Agent}}$ for the IdMless Variant

$\mathsf{G3.3}_{\mathsf{Agent}}(\mathcal{A}_{3.3}, \mathcal{D})$

for $i = 1, \ldots, q_{\mathsf{Enroll}}$ **do**
  $\mathsf{pw}_i \leftarrow\$ \mathcal{D}_{\mathsf{pw}}$
$\mathcal{A}_{3.3}^{\text{oracles}}(\mathcal{D})$
return 0

$\mathsf{OTest}_1(i, \mathsf{pw})$

$\mathsf{G3.3}_{\mathsf{Agent}}$ **return** 1 **if** $\mathsf{pw}_i = \mathsf{pw}$
return

$\mathsf{OTest}_2(i, j)$

return if $i = j$
$\mathsf{G3.3}_{\mathsf{Agent}}$ **return** 1 **if** $\mathsf{pw}_i = \mathsf{pw}_j$
return

Figure 34: $\mathsf{G3.3}_{\mathsf{Agent}}$ for the IdMless Variant

can be emulated using OTest_1. In this new game, H_p can be simulated as well.

$$\mathsf{Adv}^{\mathsf{G3.2}_{\mathsf{Agent}}}_{\mathcal{A}_{3.2}} \leq \mathsf{Adv}^{\mathsf{G3.3}_{\mathsf{Agent}}}_{\mathcal{A}_{3.3}} + \frac{q_{\mathsf{Launch}}}{2^{2\lambda}}$$

We have $q_{\mathsf{Test}} = q_{\mathsf{Launch}} + q_{\mathsf{H_p}}$ queries to either OTest oracle. **Bounding $\mathsf{G3.3}_{\mathsf{Agent}}$ (Figure 34).** The G3.3 game can generate q_{Test} hybrid games where all queries to the OTest oracles except the kth one are simulated by doing nothing. Each hybrid returns 1 with probability bounded by $2^{-E_{\mathsf{pw}}}$.

$$\mathsf{Adv}^{\mathsf{G3.3}_{\mathsf{Agent}}}_{\mathcal{A}_{3.3}} \leq \frac{q_{H_p} + q_{\mathsf{Launch}}}{2^{E_{\mathsf{pw}}}}$$

$\mathsf{G4}_{\mathsf{Agent}}$ (Figure 35). As $(\mathsf{pw}_i, \mathsf{salt}_i)$ is never queried again to H_p, we can stop storing it in LH_p. As a result, H_p can be simulated by the adversary. Finally, pw_i and salt_i do not play

$\mathsf{G4}_{\mathsf{Agent}}(\mathcal{A}_4, \mathcal{D})$

$\mathsf{consented} \leftarrow \emptyset$
$\mathsf{enrolled} \leftarrow \emptyset$
$\mathsf{LH} \leftarrow \{\}$
for $i = 1, \ldots, q_{\mathsf{Enroll}}$ **do**
  $\mathsf{seed}_i \leftarrow\$ \{0,1\}^{2\lambda}$
  $h_i \leftarrow H(\text{“seed:”}, \mathsf{seed})$
$i \leftarrow 0$
$(\mathsf{ID}^*, att^*, R^*, \pi^*) \leftarrow \mathcal{A}_4^{\text{oracles}}(\mathcal{D})$
return 0 **if** $\mathsf{ID}^* \notin$ enrolled
return 0 **if** $att^* \in \mathsf{consented}\{\mathsf{ID}^*\}$
return $\Pi.\mathsf{Verify}(\pi^*, \mathsf{com}^*_{\mathsf{ID}}, R^*, att^*)$

$H(\mathsf{input})$

if $\mathsf{input} \notin \mathsf{LH}$
  $y \leftarrow\$ \mathbb{Z}_q$
  $\mathsf{G4}_{\mathsf{Agent}}$ **abort if** $\exists x \ \mathsf{LH}[x] = y$
  $\mathsf{LH}[\mathsf{input}] \leftarrow y$
return $\mathsf{LH}[\mathsf{input}]$

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

abort if $\mathsf{ID} \in \mathsf{enrolled} \vee i \geq q_{\mathsf{Enroll}}$
$i \leftarrow i + 1$
$(\mathsf{seed}_{\mathsf{ID}}, h_{\mathsf{ID}}) \leftarrow (\mathsf{seed}_i, h_i)$
$r_{\mathsf{ID}} \leftarrow\$ \mathbb{Z}_q^2$
$(\mathsf{sk}_{\mathsf{ID}}, \mathsf{pk}_{\mathsf{ID}}) \leftarrow \Sigma.\mathsf{Gen}(\mathsf{seed}_{\mathsf{ID}})$
$\mathsf{com}_{\mathsf{ID}} \leftarrow \mathsf{Com}(\mathsf{pk}_{\mathsf{ID}}; r_{\mathsf{ID}})$
$\mathsf{enrolled} \leftarrow \mathsf{enrolled} \cup \{\mathsf{ID}\}$
return $(\mathsf{pk}_{\mathsf{ID}}, r_{\mathsf{ID}}, h_{\mathsf{ID}})$

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

abort if $\mathsf{ID} \notin \mathsf{enrolled}$
$r \leftarrow\$ \mathbb{Z}_q$
$R \leftarrow r \cdot G$
$s \leftarrow r + \mathsf{sk}_{\mathsf{ID}} \cdot H(\text{“sig:”}, R, att)$
$\mathsf{consented}\{\mathsf{ID}\} \leftarrow \mathsf{consented}\{\mathsf{ID}\} \cup \{att\}$
return (R, s)

Figure 35: $\mathsf{G4}_{\mathsf{Agent}}$ for the IdMless Variant

roles any more and can be removed.

$$\mathsf{Adv}^{\mathsf{G3}_{\mathsf{Agent}}}_{\mathcal{A}_3} \leq \mathsf{Adv}^{\mathsf{G4}_{\mathsf{Agent}}}_{\mathcal{A}_4}$$

$\mathsf{G5}_{\mathsf{Agent}}$ **(Figure 36).** We make all enrollments at the beginning and use i instead of ID. We also now sample the h_i without adding them in the LH table. The failure event is when the adversary makes in $\mathsf{G4}_{\mathsf{Agent}}$ a call to H with the query $(\text{“seed:”}, \mathsf{seed}_i)$, which implies that the adversary has found one seed. This can be turned into an adversary recovering sk_i in a multi-target setting. Other differences are when the freshly sampled y in H ends up being equal to h_i, but this case was never making $\mathsf{G4}_{\mathsf{Agent}}$ returning 1 due to the collision freeness condition in H. Therefore,

$$\mathsf{Adv}^{\mathsf{G4}_{\mathsf{Agent}}}_{\mathcal{A}_4} \leq \mathsf{Adv}^{\mathsf{G5}_{\mathsf{Agent}}}_{\mathcal{A}_5} + \mathsf{Adv}^{\mathsf{mKR}}_{\mathcal{B}}(q_{\mathsf{Enroll}})$$

$\mathsf{G6}_{\mathsf{Agent}}$ **(Figure 37).** Now, each h_i can be simulated by the adversary and seed_i is only used during the key generation. We further simulate the $\mathsf{OClientLaunch}$ oracle by first picking s and the hash then deducing R and programming the random

$\mathsf{G5}_{\mathsf{Agent}}(\mathcal{A}_5, \mathcal{D})$

$\mathsf{consented} \leftarrow \emptyset$
$\mathsf{LH} \leftarrow \{\}$
for $i = 1, \ldots, q_{\mathsf{Enroll}}$ **do**
  $\mathsf{seed}_i \leftarrow\$ \{0,1\}^{2\lambda}$
  $\boxed{h_i \leftarrow\$ \mathbb{Z}_q}$
  $\boxed{r_i \leftarrow\$ \mathbb{Z}_q^2}$
  $\boxed{(\mathsf{sk}_i, \mathsf{pk}_i) \leftarrow \Sigma.\mathsf{Gen}(\mathsf{seed}_i)}$
  $\boxed{\mathsf{com}_i \leftarrow \mathsf{Com}(\mathsf{pk}_i; r_i)}$
$(\boxed{i}, att^*, R^*, \pi^*) \leftarrow \mathcal{A}_5^{\text{oracles}}(\mathcal{D}, \boxed{(\mathsf{pk}_i, r_i, h_i)_{i=1,\ldots,q_{\mathsf{Enroll}}}})$
return 0 **if** $att^* \in \mathsf{consented}\{\boxed{i}\}$
return $\Pi.\mathsf{Verify}(\pi^*, \boxed{\mathsf{com}_i}, R^*, att^*)$

$H(\mathsf{input})$

if $\mathsf{input} \notin \mathsf{LH}$
  $y \leftarrow\$ \mathbb{Z}_q$
  $\mathsf{G5}_{\mathsf{Agent}}$ **abort if** $\exists x \ \mathsf{LH}[x] = y$
  $\mathsf{LH}[\mathsf{input}] \leftarrow y$
return $\mathsf{LH}[\mathsf{input}]$

$\mathsf{OClientLaunch}(\boxed{i}, att)$

$r \leftarrow\$ \mathbb{Z}_q$
$R \leftarrow r \cdot G$
$s \leftarrow r + \boxed{\mathsf{sk}_i} \cdot H(\text{“sig:”}, R, att)$
$\boxed{\mathsf{consented}\{i\} \leftarrow \mathsf{consented}\{i\} \cup \{att\}}$
return (R, s)

Figure 36: $\mathsf{G5}_{\mathsf{Agent}}$ for the IdMless Variant

oracle. It produces a view to the adversary which has the same distribution.

$$\mathsf{Adv}^{\mathsf{G5}_{\mathsf{Agent}}}_{\mathcal{A}_5} \leq \mathsf{Adv}^{\mathsf{G6}_{\mathsf{Agent}}}_{\mathcal{A}_6}$$

The execution of the adversary during the game can now be completely simulated by an extractor.

$\mathsf{G7}_{\mathsf{Agent}}$ **(Figure 38).** We use the Forking Lemma [Pointcheval and Stern(1996)] to extract a witness (s, r). We set ρ as the random coins of the adversary. Given ρ and LH, an extractor can perfectly simulate the same execution as the adversary in the game. Instead of using LH as a table, we use the list L of input/output to the H oracle. When the adversary succeeds, there must be an oracle call to H which defines the challenge $e = H(\text{“zk:”}, G, \mathsf{com}_i, att^*, R^*, C)$ in π^*. We let I be the index of this oracle call. We have $I = \perp$ if the adversary does not succeed and $1 \leq I \leq q_h$ otherwise. In the case of a success, $\mathcal{E}$ sets L' as the prefix of L of length $I-1$. Then, it restarts the simulation with the same ρ and a random L starting by L'. We let ρ' be the random coins of the game to define the tail of L in this second simulation. Let $p = \mathsf{Adv}^{\mathsf{G6}_{\mathsf{Agent}}}_{\mathcal{A}_6}$ and $p(\rho, \Lambda)$ be the probability that $\mathsf{G6}_{\mathsf{Agent}}$ returns $1, I-1$ equals the size of Λ, and Λ is a prefix of L, over the distribution of ρ'. We have $\mathsf{Adv}^{\mathsf{G6}_{\mathsf{Agent}}}_{\mathcal{A}_6} = p = \sum_i \sum_{\rho,\Lambda} p(\rho, \Lambda) \Pr[\rho, \Lambda]$ where the sum is over ρ and over Λ of size $i-1$. So, $\frac{p}{q_H}$ is the average of $p(\rho, \Lambda)$. We say that the fork succeeds if it goes to the second simulation and this second one ends with the

$\mathsf{G6}_{\mathsf{Agent}}(\mathcal{A}_6, \mathcal{D})$

consented $\leftarrow \emptyset$
LH $\leftarrow \{\}$
for $i = 1, \ldots, q_{\mathsf{Enroll}}$ **do**
  $r_i \leftarrow\$ \mathbb{Z}_q^2$
  $(\mathsf{sk}_i, \mathsf{pk}_i) \leftarrow$ $\boxed{\Sigma.\mathsf{Gen}}$
  $\mathsf{com}_i \leftarrow \mathsf{Com}(\mathsf{pk}_i; r_i)$
$(i, att^*, R^*, \pi^*) \leftarrow \mathcal{A}_6^{\mathsf{oracles}}(\mathcal{D}, (\mathsf{pk}_i, r_i)_{i=1,\ldots,q_{\mathsf{Enroll}}})$
return 0 **if** $att^* \in \mathsf{consented}[i]$
return $\Pi.\mathsf{Verify}(\pi^*, \mathsf{com}_i, R^*, att^*)$

$H(\mathsf{input})$

if input $\notin$ LH
  $y \leftarrow\$ \mathbb{Z}_q$
  $\mathsf{G6}_{\mathsf{Agent}}$ **abort if** $\exists x$ $\mathsf{LH}[x] = y$
  $\mathsf{LH}[\mathsf{input}] \leftarrow y$
return LH[input]

$\mathsf{OClientLaunch}(i, att)$

$\boxed{s, y \leftarrow\$ \mathbb{Z}_q}$
$\boxed{\mathsf{G6}_{\mathsf{Agent}}\ \textbf{return}\ \perp\ \textbf{if}\ \exists x\ \mathsf{LH}[x] = y}$
$\boxed{R \leftarrow s \cdot G - y \cdot \mathsf{pk}_i}$
$\boxed{\mathsf{input} \leftarrow (\text{“sig:”}, R, att)}$
$\boxed{\mathsf{G6}_{\mathsf{Agent}}\ \textbf{abort if}\ \mathsf{input} \in \mathsf{LH}}$
$\boxed{\mathsf{LH}[\mathsf{input}] \leftarrow y}$
consented$[i] \leftarrow$ consented$[i] \cup \{att\}$
return (R, s)

Figure 37: $\mathsf{G6}_{\mathsf{Agent}}$ for the IdMless Variant

same I, i.e. with probability $\sum_i \sum_{\rho,\Lambda} p(\rho,\Lambda)^2 \Pr[\rho,\Lambda]$. Using the Jensen inequality, the fork succeeds with probability at least $\frac{p^2}{q_H}$. The fork succeeds but have the same Ith result from H with probability at most $\frac{p}{q}$. Hence, Σ-extraction succeeds with probability at least $p(\frac{p}{q_H} - \frac{1}{q})$. We deduce

$$\mathsf{Adv}^{\mathsf{G6}_{\mathsf{Agent}}}_{\mathcal{A}_6} \leq \frac{q_H}{q} + \sqrt{q_H \cdot \mathsf{Adv}^{\mathsf{G7}_{\mathsf{Agent}}}_{\mathcal{A}_6, \mathcal{E}}}$$

$\mathsf{G8}_{\mathsf{Agent}}$ **(Figure 39).** We simulate most of the game inside the new adversary $\mathcal{A}_8$. Except when the adversary breaks the binding property, we obtain (s, r) such that $s \cdot G = R^* + H(\text{“sig:”}, R, att) \cdot \mathsf{pk}_i$ which we use as a new winning condition. We also revert the changes from OClientLaunch. We finally remove r_i and com_i which are no longer used.

$$\mathsf{Adv}^{\mathsf{G7}_{\mathsf{Agent}}}_{\mathcal{A}_6, \mathcal{E}} \leq \mathsf{Adv}^{\mathsf{G8}_{\mathsf{Agent}}}_{\mathcal{A}_8} + \mathsf{Adv}^{\mathsf{BIND}}_{C}$$

Wrapping up. Clearly, $\mathsf{G9}_{\mathsf{Agent}}$ reduces to a forgery game against the Schnorr signature scheme in a multi-target setting.

$$\mathsf{Adv}^{\mathsf{G9}_{\mathsf{Agent}}}_{\mathcal{A}_9} \leq \mathsf{Adv}^{\mathsf{mUF}}_{\mathcal{D}}(q_{\mathsf{Enroll}})$$

C.2 Security in the $\mathsf{CUF}_{\mathsf{IdMless}}$ Model

$\mathsf{G1}_{\mathsf{IdMless}}$ **(Figure 40).** We expand our protocol algorithms in the IdMless $\mathsf{CUF}_{\mathsf{IdMless}}$ game and introduce the random

$\mathsf{G7}_{\mathsf{Agent}}(\mathcal{A}_6, \mathcal{E}, \mathcal{D})$

consented $\leftarrow \emptyset$
LH $\leftarrow \{\}$
for $i = 1, \ldots, q_{\mathsf{Enroll}}$ **do**
  $r_i \leftarrow\$ \mathbb{Z}_q^2$
  $(\mathsf{sk}_i, \mathsf{pk}_i) \leftarrow \Sigma.\mathsf{Gen}$
  $\mathsf{com}_i \leftarrow \mathsf{Com}(\mathsf{pk}_i; r_i)$
$\boxed{\text{pick } \rho}$
$(i, att^*, R^*, \pi^*) \leftarrow \mathcal{A}_6^{\mathsf{oracles}}(\mathcal{D}, (\mathsf{pk}_i, r_i)_{i=1,\ldots,q_{\mathsf{Enroll}}}; \boxed{\rho})$
return 0 **if** $att^* \in \mathsf{consented}[i]$
$\boxed{(s, r) \leftarrow \mathcal{E}((\mathsf{pk}_i, r_i)_{i=1,\ldots,q_{\mathsf{Enroll}}}, \rho, \mathsf{LH})}$
$\boxed{\textbf{return}\ 0\ \textbf{if}\ \Pi.\mathsf{Verify}(\pi^*, \mathsf{com}^*, R^*, att^*) = 0}$
$\boxed{\textbf{return}\ \mathsf{Com}(s \cdot G; r) = \mathsf{Com}(R^*; 0) + H(\text{“sig:”}, R^*, att^*) \cdot \mathsf{com}_i}$

$H(\mathsf{input})$

if input $\notin$ LH
  $y \leftarrow\$ \mathbb{Z}_q$
  $\mathsf{G7}_{\mathsf{Agent}}$ **abort if** $\exists x$ $\mathsf{LH}[x] = y$
  $\mathsf{LH}[\mathsf{input}] \leftarrow y$
return LH[input]

$\mathsf{OClientLaunch}(i, att)$

$s, y \leftarrow\$ \mathbb{Z}_q$
$\mathsf{G7}_{\mathsf{Agent}}$ **return** $\perp$ **if** $\exists x$ $\mathsf{LH}[x] = y$
$R \leftarrow s \cdot G - y \cdot \mathsf{pk}_i$
input $\leftarrow$ (“sig:”, R, att)
$\mathsf{G7}_{\mathsf{Agent}}$ **abort if** input $\in$ LH
$\mathsf{LH}[\mathsf{input}] \leftarrow y$
consented$[i] \leftarrow$ consented$[i] \cup \{att\}$
return (R, s)

Figure 38: $\mathsf{G7}_{\mathsf{Agent}}$ for the IdMless Variant

$\mathsf{G8}_{\mathsf{Agent}}(\mathcal{A}_8, \mathcal{D})$

consented $\leftarrow \emptyset$
LH $\leftarrow \{\}$
for $i = 1, \ldots, q_{\mathsf{Enroll}}$ **do**
  $(\mathsf{sk}_i, \mathsf{pk}_i) \leftarrow \Sigma.\mathsf{Gen}(1^\lambda)$
$\boxed{(i, att^*, R^*, s) \leftarrow \mathcal{A}_8^{\mathsf{oracles}}(\mathcal{D}, (\mathsf{pk}_i)_{i=1,\ldots,q_{\mathsf{Enroll}}})}$
return 0 **if** $att^* \in \mathsf{consented}[i]$
$\boxed{\textbf{return}\ s \cdot G = H(\text{“sig:”}, R^*, att^*) \cdot \mathsf{pk}_i + R^*}$

$H(\mathsf{input})$

if input $\notin$ LH
  $y \leftarrow\$ \mathbb{Z}_q$
  $\mathsf{G8}_{\mathsf{Agent}}$ **abort if** $\exists x$ $\mathsf{LH}[x] = y$
  $\mathsf{LH}[\mathsf{input}] \leftarrow y$
return LH[input]

$\mathsf{OClientLaunch}(i, att)$

$\boxed{r \leftarrow\$ \mathbb{Z}_q}$
$\boxed{R \leftarrow r \cdot G}$
$\boxed{s \leftarrow r + \mathsf{sk}_i \cdot H(\text{“sig:”}, R, att)}$
consented $\leftarrow$ consented $\cup \{att\}$
return (R, s)

Figure 39: $\mathsf{G8}_{\mathsf{Agent}}$ for the IdMless Variant

oracles H_p and H.

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

$\mathsf{G2}_{\mathsf{IdMless}}$ **(Figure 41).** We replace the proof generation in OClientLaunch and in OClientOrder by simulators in S_2 then S_2' (as these oracles always provide valid witnesses). The failing event is when input was already queried.

$$\mathsf{Adv}^{\mathsf{G1}_{\mathsf{IdMless}}}_{\mathcal{A}} \leq \frac{(q_H + q_{\mathsf{Launch}} + q_{\mathsf{Order}})^2}{q^2} + \mathsf{Adv}^{\mathsf{G2}_{\mathsf{IdMless}}}_{\mathcal{A}}$$

Note that the adversary does not access S_2'.

$\mathsf{G3}_{\mathsf{IdMless}}$ **(Figure 42).** We note that r_{ID} is no longer used after $\mathsf{com}_{\mathsf{ID}}$ is set. We use the hiding property of the commitment to replace all $\mathsf{com}_{\mathsf{ID}}$ by some random commitment of the same random group element x. Furthermore, the correctness of the Schnorr signature implies that generating s then checking $s \cdot G = R + S_1(\text{“sig:”}, R, att) \cdot \mathsf{pk}_{\mathsf{ID}}$ in OClientLaunch is equivalent to checking $\mathsf{pk} = \mathsf{pk}_{\mathsf{ID}}$.

$$\mathsf{Adv}^{\mathsf{G2}_{\mathsf{IdMless}}}_{\mathcal{A}} \leq q_{\mathsf{Enroll}} \cdot \mathsf{Adv}^{\mathsf{HIDE}}_{\mathcal{B}} + \mathsf{Adv}^{\mathsf{G3}_{\mathsf{IdMless}}}_{\mathcal{A}}$$

$\mathsf{G4}_{\mathsf{IdMless}}$ **and** $\mathsf{G4.1}_{\mathsf{IdMless}}$ **(Figure 43, Figure 44).** We now mark which (att, R, π) has been issued by OClientLaunch or OClientOrder. We add in $\mathsf{G4}_{\mathsf{IdMless}}$ the condition that the triplet issued by the adversary must be in this set. The failing event is when it generates a triplet which is not in the set. This is modeled by the game G4.1 where a new adversary $\mathcal{A}_{4.1}$ simulates, H_p, OClientEnroll, OClientLaunch, OAgentOrder, and OAgentSalt with access to S_2'.

$$\mathsf{Adv}^{\mathsf{G3}_{\mathsf{IdMless}}}_{\mathcal{A}} \leq \mathsf{Adv}^{\mathsf{G4.1}_{\mathsf{IdMless}}}_{\mathcal{A}} + \mathsf{Adv}^{\mathsf{G4}_{\mathsf{IdMless}}}_{\mathcal{A}}$$

$\mathsf{G4.2}_{\mathsf{IdMless}}$ **(Figure 45).** We use the extractor technique.

$$\mathsf{Adv}^{\mathsf{G4.1}_{\mathsf{IdMless}}}_{\mathcal{A}_{4.1}} \leq \frac{q_H + q_{\mathsf{Launch}} + q_{\mathsf{Order}}}{q} + \sqrt{(q_H + q_{\mathsf{Launch}} + q_{\mathsf{Order}}) \cdot \mathsf{Adv}^{\mathsf{G4.2}_{\mathsf{IdMless}}}_{\mathcal{A}_{4.1}, \mathcal{E}}}$$

$\mathsf{G4.3}_{\mathsf{IdMless}}$ **(Figure 46).** We now simulate more into a new adversary which returns $\mathsf{pk} = \frac{1}{H(\text{“sig:”}, R, att^*)}(s \cdot G - R)$.

$$\mathsf{Adv}^{\mathsf{G4.2}_{\mathsf{IdMless}}}_{\mathcal{A}_{4.1}} \leq \mathsf{Adv}^{\mathsf{G4.3}_{\mathsf{IdMless}}}_{\mathcal{A}_{4.3}}$$

$\mathsf{G4.4}_{\mathsf{IdMless}}$ **(Figure 47).** We now define a new adversary who simulates the oracle and opens a random commitment.

$$\mathsf{Adv}^{\mathsf{G4.3}_{\mathsf{IdMless}}}_{\mathcal{A}_{4.3}} \leq \mathsf{Adv}^{\mathsf{G4.4}_{\mathsf{IdMless}}}_{\mathcal{A}_{4.4}}$$

Wrapping up G4.4**.** Clearly, the G4.4 boils down to a game breaking either the hiding property of the binding property.

$$\mathsf{Adv}^{\mathsf{G4.3}_{\mathsf{IdMless}}}_{\mathcal{A}_{4.4}} \leq \mathsf{Adv}^{\mathsf{HIDE}}_{\mathcal{C}} + \mathsf{Adv}^{\mathsf{BIND}}_{\mathcal{D}}$$

$\mathsf{G1}_{\mathsf{IdMless}}(\mathcal{A}, \mathcal{D})$

consented $\leftarrow \emptyset$ // genuinely consented att
enrolled $\leftarrow \emptyset$ // enrolled honest clients
$\boxed{\mathsf{LH}, \mathsf{LH}_p \leftarrow \{\}}$
$(\mathsf{ID}^*, att^*, R, e, r_1, r_2) \leftarrow \mathcal{A}^{\text{oracles}}(\mathcal{D})$
return 0 **if** $\mathsf{ID}^* \notin$ enrolled
return 0 **if** $att^* \in$ consented$\{\mathsf{ID}^*\}$
return Verify$(\mathsf{com}_{\mathsf{ID}^*}, att^*, R, e, r_1, r_2)$

$H_p(\mathsf{input})$

if input $\notin \mathsf{LH}_p$
  $y \leftarrow\$ \{0,1\}^{2\lambda}$
  $\mathsf{LH}_p[\mathsf{input}] \leftarrow y$
return $\mathsf{LH}_p[\mathsf{input}]$

$S_1(\mathsf{input})$

if input $\notin$ LH
  $y \leftarrow\$ \mathbb{Z}_q$
  LH[input] $\leftarrow y$
return LH[input]

OClientEnroll(ID)

$r_{\mathsf{ID}} \leftarrow\$ \mathbb{Z}_q^2$
$\mathsf{pw}_{\mathsf{ID}} \leftarrow\$ \mathcal{D}_{\mathsf{pw}}$
$\mathsf{salt}_{\mathsf{ID}} \leftarrow\$ \mathcal{D}_{\mathsf{salt}}$
abort if ID $\in$ enrolled
$\mathsf{seed}_{\mathsf{ID}} \leftarrow H_p(\mathsf{pw}_{\mathsf{ID}}, \mathsf{salt}_{\mathsf{ID}})$
$(\mathsf{sk}_{\mathsf{ID}}, \mathsf{pk}_{\mathsf{ID}}) \leftarrow \Sigma.\mathsf{Gen}(\mathsf{seed}_{\mathsf{ID}})$
$\mathsf{com}_{\mathsf{ID}} \leftarrow \mathsf{Com}(\mathsf{pk}_{\mathsf{ID}}; r_{\mathsf{ID}})$
$h_{\mathsf{ID}} \leftarrow \boxed{S_1}(\text{“seed:”}, \mathsf{seed}_{\mathsf{ID}})$
enrolled $\leftarrow$ enrolled $\cup \{\mathsf{ID}\}$
return $\mathsf{com}_{\mathsf{ID}}$

OClientLaunch(ID, att, [salt, h])

abort if ID $\notin$ enrolled
if (salt, h) not provided
  $(\mathsf{salt}, h) \leftarrow (\mathsf{salt}_{\mathsf{ID}}, h_{\mathsf{ID}})$
$\mathsf{seed} \leftarrow H_p(\mathsf{pw}_{\mathsf{ID}}, \mathsf{salt})$
return $\perp$ **if** $h \neq \boxed{S_1}(\text{“seed:”}, \mathsf{seed})$
$(\mathsf{sk}, \mathsf{pk}) \leftarrow \Sigma.\mathsf{Gen}(\mathsf{seed})$
$r \leftarrow\$ \mathbb{Z}_q$
$R \leftarrow r \cdot G$
$s \leftarrow r + \mathsf{sk} \cdot \boxed{S_1}(\text{“sig:”}, R, att)$
consented$\{\mathsf{ID}\} \leftarrow$ consented$\{\mathsf{ID}\} \cup \{att\}$
return $\perp$ **if** $s \cdot G \neq R + \boxed{S_1}(\text{“sig:”}, R, att) \cdot \mathsf{pk}_{\mathsf{ID}}$
$(e, r_1, r_2) \leftarrow \Pi.\mathsf{PoK}(\mathsf{com}_{\mathsf{ID}}, att, R, s, r_{\mathsf{ID}})$
return (R, e, r_1, r_2)

OAgentOrder(ID, att, R, s)

abort if ID $\notin$ enrolled
return $\perp$ **if** $s \cdot G \neq R + \boxed{S_1}(\text{“sig:”}, R, att) \cdot \mathsf{pk}_{\mathsf{ID}}$
$(e, r_1, r_2) \leftarrow \Pi.\mathsf{PoK}(\mathsf{com}_{\mathsf{ID}}, att, R, s, r_{\mathsf{ID}})$
return (R, e, r_1, r_2)

OAgentSalt(ID, pw)

abort if ID $\notin$ enrolled
return $\perp$ **if** pw $\neq \mathsf{pw}_{\mathsf{ID}}$
return $(\mathsf{salt}_{\mathsf{ID}}, h_{\mathsf{ID}})$

Figure 40: $\mathsf{G1}_{\mathsf{IdMless}}$ Game: no IdM and with an honest Agent

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

consented $\leftarrow \emptyset$ // genuinely consented att
enrolled $\leftarrow \emptyset$ // enrolled honest clients
$\mathsf{LH}, \mathsf{LH}_p \leftarrow \{\}$
$(\mathsf{ID}^*, att^*, R, e, r_1, r_2) \leftarrow \mathcal{A}^{\text{oracles but } S_2'}(\mathcal{D})$
return 0 **if** $\mathsf{ID}^* \notin$ enrolled
return 0 **if** $att^* \in$ consented$\{\mathsf{ID}^*\}$
return Verify$(\mathsf{com}_{\mathsf{ID}^*}, att^*, R, e, r_1, r_2)$

$H_p(\mathsf{input})$

if input $\notin \mathsf{LH}_p$
  $y \leftarrow\$ \{0,1\}^{2\lambda}$
  $\mathsf{LH}_p[\mathsf{input}] \leftarrow y$
return $\mathsf{LH}_p[\mathsf{input}]$

$S_1(\mathsf{input})$

if input $\notin \mathsf{LH}$
  $y \leftarrow\$ \mathbb{Z}_q$
  $\mathsf{LH}[\mathsf{input}] \leftarrow y$
return $\mathsf{LH}[\mathsf{input}]$

$S_2'(\mathsf{com}, att, R)$

$(C, e, r_1, r_2) \leftarrow \Pi.\mathsf{Sim}(\mathsf{com}, att, R)$
input $\leftarrow$ ("zk:", G, com, att, R, C)
abort if input $\in \mathsf{LH}$
$\mathsf{LH}[\mathsf{input}] \leftarrow e$
return (e, r_1, r_2)

OClientEnroll(ID)

$r_{\mathsf{ID}} \leftarrow\$ \mathbb{Z}_q^2$
$\mathsf{pw}_{\mathsf{ID}} \leftarrow\$ \mathcal{D}_{\mathsf{pw}}$
$\mathsf{salt}_{\mathsf{ID}} \leftarrow\$ \mathcal{D}_{\mathsf{salt}}$
abort if ID $\in$ enrolled
$\mathsf{seed}_{\mathsf{ID}} \leftarrow H_p(\mathsf{pw}_{\mathsf{ID}}, \mathsf{salt}_{\mathsf{ID}})$
$(\mathsf{sk}_{\mathsf{ID}}, \mathsf{pk}_{\mathsf{ID}}) \leftarrow \Sigma.\mathsf{Gen}(\mathsf{seed}_{\mathsf{ID}})$
$\mathsf{com}_{\mathsf{ID}} \leftarrow \mathsf{Com}(\mathsf{pk}_{\mathsf{ID}}; r_{\mathsf{ID}})$
$h_{\mathsf{ID}} \leftarrow S_1(\text{"seed:"}, \mathsf{seed}_{\mathsf{ID}})$
enrolled $\leftarrow$ enrolled $\cup \{\mathsf{ID}\}$
return $\mathsf{com}_{\mathsf{ID}}$

OClientLaunch(ID, att, [salt, h])

abort if ID $\notin$ enrolled
if (salt, h) not provided
  (salt, h) $\leftarrow (\mathsf{salt}_{\mathsf{ID}}, h_{\mathsf{ID}})$
seed $\leftarrow H_p(\mathsf{pw}_{\mathsf{ID}}, \mathsf{salt})$
return $\perp$ **if** $h \neq S_1(\text{"seed:"}, \mathsf{seed})$
$(\mathsf{sk}, \mathsf{pk}) \leftarrow \Sigma.\mathsf{Gen}(\mathsf{seed})$
$r \leftarrow\$ \mathbb{Z}_q$
$R \leftarrow r \cdot G$
$s \leftarrow r + \mathsf{sk} \cdot S_1(\text{"sig:"}, R, att)$
consented$\{\mathsf{ID}\} \leftarrow$ consented$\{\mathsf{ID}\} \cup \{att\}$
return $\perp$ **if** $s \cdot G \neq R + S_1(\text{"sig:"}, R, att) \cdot \mathsf{pk}_{\mathsf{ID}}$
$(e, r_1, r_2) \leftarrow$ $S_2'(\mathsf{com}_{\mathsf{ID}}, att, R)$
return (R, e, r_1, r_2)

OAgentOrder(ID, att, R, s)

abort if ID $\notin$ enrolled
return $\perp$ **if** $s \cdot G \neq R + S_1(\text{"sig:"}, R, att) \cdot \mathsf{pk}_{\mathsf{ID}}$
$(e, r_1, r_2) \leftarrow$ $S_2'(\mathsf{com}_{\mathsf{ID}}, att, R)$
return (R, e, r_1, r_2)

OAgentSalt(ID, pw)

abort if ID $\notin$ enrolled
return $\perp$ **if** pw $\neq \mathsf{pw}_{\mathsf{ID}}$
return $(\mathsf{salt}_{\mathsf{ID}}, h_{\mathsf{ID}})$

Figure 41: $\mathsf{G2}_{\mathsf{IdMless}}$ Game: no IdM and with an honest Agent

$\mathsf{G3}_{\mathsf{IdMless}}(\mathcal{A},\mathcal{D})$

consented $\leftarrow \emptyset$ // genuinely consented att
enrolled $\leftarrow \emptyset$ // enrolled honest clients
$\mathsf{LH}, \mathsf{LH}_p \leftarrow \{\}$
$x \leftarrow\$ \mathbb{G}$
$\mathsf{com}_0 \leftarrow \mathsf{Com}(x)$
$(\mathsf{ID}^*, att^*, R, e, r_1, r_2) \leftarrow \mathcal{A}^{\text{oracles but } S_2'}(\mathcal{D})$
return 0 **if** $\mathsf{ID}^* \notin$ enrolled
return 0 **if** $att^* \in$ consented$\{\mathsf{ID}^*\}$
return Verify$(\mathsf{com}_{\mathsf{ID}^*}, att^*, R, e, r_1, r_2)$

$H_p(\mathsf{input})$

if input $\notin \mathsf{LH}_p$
  $y \leftarrow\$ \{0,1\}^{2\lambda}$
  $\mathsf{LH}_p[\mathsf{input}] \leftarrow y$
return $\mathsf{LH}_p[\mathsf{input}]$

$S_1(\mathsf{input})$

if input $\notin \mathsf{LH}$
  $y \leftarrow\$ \mathbb{Z}_q$
  $\mathsf{LH}[\mathsf{input}] \leftarrow y$
return $\mathsf{LH}[\mathsf{input}]$

$S_2'(\mathsf{com}, att, R)$

$(C, e, r_1, r_2) \leftarrow \Pi.\mathsf{Sim}(\mathsf{com}, att, R)$
input $\leftarrow$ ("zk:", G, com, att, R, C)
abort if input $\in \mathsf{LH}$
$\mathsf{LH}[\mathsf{input}] \leftarrow e$
return (e, r_1, r_2)

OClientEnroll(ID)

$\mathsf{pw}_{\mathsf{ID}} \leftarrow\$ \mathcal{D}_{\mathsf{pw}}$
$\mathsf{salt}_{\mathsf{ID}} \leftarrow\$ \mathcal{D}_{\mathsf{salt}}$
abort if ID $\in$ enrolled
$\mathsf{seed}_{\mathsf{ID}} \leftarrow H_p(\mathsf{pw}_{\mathsf{ID}}, \mathsf{salt}_{\mathsf{ID}})$
$(\mathsf{sk}_{\mathsf{ID}}, \mathsf{pk}_{\mathsf{ID}}) \leftarrow \Sigma.\mathsf{Gen}(\mathsf{seed}_{\mathsf{ID}})$
$\mathsf{com}_{\mathsf{ID}} \leftarrow \mathsf{com}_0 + \mathsf{Com}(0)$
$h_{\mathsf{ID}} \leftarrow S_1(\text{"seed:"}, \mathsf{seed}_{\mathsf{ID}})$
enrolled $\leftarrow$ enrolled $\cup \{\mathsf{ID}\}$
return $\mathsf{com}_{\mathsf{ID}}$

OClientLaunch(ID, att, [salt, h])

abort if ID $\notin$ enrolled
if (salt, h) not provided
  (salt, h) $\leftarrow (\mathsf{salt}_{\mathsf{ID}}, h_{\mathsf{ID}})$
seed $\leftarrow H_p(\mathsf{pw}_{\mathsf{ID}}, \mathsf{salt})$
return $\perp$ **if** $h \neq S_1(\text{"seed:"}, \mathsf{seed})$
$(\mathsf{sk}, \mathsf{pk}) \leftarrow \Sigma.\mathsf{Gen}(\mathsf{seed})$
$r \leftarrow\$ \mathbb{Z}_q$
$R \leftarrow r \cdot G$
consented$\{\mathsf{ID}\} \leftarrow$ consented$\{\mathsf{ID}\} \cup \{att\}$
return $\perp$ **if** $\mathsf{pk} \neq \mathsf{pk}_{\mathsf{ID}}$
$(e, r_1, r_2) \leftarrow S_2'(\mathsf{com}_{\mathsf{ID}}, att, R)$
return (R, e, r_1, r_2)

OAgentOrder(ID, att, R, s)

abort if ID $\notin$ enrolled
return $\perp$ **if** $s \cdot G \neq R + S_1(\text{"sig:"}, R, att) \cdot \mathsf{pk}_{\mathsf{ID}}$
$(e, r_1, r_2) \leftarrow S_2'(\mathsf{com}_{\mathsf{ID}}, att, R)$
return (R, e, r_1, r_2)

OAgentSalt(ID, pw)

abort if ID $\notin$ enrolled
return $\perp$ **if** pw $\neq \mathsf{pw}_{\mathsf{ID}}$
return $(\mathsf{salt}_{\mathsf{ID}}, h_{\mathsf{ID}})$

Figure 42: $\mathsf{G3}_{\mathsf{IdMless}}$ Game: no IdM and with an honest Agent

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

1 : $\boxed{\mathsf{issued} \leftarrow \emptyset}$
2 : $\mathsf{consented} \leftarrow \emptyset$ // genuinely consented att
3 : $\mathsf{enrolled} \leftarrow \emptyset$ // enrolled honest clients
4 : $\mathsf{LH}, \mathsf{LH}_p \leftarrow \{\}$
5 : $x \leftarrow\$ \mathbb{G}$
6 : $\mathsf{com}_0 \leftarrow \mathsf{Com}(x)$
7 : $(\mathsf{ID}^*, att^*, R, e, r_1, r_2) \leftarrow \mathcal{A}^{\text{oracles but } S'_2}(\mathcal{D})$
8 : **return** 0 **if** $\mathsf{ID}^* \notin \mathsf{enrolled}$
9 : $\boxed{\textbf{return } 0 \textbf{ if } (\mathsf{com}_{\mathsf{ID}^*}, att, R, e, r_1, r_2) \notin \mathsf{issued}}$
10 : **return** 0 **if** $att^* \in \mathsf{consented}\{\mathsf{ID}^*\}$
11 : **return** $\mathsf{Verify}(\mathsf{com}_{\mathsf{ID}^*}, att^*, R, e, r_1, r_2)$

$H_p(\mathsf{input})$

1 : **if** $\mathsf{input} \notin \mathsf{LH}_p$
2 : $\quad y \leftarrow\$ \{0,1\}^{2\lambda}$
3 : $\quad \mathsf{LH}_p[\mathsf{input}] \leftarrow y$
4 : **return** $\mathsf{LH}_p[\mathsf{input}]$

$S_1(\mathsf{input})$

1 : **if** $\mathsf{input} \notin \mathsf{LH}$
2 : $\quad y \leftarrow\$ \mathbb{Z}_q$
3 : $\quad \mathsf{LH}[\mathsf{input}] \leftarrow y$
4 : **return** $\mathsf{LH}[\mathsf{input}]$

$S'_2(\mathsf{com}, att, R)$

1 : $(C, e, r_1, r_2) \leftarrow \Pi.\mathsf{Sim}(\mathsf{com}, att, R)$
2 : $\mathsf{input} \leftarrow (\text{“zk:”}, G, \mathsf{com}, att, R, C)$
3 : **abort if** $\mathsf{input} \in \mathsf{LH}$
4 : $\mathsf{LH}[\mathsf{input}] \leftarrow e$
5 : $\boxed{\mathsf{issued} \leftarrow \mathsf{issued} \cup \{(\mathsf{com}, att, R, e, r_1, r_2\}}$
6 : **return** (e, r_1, r_2)

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

1 : $\mathsf{pw}_{\mathsf{ID}} \leftarrow\$ \mathcal{D}_{\mathsf{pw}}$
2 : $\mathsf{salt}_{\mathsf{ID}} \leftarrow\$ \mathcal{D}_{\mathsf{salt}}$
3 : **abort if** $\mathsf{ID} \in \mathsf{enrolled}$
4 : $\mathsf{seed}_{\mathsf{ID}} \leftarrow H_p(\mathsf{pw}_{\mathsf{ID}}, \mathsf{salt}_{\mathsf{ID}})$
5 : $(\mathsf{sk}_{\mathsf{ID}}, \mathsf{pk}_{\mathsf{ID}}) \leftarrow \Sigma.\mathsf{Gen}(\mathsf{seed}_{\mathsf{ID}})$
6 : $\mathsf{com}_{\mathsf{ID}} \leftarrow \mathsf{com}_0 + \mathsf{Com}(0)$
7 : $h_{\mathsf{ID}} \leftarrow S_1(\text{“seed:”}, \mathsf{seed}_{\mathsf{ID}})$
8 : $\mathsf{enrolled} \leftarrow \mathsf{enrolled} \cup \{\mathsf{ID}\}$
9 : **return** $\mathsf{com}_{\mathsf{ID}}$

$\mathsf{OClientLaunch}(\mathsf{ID}, att, [\mathsf{salt}, h])$

1 : **abort if** $\mathsf{ID} \notin \mathsf{enrolled}$
2 : **if** (salt, h) not provided
3 : $\quad (\mathsf{salt}, h) \leftarrow (\mathsf{salt}_{\mathsf{ID}}, h_{\mathsf{ID}})$
4 : $\mathsf{seed} \leftarrow H_p(\mathsf{pw}_{\mathsf{ID}}, \mathsf{salt})$
5 : **return** $\perp$ **if** $h \neq S_1(\text{“seed:”}, \mathsf{seed})$
6 : $(\mathsf{sk}, \mathsf{pk}) \leftarrow \Sigma.\mathsf{Gen}(\mathsf{seed})$
7 : $r \leftarrow\$ \mathbb{Z}_q$
8 : $R \leftarrow r \cdot G$
9 : $\mathsf{consented}\{\mathsf{ID}\} \leftarrow \mathsf{consented}\{\mathsf{ID}\} \cup \{att\}$
10 : **return** $\perp$ **if** $\mathsf{pk} \neq \mathsf{pk}_{\mathsf{ID}}$
11 : $(e, r_1, r_2) \leftarrow S'_2(\mathsf{com}_{\mathsf{ID}}, att, R)$
12 : **return** (R, e, r_1, r_2)

$\mathsf{OAgentOrder}(\mathsf{ID}, att, R, s)$

1 : **abort if** $\mathsf{ID} \notin \mathsf{enrolled}$
2 : **return** $\perp$ **if** $s \cdot G \neq R + S_1(\text{“sig:”}, R, att) \cdot \mathsf{pk}_{\mathsf{ID}}$
3 : $(e, r_1, r_2) \leftarrow S'_2(\mathsf{com}_{\mathsf{ID}}, att, R)$
4 : **return** (R, e, r_1, r_2)

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

1 : **abort if** $\mathsf{ID} \notin \mathsf{enrolled}$
2 : **return** $\perp$ **if** $\mathsf{pw} \neq \mathsf{pw}_{\mathsf{ID}}$
3 : **return** $(\mathsf{salt}_{\mathsf{ID}}, h_{\mathsf{ID}})$

Figure 43: $\mathsf{G4}_{\mathsf{IdMless}}$ Game: no IdM and with an honest Agent

$\mathsf{G4.1}_{\mathsf{IdMless}}(\mathcal{A}_{4.1},\mathcal{D})$

1 : $\mathsf{issued} \leftarrow \emptyset$
2 : $\mathsf{enrolled} \leftarrow \emptyset$ // enrolled honest clients
3 : $\mathsf{LH} \leftarrow \{\}$
4 : $x \leftarrow\$ \mathbb{G}$
5 : $\mathsf{com}_0 \leftarrow \mathsf{Com}(x)$
6 : $\boxed{(\mathsf{ID}^*, att^*, R, e, r_1, r_2) \leftarrow \mathcal{A}_{4.1}^{\text{oracles}}(\mathcal{D})}$
7 : **return** 0 **if** $\mathsf{ID}^* \notin \mathsf{enrolled}$
8 : $\boxed{\textbf{return } 0 \textbf{ if } (\mathsf{com}_{\mathsf{ID}^*}, att, R, e, r_1, r_2) \in \mathsf{issued}}$
9 : **return** $\mathsf{Verify}(\mathsf{com}_{\mathsf{ID}^*}, att^*, R, e, r_1, r_2)$

$S_1(\mathsf{input})$

1 : **if** $\mathsf{input} \notin \mathsf{LH}$
2 : $\quad y \leftarrow\$ \mathbb{Z}_q$
3 : $\quad \mathsf{LH}[\mathsf{input}] \leftarrow y$
4 : **return** $\mathsf{LH}[\mathsf{input}]$

$S'_2(\mathsf{com}, att, R)$

1 : $(C, e, r_1, r_2) \leftarrow \Pi.\mathsf{Sim}(\mathsf{com}, att, R)$
2 : $\mathsf{input} \leftarrow (\text{“zk:”}, G, \mathsf{com}, att, R, C)$
3 : **abort if** $\mathsf{input} \in \mathsf{LH}$
4 : $\mathsf{LH}[\mathsf{input}] \leftarrow e$
5 : $\mathsf{issued} \leftarrow \mathsf{issued} \cup \{(\mathsf{com}, att, R, e, r_1, r_2\}$
6 : **return** (e, r_1, r_2)

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

1 : **abort if** $\mathsf{ID} \in \mathsf{enrolled}$
2 : $\mathsf{com}_{\mathsf{ID}} \leftarrow \mathsf{com}_0 + \mathsf{Com}(0)$
3 : $\mathsf{enrolled} \leftarrow \mathsf{enrolled} \cup \{\mathsf{ID}\}$
4 : **return** $\mathsf{com}_{\mathsf{ID}}$

Figure 44: $\mathsf{G4.1}_{\mathsf{IdMless}}$ Game: no IdM and with an honest Agent

$\mathsf{G4.2}_{\mathsf{IdMless}}(\mathcal{A}_{4.1}, \mathcal{E}, \mathcal{D})$

1 : $\mathsf{issued} \leftarrow \emptyset$
2 : $\mathsf{enrolled} \leftarrow \emptyset$ // enrolled honest clients
3 : $\mathsf{LH} \leftarrow \{\}$
4 : $x \leftarrow\$ \mathbb{G}$
5 : $\mathsf{com}_0 \leftarrow \mathsf{Com}(x)$
6 : pick ρ
7 : $(\mathsf{ID}^*, att^*, R, e, r_1, r_2) \leftarrow \mathcal{A}_{4.1}^{\text{oracles}}(\mathcal{D}; \rho)$
8 : $(s, r_{\mathsf{ID}}) \leftarrow \mathcal{E}(\rho, \mathsf{LH}, \mathsf{issued})$
9 : **return** 0 **if** $\mathsf{ID}^* \notin \mathsf{enrolled}$
10 : **return** 0 **if** $(\mathsf{com}_{\mathsf{ID}^*}, att, R, e, r_1, r_2) \in \mathsf{issued}$
11 : **return** 0 **if** $\mathsf{Com}(s \cdot G; r_{\mathsf{ID}}) \neq \mathsf{Com}(R; 0) + H(\text{“sig:”}, R, att^*) \cdot \mathsf{com}_{\mathsf{ID}^*}$
12 : **return** $\mathsf{Verify}(\mathsf{com}_{\mathsf{ID}^*}, att^*, R, e, r_1, r_2)$

$S_1(\mathsf{input})$

1 : **if** $\mathsf{input} \notin \mathsf{LH}$
2 : $y \leftarrow\$ \mathbb{Z}_q$
3 : $\mathsf{LH}[\mathsf{input}] \leftarrow y$
4 : **return** $\mathsf{LH}[\mathsf{input}]$

$S_2'(\mathsf{com}, att, R)$

1 : $(C, e, r_1, r_2) \leftarrow \Pi.\mathsf{Sim}(\mathsf{com}, att, R)$
2 : $\mathsf{input} \leftarrow (\text{“zk:”}, G, \mathsf{com}, att, R, C)$
3 : **abort if** $\mathsf{input} \in \mathsf{LH}$
4 : $\mathsf{LH}[\mathsf{input}] \leftarrow e$
5 : $\mathsf{issued} \leftarrow \mathsf{issued} \cup \{(\mathsf{com}, att, R, e, r_1, r_2\}$
6 : **return** (e, r_1, r_2)

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

1 : **abort if** $\mathsf{ID} \in \mathsf{enrolled}$
2 : $\mathsf{com}_{\mathsf{ID}} \leftarrow \mathsf{com}_0 + \mathsf{Com}(0)$
3 : $\mathsf{enrolled} \leftarrow \mathsf{enrolled} \cup \{\mathsf{ID}\}$
4 : **return** $\mathsf{com}_{\mathsf{ID}}$

Figure 45: $\mathsf{G4.2}_{\mathsf{IdMless}}$ Game: no IdM and with an honest Agent

$\mathsf{G4.3}_{\mathsf{IdMless}}(\mathcal{A}_{4.3}, \mathcal{D})$

1 : $\mathsf{enrolled} \leftarrow \emptyset$ // enrolled honest clients
2 : $x \leftarrow\$ \mathbb{G}$
3 : $\mathsf{com}_0 \leftarrow \mathsf{Com}(x)$
4 : $(\mathsf{ID}^*, \mathsf{pk}, r_{\mathsf{ID}}) \leftarrow \mathcal{A}_{4.3}^{\text{oracles}}(\mathcal{D})$
5 : **return** 0 **if** $\mathsf{ID}^* \notin \mathsf{enrolled}$
6 : **return** $\mathsf{Com}(\mathsf{pk}; r_{\mathsf{ID}}) = \mathsf{com}_{\mathsf{ID}^*}$

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

1 : **abort if** $\mathsf{ID} \in \mathsf{enrolled}$
2 : $\mathsf{com}_{\mathsf{ID}} \leftarrow \mathsf{com}_0 + \mathsf{Com}(0)$
3 : $\mathsf{enrolled} \leftarrow \mathsf{enrolled} \cup \{\mathsf{ID}\}$
4 : **return** $\mathsf{com}_{\mathsf{ID}}$

Figure 46: $\mathsf{G4.3}_{\mathsf{IdMless}}$ Game: no IdM and with an honest Agent

$\mathsf{G4.4}_{\mathsf{IdMless}}(\mathcal{A}_{4.4}, \mathcal{D})$

1 : $x \leftarrow\$ \mathbb{G}$
2 : $\mathsf{com}_0 \leftarrow \mathsf{Com}(x)$
3 : $(\mathsf{pk}, r_{\mathsf{ID}}) \leftarrow \mathcal{A}_{4.4}(\mathcal{D}, \mathsf{com}_0)$
4 : **return** $\mathsf{Com}(\mathsf{pk}; r_{\mathsf{ID}}) = \mathsf{com}_0$

Figure 47: $\mathsf{G4.4}_{\mathsf{IdMless}}$ Game: no IdM and with an honest Agent

$\mathsf{G5}_{\mathsf{IdMless}}$ **(Figure 48).** The final $(\mathsf{ID}^*, att^*, R, e, r_1, r_2)$ cannot have been made by the OClientLaunch oracle since it consented on att^* for ID^*. So we let OAgentOrder insert the forged consent in issued and we can drop the management of consented. We now require that H_p and H have no collision.

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{G4}_{\mathsf{IdMless}}} \leq \frac{q_{H_p}(q_{H_p}-1)}{2 \cdot 2^{2\lambda}} + \frac{q_H(q_H-1)}{2q} + \mathsf{Adv}_{\mathcal{A}}^{\mathsf{G5}_{\mathsf{IdMless}}}$$

$\mathsf{G6}_{\mathsf{IdMless}}$ **(Figure 49).** In OClientLaunch, when the provided salt is different from $\mathsf{salt}_{\mathsf{ID}}$, since there is no collision on H_p, we have $\mathsf{seed} \neq \mathsf{seed}_{\mathsf{ID}}$, and since we have no collision on H, we have $\mathsf{sk} \neq \mathsf{sk}_{\mathsf{ID}}$ therefore $\mathsf{pk} \neq \mathsf{pk}_{\mathsf{ID}}$, so the oracle returns $\perp$. If $\mathsf{salt} = \mathsf{salt}_{\mathsf{ID}}$ but $h \neq h_{\mathsf{ID}}$, it also returns $\perp$. Hence, we can simplify this oracle.

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

$\mathsf{G7}_{\mathsf{IdMless}}$ **and** $\mathsf{G7.1}_{\mathsf{IdMless}}$ **(Figure 50, Figure 51).** We add as a winning condition that no query to OAgentSalt was done with $\mathsf{pw} = \mathsf{pw}_{\mathsf{ID}}$. The failing event is modelled by the game $\mathsf{G7.1}_{\mathsf{IdMless}}$ which returns 1 only if the adversary makes a query with $\mathsf{pw}_{\mathsf{ID}}$. In this game, issued becomes useless.

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{G6}_{\mathsf{IdMless}}} \leq \mathsf{Adv}_{\mathcal{A}}^{\mathsf{G7}_{\mathsf{IdMless}}} + \mathsf{Adv}_{\mathcal{A}}^{\mathsf{G7.1}_{\mathsf{IdMless}}}$$

$\mathsf{G7.2}_{\mathsf{IdMless}}$ **(Figure 52).** We add as extra conditions that the adversary makes no query to OClientLaunch with $\mathsf{salt} = \mathsf{salt}_{\mathsf{ID}}$, no query to H_p with input of form $(.,\mathsf{salt}_{\mathsf{ID}})$ for any ID, no query to H with input $(\text{“sk:”}, \mathsf{seed}_{\mathsf{ID}})$ for any ID, and no query to OAgentOrder which does not return $\perp$. Each of these restriction means guessing either any $\mathsf{salt}_{\mathsf{ID}}$, $\mathsf{seed}_{\mathsf{ID}}$, or $\mathsf{pk}_{\mathsf{ID}}$. Since no information them leaks any longer, this failing event is bounded by $(q_{\mathsf{Launch}} + q_{\mathsf{Enroll}} q_{H_p}) 2^{-E_{\mathsf{salt}}} + q_{\mathsf{Enroll}} q_H 2^{-2\lambda} + \frac{q_{\mathsf{Order}}}{q}$. The OClientLaunch oracle further simplifies to the test to return $\perp$.

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{G7.1}_{\mathsf{IdMless}}} \leq \frac{q_{H_p} q_{\mathsf{Enroll}} + q_{\mathsf{Launch}}}{2^{E_{\mathsf{salt}}}} + \frac{q_{\mathsf{Enroll}} q_H}{2^{2\lambda}} + \frac{q_{\mathsf{Order}}}{q} + \mathsf{Adv}_{\mathcal{A}}^{\mathsf{G7.2}_{\mathsf{IdMless}}}$$

$\mathsf{G7.3}_{\mathsf{IdMless}}$ **(Figure 53).** We can now replace the generation of $\mathsf{seed}_{\mathsf{ID}}$ by a random selection without any S_1 query.

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{G7.2}_{\mathsf{IdMless}}} = \mathsf{Adv}_{\mathcal{A}}^{\mathsf{G7.3}_{\mathsf{IdMless}}}$$

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

issued $\leftarrow \emptyset$
enrolled $\leftarrow \emptyset$ // enrolled honest clients
$\mathsf{LH}, \mathsf{LH}_p \leftarrow \{\}$
$x \leftarrow\$ \mathbb{G}$
$\mathsf{com}_0 \leftarrow \mathsf{Com}(x)$
$(\mathsf{ID}^*, att^*, R, e, r_1, r_2) \leftarrow \mathcal{A}^{\text{oracles but } S'_2}(\mathcal{D})$
return 0 **if** $(\mathsf{com}_{\mathsf{ID}^*}, att, R, e, r_1, r_2) \notin$ issued
return 0 **if** $\mathsf{ID}^* \notin$ enrolled
return $\mathsf{Verify}(\mathsf{com}_{\mathsf{ID}^*}, att^*, R, e, r_1, r_2)$

$H_p(\mathsf{input})$

if input $\notin \mathsf{LH}_p$
  $y \leftarrow\$ \{0,1\}^{2\lambda}$
  $\boxed{\mathsf{G5}_{\mathsf{IdMless}}\ \textbf{abort if}\ \exists x\ \ \mathsf{LH}_p[x] = y}$
  $\mathsf{LH}_p[\mathsf{input}] \leftarrow y$
return $\mathsf{LH}_p[\mathsf{input}]$

$S_1(\mathsf{input})$

if input $\notin \mathsf{LH}$
  $y \leftarrow\$ \mathbb{Z}_q$
  $\boxed{\mathsf{G5}_{\mathsf{IdMless}}\ \textbf{abort if}\ \exists x\ \ \mathsf{LH}[x] = y}$
  $\mathsf{LH}[\mathsf{input}] \leftarrow y$
return $\mathsf{LH}[\mathsf{input}]$

$S'_2(\mathsf{com}, att, R)$

$(C, e, r_1, r_2) \leftarrow \Pi.\mathsf{Sim}(\mathsf{com}, att, R)$
$\mathsf{input} \leftarrow (\text{“zk:”}, G, \mathsf{com}, att, R, C)$
abort if input $\in \mathsf{LH}$
$\mathsf{LH}[\mathsf{input}] \leftarrow e$
return (e, r_1, r_2)

OClientEnroll(ID)

$\mathsf{pw}_{\mathsf{ID}} \leftarrow\$ \mathcal{D}_{\mathsf{pw}}$
$\mathsf{salt}_{\mathsf{ID}} \leftarrow\$ \mathcal{D}_{\mathsf{salt}}$
abort if $\mathsf{ID} \in$ enrolled
$\mathsf{seed}_{\mathsf{ID}} \leftarrow H_p(\mathsf{pw}_{\mathsf{ID}}, \mathsf{salt}_{\mathsf{ID}})$
$(\mathsf{sk}_{\mathsf{ID}}, \mathsf{pk}_{\mathsf{ID}}) \leftarrow \Sigma.\mathsf{Gen}(\mathsf{seed}_{\mathsf{ID}})$
$\mathsf{com}_{\mathsf{ID}} \leftarrow \mathsf{com}_0 + \mathsf{Com}(0)$
$h_{\mathsf{ID}} \leftarrow S_1(\text{“seed:”}, \mathsf{seed}_{\mathsf{ID}})$
enrolled $\leftarrow$ enrolled $\cup \{\mathsf{ID}\}$
return $\mathsf{com}_{\mathsf{ID}}$

OClientLaunch(ID, att, [salt, h])

abort if $\mathsf{ID} \notin$ enrolled
if (salt, h) not provided
  $(\mathsf{salt}, h) \leftarrow (\mathsf{salt}_{\mathsf{ID}}, h_{\mathsf{ID}})$
$\mathsf{seed} \leftarrow H_p(\mathsf{pw}_{\mathsf{ID}}, \mathsf{salt})$
return $\perp$ **if** $h \neq S_1(\text{“seed:”}, \mathsf{seed})$
$(\mathsf{sk}, \mathsf{pk}) \leftarrow \Sigma.\mathsf{Gen}(\mathsf{seed})$
$r \leftarrow\$ \mathbb{Z}_q$
$R \leftarrow r \cdot G$
return $\perp$ **if** $\mathsf{pk} \neq \mathsf{pk}_{\mathsf{ID}}$
$(e, r_1, r_2) \leftarrow S'_2(\mathsf{com}_{\mathsf{ID}}, att, R)$
return (R, e, r_1, r_2)

OAgentOrder(ID, att, R, s)

abort if $\mathsf{ID} \notin$ enrolled
return $\perp$ **if** $s \cdot G \neq R + S_1(\text{“sig:”}, R, att) \cdot \mathsf{pk}_{\mathsf{ID}}$
$(e, r_1, r_2) \leftarrow S'_2(\mathsf{com}_{\mathsf{ID}}, att, R)$
$\boxed{\mathsf{issued} \leftarrow \mathsf{issued} \cup \{(\mathsf{com}_{\mathsf{ID}}, att, R, e, r_1, r_2\}}$
return (R, e, r_1, r_2)

OAgentSalt(ID, pw)

abort if $\mathsf{ID} \notin$ enrolled
return $\perp$ **if** $\mathsf{pw} \neq \mathsf{pw}_{\mathsf{ID}}$
return $(\mathsf{salt}_{\mathsf{ID}}, h_{\mathsf{ID}})$

Figure 48: $\mathsf{G5}_{\mathsf{IdMless}}$ Game: no IdM and with an honest Agent

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

issued $\leftarrow \emptyset$
enrolled $\leftarrow \emptyset$ // enrolled honest clients
$\mathsf{LH}, \mathsf{LH}_p \leftarrow \{\}$
$x \leftarrow\$ \mathbb{G}$
$\mathsf{com}_0 \leftarrow \mathsf{Com}(x)$
$(\mathsf{ID}^*, att^*, R, e, r_1, r_2) \leftarrow \mathcal{A}^{\text{oracles but } S'_2}(\mathcal{D})$
return 0 **if** $(\mathsf{com}_{\mathsf{ID}^*}, att, R, e, r_1, r_2) \notin$ issued
return 0 **if** $\mathsf{ID}^* \notin$ enrolled
return $\mathsf{Verify}(\mathsf{com}_{\mathsf{ID}^*}, att^*, R, e, r_1, r_2)$

$H_p(\mathsf{input})$

if input $\notin \mathsf{LH}_p$
  $y \leftarrow\$ \{0,1\}^{2\lambda}$
  $\mathsf{G6}_{\mathsf{IdMless}}\ \textbf{abort if}\ \exists x\ \ \mathsf{LH}_p[x] = y$
  $\mathsf{LH}_p[\mathsf{input}] \leftarrow y$
return $\mathsf{LH}_p[\mathsf{input}]$

$S_1(\mathsf{input})$

if input $\notin \mathsf{LH}$
  $y \leftarrow\$ \mathbb{Z}_q$
  $\mathsf{G6}_{\mathsf{IdMless}}\ \textbf{abort if}\ \exists x\ \ \mathsf{LH}[x] = y$
  $\mathsf{LH}[\mathsf{input}] \leftarrow y$
return $\mathsf{LH}[\mathsf{input}]$

$S'_2(\mathsf{com}, att, R)$

$(C, e, r_1, r_2) \leftarrow \Pi.\mathsf{Sim}(\mathsf{com}, att, R)$
$\mathsf{input} \leftarrow (\text{“zk:”}, G, \mathsf{com}, att, R, C)$
abort if input $\in \mathsf{LH}$
$\mathsf{LH}[\mathsf{input}] \leftarrow e$
return (e, r_1, r_2)

OClientEnroll(ID)

$\mathsf{pw}_{\mathsf{ID}} \leftarrow\$ \mathcal{D}_{\mathsf{pw}}$
$\mathsf{salt}_{\mathsf{ID}} \leftarrow\$ \mathcal{D}_{\mathsf{salt}}$
abort if $\mathsf{ID} \in$ enrolled
$\mathsf{seed}_{\mathsf{ID}} \leftarrow H_p(\mathsf{pw}_{\mathsf{ID}}, \mathsf{salt}_{\mathsf{ID}})$
$(\mathsf{sk}_{\mathsf{ID}}, \mathsf{pk}_{\mathsf{ID}}) \leftarrow \Sigma.\mathsf{Gen}(\mathsf{seed}_{\mathsf{ID}})$
$\mathsf{com}_{\mathsf{ID}} \leftarrow \mathsf{com}_0 + \mathsf{Com}(0)$
$h_{\mathsf{ID}} \leftarrow S_1(\text{“seed:”}, \mathsf{seed}_{\mathsf{ID}})$
enrolled $\leftarrow$ enrolled $\cup \{\mathsf{ID}\}$
return $\mathsf{com}_{\mathsf{ID}}$

OClientLaunch(ID, att, [salt, h])

abort if $\mathsf{ID} \notin$ enrolled
$\boxed{\textbf{return}\ \perp\ \textbf{if}\ (\mathsf{salt}, h) \text{ provided and } (\mathsf{salt}, h) \neq (\mathsf{salt}_{\mathsf{ID}}, h_{\mathsf{ID}})}$
$r \leftarrow\$ \mathbb{Z}_q$
$R \leftarrow r \cdot G$
$(e, r_1, r_2) \leftarrow S'_2(\mathsf{com}_{\mathsf{ID}}, att, R)$
return (R, e, r_1, r_2)

OAgentOrder(ID, att, R, s)

abort if $\mathsf{ID} \notin$ enrolled
return $\perp$ **if** $s \cdot G \neq R + S_1(\text{“sig:”}, R, att) \cdot \mathsf{pk}_{\mathsf{ID}}$
$(e, r_1, r_2) \leftarrow S'_2(\mathsf{com}_{\mathsf{ID}}, att, R)$
issued $\leftarrow$ issued $\cup \{(\mathsf{com}_{\mathsf{ID}}, att, R, e, r_1, r_2\}$
return (R, e, r_1, r_2)

OAgentSalt(ID, pw)

abort if $\mathsf{ID} \notin$ enrolled
return $\perp$ **if** $\mathsf{pw} \neq \mathsf{pw}_{\mathsf{ID}}$
return $(\mathsf{salt}_{\mathsf{ID}}, h_{\mathsf{ID}})$

Figure 49: $\mathsf{G6}_{\mathsf{IdMless}}$ Game: no IdM and with an honest Agent

$\mathsf{G7}_{\mathsf{IdMless}}(\mathcal{A},\mathcal{D})$

1 : $\mathsf{issued} \leftarrow \emptyset$
2 : $\mathsf{enrolled} \leftarrow \emptyset$ // enrolled honest clients
3 : $\mathsf{LH}, \mathsf{LH}_p \leftarrow \{\}$
4 : $x \leftarrow\!\!\$\ \mathbb{G}$
5 : $\mathsf{com}_0 \leftarrow \mathsf{Com}(x)$
6 : $(\mathsf{ID}^*, att^*, R, e, r_1, r_2) \leftarrow \mathcal{A}^{\text{oracles but } S'_2}(\mathcal{D})$
7 : **return** 0 **if** $(\mathsf{com}_{\mathsf{ID}^*}, att, R, e, r_1, r_2) \notin \mathsf{issued}$
8 : **return** 0 **if** $\mathsf{ID}^* \notin \mathsf{enrolled}$
9 : **return** $\mathsf{Verify}(\mathsf{com}_{\mathsf{ID}^*}, att^*, R, e, r_1, r_2)$

$H_p(\mathsf{input})$

1 : **if** $\mathsf{input} \notin \mathsf{LH}_p$
2 : $\quad y \leftarrow\!\!\$\ \{0,1\}^{2\lambda}$
3 : $\quad \mathsf{G7}_{\mathsf{IdMless}}$ **abort if** $\exists x\ \mathsf{LH}_p[x] = y$
4 : $\quad \mathsf{LH}_p[\mathsf{input}] \leftarrow y$
5 : **return** $\mathsf{LH}_p[\mathsf{input}]$

$S_1(\mathsf{input})$

1 : **if** $\mathsf{input} \notin \mathsf{LH}$
2 : $\quad y \leftarrow\!\!\$\ \mathbb{Z}_q$
3 : $\quad \mathsf{G7}_{\mathsf{IdMless}}$ **abort if** $\exists x\ \mathsf{LH}[x] = y$
4 : $\quad \mathsf{LH}[\mathsf{input}] \leftarrow y$
5 : **return** $\mathsf{LH}[\mathsf{input}]$

$S'_2(\mathsf{com}, att, R)$

1 : $(C, e, r_1, r_2) \leftarrow \Pi.\mathsf{Sim}(\mathsf{com}, att, R)$
2 : $\mathsf{input} \leftarrow (\text{“zk:”}, G, \mathsf{com}, att, R, C)$
3 : **abort if** $\mathsf{input} \in \mathsf{LH}$
4 : $\mathsf{LH}[\mathsf{input}] \leftarrow e$
5 : **return** (e, r_1, r_2)

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

1 : $\mathsf{pw}_{\mathsf{ID}} \leftarrow\!\!\$\ \mathcal{D}_{\mathsf{pw}}$
2 : $\mathsf{salt}_{\mathsf{ID}} \leftarrow\!\!\$\ \mathcal{D}_{\mathsf{salt}}$
3 : **abort if** $\mathsf{ID} \in \mathsf{enrolled}$
4 : $\mathsf{seed}_{\mathsf{ID}} \leftarrow H_p(\mathsf{pw}_{\mathsf{ID}}, \mathsf{salt}_{\mathsf{ID}})$
5 : $(\mathsf{sk}_{\mathsf{ID}}, \mathsf{pk}_{\mathsf{ID}}) \leftarrow \Sigma.\mathsf{Gen}(\mathsf{seed}_{\mathsf{ID}})$
6 : $\mathsf{com}_{\mathsf{ID}} \leftarrow \mathsf{com}_0 + \mathsf{Com}(0)$
7 : $h_{\mathsf{ID}} \leftarrow S_1(\text{“seed:”}, \mathsf{seed}_{\mathsf{ID}})$
8 : $\mathsf{enrolled} \leftarrow \mathsf{enrolled} \cup \{\mathsf{ID}\}$
9 : **return** $\mathsf{com}_{\mathsf{ID}}$

$\mathsf{OClientLaunch}(\mathsf{ID}, att, [\mathsf{salt}, h])$

1 : **abort if** $\mathsf{ID} \notin \mathsf{enrolled}$
2 : **return** $\perp$ **if** (salt, h) provided and $(\mathsf{salt}, h) \neq (\mathsf{salt}_{\mathsf{ID}}, h_{\mathsf{ID}})$
3 : $r \leftarrow\!\!\$\ \mathbb{Z}_q$
4 : $R \leftarrow r \cdot G$
5 : $(e, r_1, r_2) \leftarrow S'_2(\mathsf{com}_{\mathsf{ID}}, att, R)$
6 : **return** (R, e, r_1, r_2)

$\mathsf{OAgentOrder}(\mathsf{ID}, att, R, s)$

1 : **abort if** $\mathsf{ID} \notin \mathsf{enrolled}$
2 : **return** $\perp$ **if** $s \cdot G \neq R + S_1(\text{“sig:”}, R, att) \cdot \mathsf{pk}_{\mathsf{ID}}$
3 : $(e, r_1, r_2) \leftarrow S'_2(\mathsf{com}_{\mathsf{ID}}, att, R)$
4 : $\mathsf{issued} \leftarrow \mathsf{issued} \cup \{(\mathsf{com}_{\mathsf{ID}}, att, R, e, r_1, r_2\}$
5 : **return** (R, e, r_1, r_2)

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

1 : **abort if** $\mathsf{ID} \notin \mathsf{enrolled}$
2 : **return** $\perp$ **if** $\mathsf{pw} \neq \mathsf{pw}_{\mathsf{ID}}$
3 : $\boxed{\mathsf{G7}_{\mathsf{IdMless}}\ \textbf{abort}}$

Figure 50: $\mathsf{G7}_{\mathsf{IdMless}}$ Game: no IdM and with an honest Agent

$\mathsf{G7.1}_{\mathsf{IdMless}}(\mathcal{A},\mathcal{D})$

1 : $\mathsf{enrolled} \leftarrow \emptyset$ // enrolled honest clients
2 : $\mathsf{LH}, \mathsf{LH}_p \leftarrow \{\}$
3 : $x \leftarrow\!\!\$\ \mathbb{G}$
4 : $\mathsf{com}_0 \leftarrow \mathsf{Com}(x)$
5 : $\mathcal{A}^{\text{oracles but } S'_2}(\mathcal{D})$
6 : **return** 0

$H_p(\mathsf{input})$

1 : **if** $\mathsf{input} \notin \mathsf{LH}_p$
2 : $\quad y \leftarrow\!\!\$\ \{0,1\}^{2\lambda}$
3 : $\quad \mathsf{G7.1}_{\mathsf{IdMless}}$ **abort if** $\exists x\ \mathsf{LH}_p[x] = y$
4 : $\quad \mathsf{LH}_p[\mathsf{input}] \leftarrow y$
5 : **return** $\mathsf{LH}_p[\mathsf{input}]$

$S_1(\mathsf{input})$

1 : **if** $\mathsf{input} \notin \mathsf{LH}$
2 : $\quad y \leftarrow\!\!\$\ \mathbb{Z}_q$
3 : $\quad \mathsf{G7.1}_{\mathsf{IdMless}}$ **abort if** $\exists x\ \mathsf{LH}[x] = y$
4 : $\quad \mathsf{LH}[\mathsf{input}] \leftarrow y$
5 : **return** $\mathsf{LH}[\mathsf{input}]$

$S'_2(\mathsf{com}, att, R)$

1 : $(C, e, r_1, r_2) \leftarrow \Pi.\mathsf{Sim}(\mathsf{com}, att, R)$
2 : $\mathsf{input} \leftarrow (\text{“zk:”}, G, \mathsf{com}, att, R, C)$
3 : **abort if** $\mathsf{input} \in \mathsf{LH}$
4 : $\mathsf{LH}[\mathsf{input}] \leftarrow e$
5 : **return** (e, r_1, r_2)

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

1 : $\mathsf{pw}_{\mathsf{ID}} \leftarrow\!\!\$\ \mathcal{D}_{\mathsf{pw}}$
2 : $\mathsf{salt}_{\mathsf{ID}} \leftarrow\!\!\$\ \mathcal{D}_{\mathsf{salt}}$
3 : **abort if** $\mathsf{ID} \in \mathsf{enrolled}$
4 : $\mathsf{seed}_{\mathsf{ID}} \leftarrow H_p(\mathsf{pw}_{\mathsf{ID}}, \mathsf{salt}_{\mathsf{ID}})$
5 : $(\mathsf{sk}_{\mathsf{ID}}, \mathsf{pk}_{\mathsf{ID}}) \leftarrow \Sigma.\mathsf{Gen}(\mathsf{seed}_{\mathsf{ID}})$
6 : $\mathsf{com}_{\mathsf{ID}} \leftarrow \mathsf{com}_0 + \mathsf{Com}(0)$
7 : $h_{\mathsf{ID}} \leftarrow S_1(\text{“seed:”}, \mathsf{seed}_{\mathsf{ID}})$
8 : $\mathsf{enrolled} \leftarrow \mathsf{enrolled} \cup \{\mathsf{ID}\}$
9 : **return** $\mathsf{com}_{\mathsf{ID}}$

$\mathsf{OClientLaunch}(\mathsf{ID}, att, [\mathsf{salt}, h])$

1 : **abort if** $\mathsf{ID} \notin \mathsf{enrolled}$
2 : **return** $\perp$ **if** (salt, h) provided and $(\mathsf{salt}, h) \neq (\mathsf{salt}_{\mathsf{ID}}, h_{\mathsf{ID}})$
3 : $r \leftarrow\!\!\$\ \mathbb{Z}_q$
4 : $R \leftarrow r \cdot G$
5 : $(e, r_1, r_2) \leftarrow S'_2(\mathsf{com}_{\mathsf{ID}}, att, R)$
6 : **return** (R, e, r_1, r_2)

$\mathsf{OAgentOrder}(\mathsf{ID}, att, R, s)$

1 : **abort if** $\mathsf{ID} \notin \mathsf{enrolled}$
2 : **return** $\perp$ **if** $s \cdot G \neq R + S_1(\text{“sig:”}, R, att) \cdot \mathsf{pk}_{\mathsf{ID}}$
3 : $(e, r_1, r_2) \leftarrow S'_2(\mathsf{com}_{\mathsf{ID}}, att, R)$
4 : **return** (R, e, r_1, r_2)

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

1 : **abort if** $\mathsf{ID} \notin \mathsf{enrolled}$
2 : **return** $\perp$ **if** $\mathsf{pw} \neq \mathsf{pw}_{\mathsf{ID}}$
3 : $\boxed{\mathsf{G7.1}_{\mathsf{IdMless}}\ \textbf{return}\ 1}$

Figure 51: $\mathsf{G7.1}_{\mathsf{IdMless}}$ Game: no IdM and with an honest Agent

$\mathsf{G7.2}_{\mathsf{IdMless}}(\mathcal{A},\mathcal{D})$

enrolled $\leftarrow \emptyset$ // enrolled honest clients
$x \leftarrow\$ \mathbb{G}$
$\mathsf{com}_0 \leftarrow \mathsf{Com}(x)$
$\mathsf{LH}, \mathsf{LH}_p \leftarrow \{\}$
$\mathcal{A}^{\text{oracles but } S'_2}(\mathcal{D})$
return 0

$H_p(\mathsf{input})$

if input $\notin \mathsf{LH}_p$
  $y \leftarrow\$ \{0,1\}^{2\lambda}$
  $\mathsf{G7.2}_{\mathsf{IdMless}}$ **abort if** $\exists x\ \mathsf{LH}_p[x] = y$
  $\boxed{\mathsf{G7.2}_{\mathsf{IdMless}}\ \textbf{abort if}\ \exists \mathsf{ID} \in \text{enrolled}\ x = (.,\mathsf{salt}_{\mathsf{ID}})}$
  $\mathsf{LH}_p[\text{input}] \leftarrow y$
return $\mathsf{LH}_p[\text{input}]$

$S_1(\mathsf{input})$

if input $\notin \mathsf{LH}$
  $y \leftarrow\$ \mathbb{Z}_q$
  $\mathsf{G7.2}_{\mathsf{IdMless}}$ **abort if** $\exists x\ \mathsf{LH}[x] = y$
  $\boxed{\mathsf{G7.2}_{\mathsf{IdMless}}\ \textbf{abort if}\ \exists \mathsf{ID} \in \text{enrolled}\ x = (\text{“sk:”},\mathsf{seed}_{\mathsf{ID}})}$
  $\mathsf{LH}[\text{input}] \leftarrow y$
return $\mathsf{LH}[\text{input}]$

$S'_2(\mathsf{com}, att, R)$

$(C, e, r_1, r_2) \leftarrow \Pi.\mathsf{Sim}(\mathsf{com}, att, R)$
$\mathsf{input} \leftarrow (\text{“zk:”}, G, \mathsf{com}, att, R, C)$
abort if input $\in \mathsf{LH}$
$\mathsf{LH}[\text{input}] \leftarrow e$
return (e, r_1, r_2)

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

$\mathsf{pw}_{\mathsf{ID}} \leftarrow\$ \mathcal{D}_{\mathsf{pw}}$
$\mathsf{salt}_{\mathsf{ID}} \leftarrow\$ \mathcal{D}_{\mathsf{salt}}$
abort if ID $\in$ enrolled
$\mathsf{seed}_{\mathsf{ID}} \leftarrow H_p(\mathsf{pw}_{\mathsf{ID}}, \mathsf{salt}_{\mathsf{ID}})$
$(\mathsf{sk}_{\mathsf{ID}}, \mathsf{pk}_{\mathsf{ID}}) \leftarrow \Sigma.\mathsf{Gen}(\mathsf{seed}_{\mathsf{ID}})$
$\mathsf{com}_{\mathsf{ID}} \leftarrow \mathsf{com}_0 + \mathsf{Com}(0)$
$h_{\mathsf{ID}} \leftarrow S_1(\text{“seed:”}, \mathsf{seed}_{\mathsf{ID}})$
enrolled $\leftarrow$ enrolled $\cup \{\mathsf{ID}\}$
return $\mathsf{com}_{\mathsf{ID}}$

$\mathsf{OClientLaunch}(\mathsf{ID}, att, [\mathsf{salt}, h])$

abort if ID $\notin$ enrolled
$\boxed{\mathsf{G7.2}_{\mathsf{IdMless}}\ \textbf{abort if}\ \text{salt provided and salt} = \mathsf{salt}_{\mathsf{ID}}}$
return $\perp$ $\boxed{\textbf{if}\ (\mathsf{salt}, h)\ \text{provided}}$
$r \leftarrow\$ \mathbb{Z}_q$
$R \leftarrow r \cdot G$
$(e, r_1, r_2) \leftarrow S'_2(\mathsf{com}_{\mathsf{ID}}, att, R)$
return (R, e, r_1, r_2)

$\mathsf{OAgentOrder}(\mathsf{ID}, att, R, s)$

abort if ID $\notin$ enrolled
return $\perp$ **if** $s \cdot G \neq R + S_1(\text{“sig:”}, R, att) \cdot \mathsf{pk}_{\mathsf{ID}}$
$\boxed{\mathsf{G7.2}_{\mathsf{IdMless}}\ \textbf{return}\ 0}$

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

abort if ID $\notin$ enrolled
return $\perp$ **if** $\mathsf{pw} \neq \mathsf{pw}_{\mathsf{ID}}$
$\mathsf{G7.2}_{\mathsf{IdMless}}$ **return** 1

Figure 52: $\mathsf{G7.2}_{\mathsf{IdMless}}$ Game: no IdM and with an honest Agent

$\mathsf{G7.3}_{\mathsf{IdMless}}(\mathcal{A},\mathcal{D})$

enrolled $\leftarrow \emptyset$ // enrolled honest clients
$\mathsf{LH}, \mathsf{LH}_p \leftarrow \{\}$
$x \leftarrow\$ \mathbb{G}$
$\mathsf{com}_0 \leftarrow \mathsf{Com}(x)$
$\mathcal{A}^{\text{oracles but } S'_2}(\mathcal{D})$
return 0

$H_p(\mathsf{input})$

if input $\notin \mathsf{LH}_p$
  $y \leftarrow\$ \{0,1\}^{2\lambda}$
  $\mathsf{G7.3}_{\mathsf{IdMless}}$ **abort if** $\exists x\ \mathsf{LH}_p[x] = y$
  $\mathsf{G7.3}_{\mathsf{IdMless}}\ \textbf{abort if}\ \exists \mathsf{ID} \in \text{enrolled}\ x = (.,\mathsf{salt}_{\mathsf{ID}})$
  $\mathsf{LH}_p[\text{input}] \leftarrow y$
return $\mathsf{LH}_p[\text{input}]$

$S_1(\mathsf{input})$

if input $\notin \mathsf{LH}$
  $y \leftarrow\$ \mathbb{Z}_q$
  $\mathsf{G7.3}_{\mathsf{IdMless}}$ **abort if** $\exists x\ \mathsf{LH}[x] = y$
  $\mathsf{G7.3}_{\mathsf{IdMless}}\ \textbf{abort if}\ \exists \mathsf{ID} \in \text{enrolled}\ x = (\text{“sk:”},\mathsf{seed}_{\mathsf{ID}})$
  $\mathsf{LH}[\text{input}] \leftarrow y$
return $\mathsf{LH}[\text{input}]$

$S'_2(\mathsf{com}, att, R)$

$(C, e, r_1, r_2) \leftarrow \Pi.\mathsf{Sim}(\mathsf{com}, att, R)$
$\mathsf{input} \leftarrow (\text{“zk:”}, G, \mathsf{com}, att, R, C)$
abort if input $\in \mathsf{LH}$
$\mathsf{LH}[\text{input}] \leftarrow e$
return (e, r_1, r_2)

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

$\mathsf{pw}_{\mathsf{ID}} \leftarrow\$ \mathcal{D}_{\mathsf{pw}}$
$\mathsf{salt}_{\mathsf{ID}} \leftarrow\$ \mathcal{D}_{\mathsf{salt}}$
abort if ID $\in$ enrolled
$\boxed{\mathsf{seed}_{\mathsf{ID}} \leftarrow\$ \{0,1\}^{2\lambda}}$
$(\mathsf{sk}_{\mathsf{ID}}, \mathsf{pk}_{\mathsf{ID}}) \leftarrow \Sigma.\mathsf{Gen}(\mathsf{seed}_{\mathsf{ID}})$
$\mathsf{com}_{\mathsf{ID}} \leftarrow \mathsf{com}_0 + \mathsf{Com}(0)$
$h_{\mathsf{ID}} \leftarrow S_1(\text{“seed:”}, \mathsf{seed}_{\mathsf{ID}})$
enrolled $\leftarrow$ enrolled $\cup \{\mathsf{ID}\}$
return $\mathsf{com}_{\mathsf{ID}}$

$\mathsf{OClientLaunch}(\mathsf{ID}, att, [\mathsf{salt}, h])$

abort if ID $\notin$ enrolled
$\mathsf{G7.3}_{\mathsf{IdMless}}$ **abort if** salt provided and salt $= \mathsf{salt}_{\mathsf{ID}}$
return $\perp$ **if** (salt, h) provided
$r \leftarrow\$ \mathbb{Z}_q$
$R \leftarrow r \cdot G$
$(e, r_1, r_2) \leftarrow S'_2(\mathsf{com}_{\mathsf{ID}}, att, R)$
return (R, e, r_1, r_2)

$\mathsf{OAgentOrder}(\mathsf{ID}, att, R, s)$

abort if ID $\notin$ enrolled
return $\perp$ **if** $s \cdot G \neq R + S_1(\text{“sig:”}, R, att) \cdot \mathsf{pk}_{\mathsf{ID}}$
$\mathsf{G7.3}_{\mathsf{IdMless}}$ **return** 0

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

abort if ID $\notin$ enrolled
return $\perp$ **if** $\mathsf{pw} \neq \mathsf{pw}_{\mathsf{ID}}$
$\mathsf{G7.3}_{\mathsf{IdMless}}$ **return** 1

Figure 53: $\mathsf{G7.3}_{\mathsf{IdMless}}$ Game: no IdM and with an honest Agent

Wrapping up G7.3**.** $\mathsf{pw}_{\mathsf{ID}}$ is no longer used, except in the test oracle OAgentSalt. So we can bound by the success probability of an online dictionary attack.

$$\mathsf{Adv}^{\mathsf{G7.3}_{\mathsf{IdMless}}}_{\mathcal{A}} \leq \frac{q_{\mathsf{Salt}}}{2^{E_{\mathsf{pw}}}}$$

$\mathsf{G8}_{\mathsf{IdMless}}$ **(Figure 54).** We add as an extra condition that the adversary makes no query to OClientLaunch with $\mathsf{salt} = \mathsf{salt}_{\mathsf{ID}}$ and no query to H_p with input of form $(.,\mathsf{salt}_{\mathsf{ID}})$ for any ID. Since no information about $\mathsf{salt}_{\mathsf{ID}}$ leaks any longer besides this test, this failing event is bounded by $(q_{\mathsf{Launch}} + q_{\mathsf{Enroll}} q_{H_p}) 2^{-E_{\mathsf{salt}}}$. The OClientLaunch oracle further simplifies to the test to return $\perp$.

$$\mathsf{Adv}^{\mathsf{G7}_{\mathsf{IdMless}}}_{\mathcal{A}} \leq \frac{q_{H_p} q_{\mathsf{Enroll}} + q_{\mathsf{Launch}}}{2^{E_{\mathsf{salt}}}} + \mathsf{Adv}^{\mathsf{G8}_{\mathsf{IdMless}}}_{\mathcal{A}}$$

$\mathsf{G9}_{\mathsf{IdMless}}$ **(Figure 55).** We can now generate $\mathsf{seed}_{\mathsf{ID}}$ without making a query to H_p. We remove the generation of the h_{ID} which is no longer used. We add as an extra condition that the adversary makes no query to H with input of form $(\text{“sk:”}, \mathsf{seed}_{\mathsf{ID}})$ for any ID. Since no information about $\mathsf{seed}_{\mathsf{ID}}$ leaks any longer besides this test, this failing event is bounded by $q_{\mathsf{Enroll}} q_H 2^{-2\lambda}$. The key generation in OClientEnroll further simplifies to the random selection of $\mathsf{pk}_{\mathsf{ID}}$.

$$\mathsf{Adv}^{\mathsf{G8}_{\mathsf{IdMless}}}_{\mathcal{A}} \leq \frac{q_H q_{\mathsf{Enroll}}}{2^{2\lambda}} + \mathsf{Adv}^{\mathsf{G9}_{\mathsf{IdMless}}}_{\mathcal{A}}$$

Wrapping up. Now that $\mathsf{pk}_{\mathsf{ID}}$ is freshly generated and only used in OAgentOrder, the probability to pass the test in this oracle is bounded by $\frac{q_{\mathsf{Order}}}{q}$. When this does not happen, the set issued remains empty and the game does not return 1.

$$\mathsf{Adv}^{\mathsf{G9}_{\mathsf{IdMless}}}_{\mathcal{A}} \leq \frac{q_{\mathsf{Order}}}{q}$$

D PBS based on blind RSA

In this section, we explain the PBS based on RSA from [Sangeetha Jose and Rangan(2013)]. 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.

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.

$\mathsf{G8}_{\mathsf{IdMless}}(\mathcal{A}, \mathcal{D})$

issued $\leftarrow \emptyset$
enrolled $\leftarrow \emptyset$ // enrolled honest clients
$\mathsf{LH}, \mathsf{LH}_p \leftarrow \{\}$
$x \leftarrow\$ \mathbb{G}$
$\mathsf{com}_0 \leftarrow \mathsf{Com}(x)$
$(\mathsf{ID}^*, att^*, R, e, r_1, r_2) \leftarrow \mathcal{A}^{\text{oracles but } S'_2}(\mathcal{D})$
return 0 **if** $(\mathsf{com}_{\mathsf{ID}^*}, att, R, e, r_1, r_2) \notin$ issued
return 0 **if** $\mathsf{ID}^* \notin$ enrolled
return $\mathsf{Verify}(\mathsf{com}_{\mathsf{ID}^*}, att^*, R, e, r_1, r_2)$

$H_p(\mathsf{input})$

if input $\notin \mathsf{LH}_p$
  $y \leftarrow\$ \{0,1\}^{2\lambda}$
  $\mathsf{G8}_{\mathsf{IdMless}}$ **abort if** $\exists x$ $\mathsf{LH}_p[x] = y$
  $\boxed{\mathsf{G8}_{\mathsf{IdMless}} \textbf{ abort if } \exists \mathsf{ID} \in \text{enrolled } x = (.,\mathsf{salt}_{\mathsf{ID}})}$
  $\mathsf{LH}_p[\mathsf{input}] \leftarrow y$
return $\mathsf{LH}_p[\mathsf{input}]$

$S_1(\mathsf{input})$

if input $\notin \mathsf{LH}$
  $y \leftarrow\$ \mathbb{Z}_q$
  $\mathsf{G8}_{\mathsf{IdMless}}$ **abort if** $\exists x$ $\mathsf{LH}[x] = y$
  $\mathsf{LH}[\mathsf{input}] \leftarrow y$
return $\mathsf{LH}[\mathsf{input}]$

$S'_2(\mathsf{com}, att, R)$

$(C, e, r_1, r_2) \leftarrow \Pi.\mathsf{Sim}(\mathsf{com}, att, R)$
$\mathsf{input} \leftarrow (\text{“zk:”}, G, \mathsf{com}, att, R, C)$
abort if input $\in \mathsf{LH}$
$\mathsf{LH}[\mathsf{input}] \leftarrow e$
return (e, r_1, r_2)

OClientEnroll(ID)

$\mathsf{pw}_{\mathsf{ID}} \leftarrow\$ \mathcal{D}_{\mathsf{pw}}$
$\mathsf{salt}_{\mathsf{ID}} \leftarrow\$ \mathcal{D}_{\mathsf{salt}}$
abort if ID $\in$ enrolled
$\mathsf{seed}_{\mathsf{ID}} \leftarrow H_p(\mathsf{pw}_{\mathsf{ID}}, \mathsf{salt}_{\mathsf{ID}})$
$(\mathsf{sk}_{\mathsf{ID}}, \mathsf{pk}_{\mathsf{ID}}) \leftarrow \Sigma.\mathsf{Gen}(\mathsf{seed}_{\mathsf{ID}})$
$\mathsf{com}_{\mathsf{ID}} \leftarrow \mathsf{com}_0 + \mathsf{Com}(0)$
$h_{\mathsf{ID}} \leftarrow S_1(\text{“seed:”}, \mathsf{seed}_{\mathsf{ID}})$
enrolled $\leftarrow$ enrolled $\cup \{\mathsf{ID}\}$
return $\mathsf{com}_{\mathsf{ID}}$

OClientLaunch(ID, att, [salt, h])

abort if ID $\notin$ enrolled
$\boxed{\mathsf{G8}_{\mathsf{IdMless}} \textbf{ abort if } \mathsf{salt} \text{ provided and } \mathsf{salt} = \mathsf{salt}_{\mathsf{ID}}}$
return $\perp$ $\boxed{\textbf{if } (\mathsf{salt}, h) \text{ provided}}$
$r \leftarrow\$ \mathbb{Z}_q$
$R \leftarrow r \cdot G$
$(e, r_1, r_2) \leftarrow S'_2(\mathsf{com}_{\mathsf{ID}}, att, R)$
return (R, e, r_1, r_2)

OAgentOrder(ID, att, R, s)

abort if ID $\notin$ enrolled
return $\perp$ **if** $s \cdot G \neq R + S_1(\text{“sig:”}, R, att) \cdot \mathsf{pk}_{\mathsf{ID}}$
$(e, r_1, r_2) \leftarrow S'_2(\mathsf{com}_{\mathsf{ID}}, att, R)$
issued $\leftarrow$ issued $\cup \{(\mathsf{com}_{\mathsf{ID}}, att, R, e, r_1, r_2\}$
return (R, e, r_1, r_2)

OAgentSalt(ID, pw)

abort if ID $\notin$ enrolled
return $\perp$ **if** $\mathsf{pw} \neq \mathsf{pw}_{\mathsf{ID}}$
$\mathsf{G8}_{\mathsf{IdMless}}$ **abort**

Figure 54: $\mathsf{G8}_{\mathsf{IdMless}}$ Game: no IdM and with an honest Agent

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

issued $\leftarrow \emptyset$
enrolled $\leftarrow \emptyset$ // enrolled honest clients
$\mathsf{LH}, \mathsf{LH}_p \leftarrow \{\}$
$x \leftarrow\$ \mathbb{G}$
$\mathsf{com}_0 \leftarrow \mathsf{Com}(x)$
$(\mathsf{ID}^*, att^*, R, e, r_1, r_2) \leftarrow \mathcal{A}^{\text{oracles but } S'_2(\mathcal{D})}$
return 0 **if** $(\mathsf{com}_{\mathsf{ID}^*}, att, R, e, r_1, r_2) \notin$ issued
return 0 **if** $\mathsf{ID}^* \notin$ enrolled
return $\mathsf{Verify}(\mathsf{com}_{\mathsf{ID}^*}, att^*, R, e, r_1, r_2)$

$H_p(\mathsf{input})$

if input $\notin \mathsf{LH}_p$
  $y \leftarrow\$ \{0,1\}^{2\lambda}$
  $\mathsf{G9}_{\mathsf{IdMless}}$ **abort if** $\exists x\ \mathsf{LH}_p[x] = y$
  $\mathsf{G9}_{\mathsf{IdMless}}$ **abort if** $\exists \mathsf{ID} \in$ enrolled $x = (.,\mathsf{salt}_{\mathsf{ID}})$
  $\mathsf{LH}_p[\mathsf{input}] \leftarrow y$
return $\mathsf{LH}_p[\mathsf{input}]$

$S_1(\mathsf{input})$

if input $\notin \mathsf{LH}$
  $y \leftarrow\$ \mathbb{Z}_q$
  $\mathsf{G9}_{\mathsf{IdMless}}$ **abort if** $\exists x\ \mathsf{LH}[x] = y$
  $\mathsf{G9}_{\mathsf{IdMless}}$ **abort if** $\exists \mathsf{ID} \in$ enrolled $x = (\text{“sk:”}, \mathsf{seed}_{\mathsf{ID}})$
  $\mathsf{LH}[\mathsf{input}] \leftarrow y$
return $\mathsf{LH}[\mathsf{input}]$

$S'_2(\mathsf{com}, att, R)$

$(C, e, r_1, r_2) \leftarrow \Pi.\mathsf{Sim}(\mathsf{com}, att, R)$
input $\leftarrow$ (“zk:”, $G, \mathsf{com}, att, R, C$)
abort if input $\in \mathsf{LH}$
$\mathsf{LH}[\mathsf{input}] \leftarrow e$
return (e, r_1, r_2)

OClientEnroll(ID)

$\mathsf{pw}_{\mathsf{ID}} \leftarrow\$ \mathcal{D}_{\mathsf{pw}}$
$\mathsf{salt}_{\mathsf{ID}} \leftarrow\$ \mathcal{D}_{\mathsf{salt}}$
abort if ID $\in$ enrolled
$\mathsf{seed}_{\mathsf{ID}} \leftarrow\$ \{0,1\}^{2\lambda}$
$\mathsf{pk}_{\mathsf{ID}} \leftarrow\$ \mathbb{G}$
$\mathsf{com}_{\mathsf{ID}} \leftarrow \mathsf{com}_0 + \mathsf{Com}(0)$
enrolled $\leftarrow$ enrolled $\cup \{\mathsf{ID}\}$
return $\mathsf{com}_{\mathsf{ID}}$

OClientLaunch(ID, att, [salt, h])

abort if ID $\notin$ enrolled
$\mathsf{G9}_{\mathsf{IdMless}}$ **abort if** salt provided and salt $= \mathsf{salt}_{\mathsf{ID}}$
return $\perp$ **if** (salt, h) provided
$r \leftarrow\$ \mathbb{Z}_q$
$R \leftarrow r \cdot G$
$(e, r_1, r_2) \leftarrow S'_2(\mathsf{com}_{\mathsf{ID}}, att, R)$
return (R, e, r_1, r_2)

OAgentOrder(ID, att, R, s)

abort if ID $\notin$ enrolled
return $\perp$ **if** $s \cdot G \neq R + S_1(\text{“sig:”}, R, att) \cdot \mathsf{pk}_{\mathsf{ID}}$
$(e, r_1, r_2) \leftarrow S'_2(\mathsf{com}_{\mathsf{ID}}, att, R)$
issued $\leftarrow$ issued $\cup \{(\mathsf{com}_{\mathsf{ID}}, att, R, e, r_1, r_2\}$
return (R, e, r_1, r_2)

OAgentSalt(ID, pw)

abort if ID $\notin$ enrolled
return $\perp$ **if** pw $\neq \mathsf{pw}_{\mathsf{ID}}$
$\mathsf{G8}_{\mathsf{IdMless}}$ **abort**

Figure 55: $\mathsf{G9}_{\mathsf{IdMless}}$ Game: no IdM and with an honest Agent

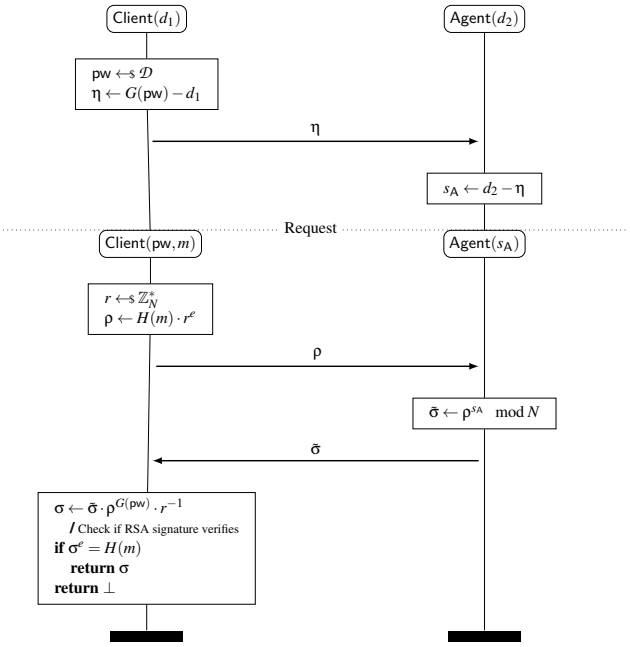

Figure 56: Password Based Signature Based on RSA

E PBS based on CL Signatures

In [Sangeetha Jose and Rangan(2013)], the authors also propose a PBS protocol based on CL Signatures [Camenisch and Lysyanskaya(2001)]. 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. 57 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.

F Server Aided Digital Signatures (SADS)

Notation for the SADS protocol [He et al.(2005)] below:

- 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

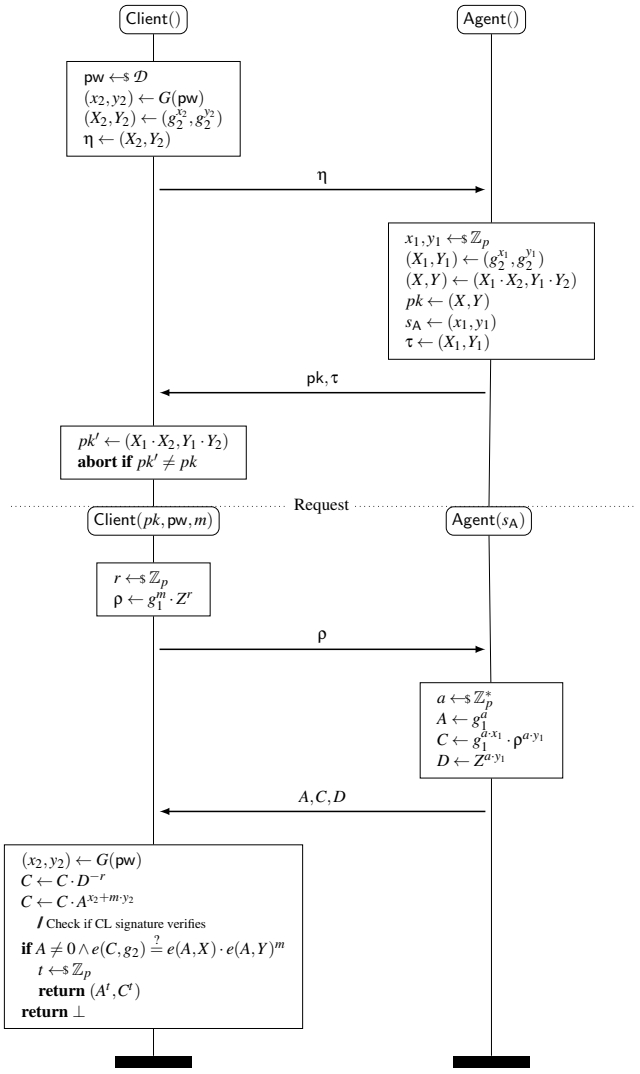

Figure 57: Password Based Signature Based on CL Signatures

- $\mathsf{Dec}_{key}(c)$: 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.

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

1 : $\delta_1, \delta_2 \leftarrow\$ \mathbb{Z}_q$
2 : $key \leftarrow H(H(\mathsf{pw})^{\delta_1+\delta_2} \mod p)$
3 : $a_1 \leftarrow H(H(\mathsf{pw})^{\delta_1} \mod p)$
4 : $a_2 \leftarrow H(H(\mathsf{pw})^{\delta_2} \mod p)$
5 : $e, d, N \leftarrow\$ \mathsf{RSA.Gen}(1^\lambda)$
6 : $d_1 \leftarrow\$ \mathbb{Z}_{\phi(N)}$
7 : $d_2 = d - d_1 \mod \phi(N)$
8 : $A \leftarrow \mathsf{Enc}_{key}(d_1||N||\mathsf{MAC}_{key}(d_1||N))$
9 : $ssk_1, spk_1 \leftarrow \mathsf{Sig.Gen}(\lambda)$
10 : $sk_1, pk_1 \leftarrow \mathsf{PKE.Gen}(\lambda)$
11 : $sk_2, pk_2 \leftarrow \mathsf{PKE.Gen}(\lambda)$
12 : **return to Server 1**$(A, \delta_1, a_2, ssk_1, sk_1)$
13 : **return to Server 2**$(\delta_2, d_2, a_1, sk_2)$
14 : **return to All**$(p, N, e, spk_1, pk_1, pk_2)$

Figure 58: KeyGen for SADS

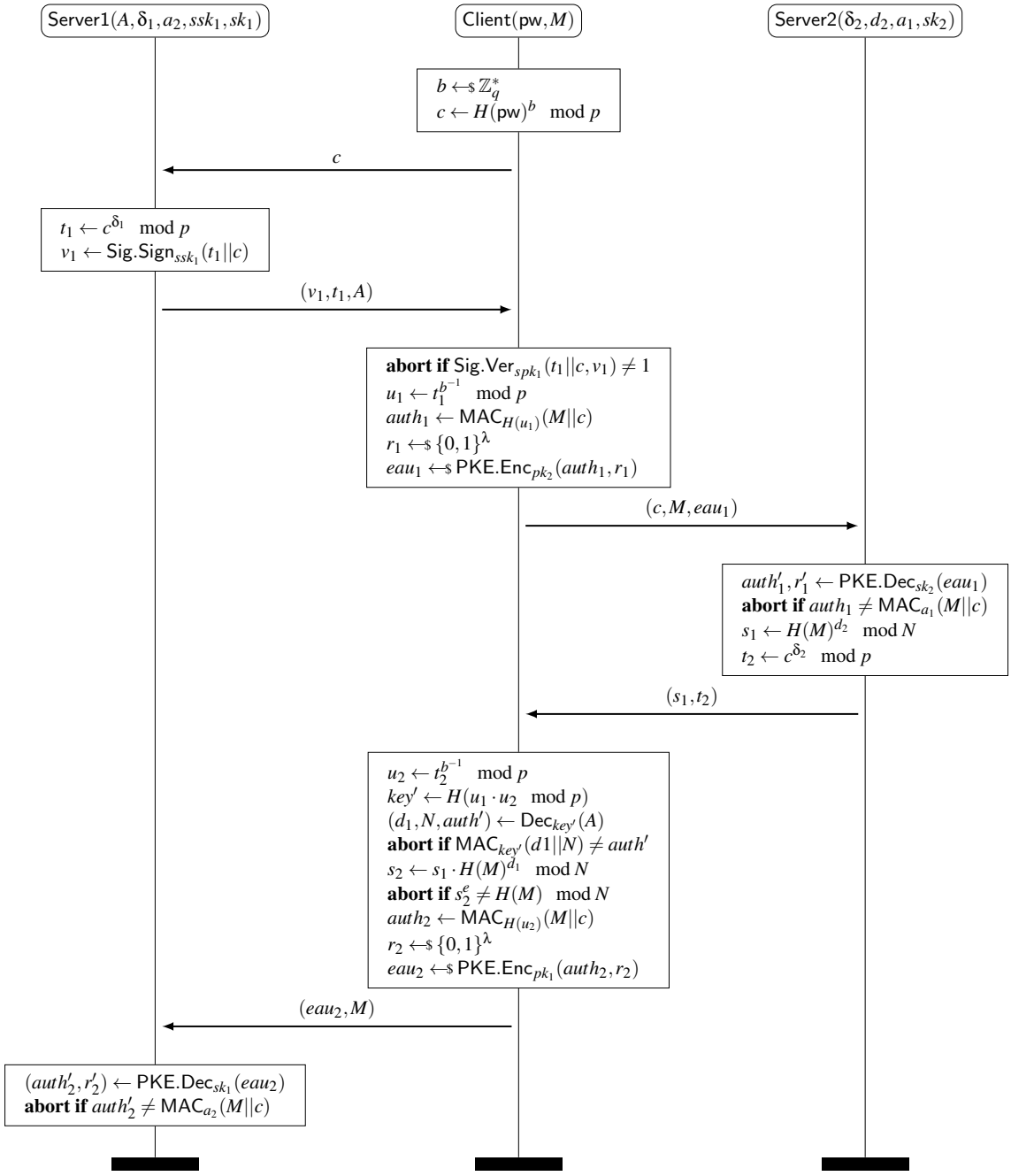

Figure 59: SADS Protocol Construction