Updatable Anonymous Credentials and Applications to Incentive Systems

Johannes Blömer Jan Bobolz Denis Diemert Fabian Eidens

August 23, 2019
Department of Computer Science
Paderborn University, Germany
{jan.bobolz,fabian.eidens}@uni-paderborn.de

Abstract

We introduce updatable anonymous credential systems UACS and use them to construct a new privacy-preserving incentive system. In a UACS, a user holding a credential certifying some attributes can interact with the corresponding issuer to update his attributes. During this, the issuer knows which update function is run, but does not learn the user's previous attributes. Hence the update process preserves anonymity of the user. One example for a class of update functions are additive updates of integer attributes, where the issuer increments an unknown integer attribute value v by some known value k. This kind of update is motivated by an application of UACS to incentive systems. Users in an incentive system can anonymously accumulate points, e.g. in a shop at checkout, and spend them later, e.g. for a discount.

In this paper, we (1) formally define UACS and their security, (2) give a generic construction for UACS supporting arbitrary update functions, and (3) construct a new incentive system using UACS that is efficient while offering offline double-spending protection and partial spending.

Keywords: Anonymous Credentials, Updatable Anonymous Credentials, Privacy, Incentive System, Incentive Collection, Customer Loyalty Program

1 Introduction

Updatable anonymous credential systems. Anonymous credential systems provide a privacy-preserving way of authentication in contrast to the standard authentication through identification via username and password. Authentication with identifying information allows service providers to collect and exchange user-specific data to build a comprehensive user profile without the user's consent. Anonymous credentials mitigate such problems, provide anonymity, and support authentication policies [BCKL08, BBB$^+$18, CL01, CL04, DMM$^+$18]. A credential is parameterized with a vector of attributes (e.g., `birth date`, `affiliation`, `subscription_end`) and when authenticating, users can prove possession of a credential that fulfills a certain access policy (e.g., "`affiliation = university` *or* `subscription_end > [today]`") without revealing anything about the attributes except that they fulfill the access policy.

An extended abstract of this paper appears in the proceedings of CCS'19, ACM.

This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre "On-The-Fly Computing" under the project number 160364472 – SFB 901/3.

While authentication is perfectly anonymous in an anonymous credential system, the *issuer* of a credential always learns the credentials' plaintext attributes. Suppose that a user wants to extend a subscription for which she has a credential as described above. To extend the subscription in a traditional anonymous credential system, she would reveal all her attribute values to the issuer, who would then issue a new credential containing her old attributes and the newly updated `subscription_end` value. This means that updating attributes is not privacy-preserving.

To solve this problem, we introduce *updatable* anonymous credential systems (UACS). A UACS has, in addition to the usual (issue and show) protocols of anonymous credential systems, an *update* protocol. This allows a user to interact with a credential issuer in order to update attributes in a privacy-preserving manner. More specifically, the update protocol takes as input an update function ψ. The user contributes a hidden parameter α and her old credential with attributes $\vec{A}$. By running the protocol with the issuer, the user obtains a new credential on attributes $\vec{A}^* = \psi(\vec{A}, \alpha)$. The issuer only learns what update function ψ is applied, but does not learn $\vec{A}$ or α.

In the subscription update scenario, to add 30 days to the current `subscription_end`, the update function would be defined as $\psi((A, \mathit{subscription\_end}), \alpha) = (A, \mathit{subscription\_end} + 30)$. In this particular case, the hidden parameter α is ignored by ψ, but we will later see update functions that depend on α, e.g. to issue hidden attributes.

The update protocol can be efficiently realized using only building blocks already used by most anonymous credential constructions: Zero-knowledge proofs, commitments, and blind signature schemes with efficient "signing a committed value" protocols.

The idea to implement the UACS update protocol between a user and issuer is as follows: A user's credential on attributes $\vec{A}$ is a digital signature on $\vec{A}$ by the issuer. The user prepares the update by computing her updated attributes $\vec{A}^* = \psi(\vec{A}, \alpha)$ and committing to $\vec{A}^*$. The user then proves that she possesses a valid signature on her old attributes $\vec{A}$ under the public key of the issuer and that she knows α such that the commitment can be opened to $\psi(\vec{A}, \alpha)$. Afterwards, issuer and user run a blind signature protocol to jointly compute a signature on the committed $\vec{A}^*$ (i.e. the updated attributes) without revealing $\vec{A}^*$ to the issuer.

The lack of privacy-preserving updates (as explained above) limits the usefulness of anonymous credentials in several practical applications, such as service subscription management and point collection. In such applications, attributes (such as the subscription end or collected point total) are *routinely* updated and users would prefer not to be tracked through these updates. As a specific example of what UACS enables, we look at *incentive systems*, which is essentially a point collection application.

Incentive systems. An incentive system allows users to collect points (e.g., for every purchase they make), which they can redeem for bonus items or discounts. Such systems aim at reinforcing customer loyalty and incentivize certain behavior through points. In practice, such systems are centralized services, e.g. German Payback [PAY19] and American Express Membership Rewards program [Ame19]. In order to earn points for a purchase, the user reveals her customer ID (e.g., by showing a card). This means that the user's privacy is not protected as every purchase made can be linked to the user's identity by the incentive system provider.

To remedy this, cryptographic incentive systems [MDPD15, JR16, HHNR17] aim at allowing users to earn and spend points anonymously. The general idea is that users store their own points in authenticated form (e.g., in the form of a credential).

We would expect a cryptographic incentive system to offer the following features.

- Anonymity: Providers are unable to link earn/spend transactions to users. In practice, this protects users from having their shopping history linked to their identity and point values.

- Online double-spending protection: A user cannot spend more points than they have earned.

Given continuous access to a central database, the provider can immediately detect double-spending.

- Offline double-spending protection: Detecting double-spending works for stores without continuous access to a central database. Double-spending transactions can be detected and the perpetrating user can be identified. Losses incurred by double-spending can be reclaimed from that user.
- Partial spending: Users do not have to spend all of their points at once.
- Efficiency: The process of earning and spending points can be run on a consumer phone and is fast enough to be accepted by users.

The current state of the art either does not have offline double-spending protection, or does not handle the combination of partial spending and offline double-spending securely. Our system will offer all of these features simultaneously.

We will now explain our UACS-based incentive system by constructing it step by step. As a first sketch, let us assume that the user stores her point count v as an attribute in her credential. When the user earns k additional points, the incentive system provider runs an update on the user's credential, adding k points to her current point count attribute v, i.e. they use update function $\psi((v), \alpha) = (v + k)$. When the user wants to (partially) spend $k \leq v$ points, they run an update ψ such that $\psi((v), \alpha) = v - k$ if $v \geq k$ and $\psi((v), \alpha) = \perp$ otherwise.

Of course, this first sketch does not prevent users from double-spending their points: the spend update operation creates a new credential with lowered point count, but there is no mechanism that forces the user to use the new credential. She can instead keep using the old one, which certifies a higher point count. Hence we modify the first sketch with basic online double-spending protection: The attributes now include a random double-spend identifier *dsid*, i.e. attribute vectors are of the form $\vec{A} = (\mathit{dsid}, v)$. To earn points, the update function still just increases the point count ($\psi((\mathit{dsid}, v), \alpha) = (\mathit{dsid}, v + k)$). When the user wants to spend points, she reveals her *dsid* to the provider and the provider checks that her specific *dsid* has never been revealed to it (spent) before. If that check succeeds, the user chooses a random successor double-spend identifier dsid^* and sets her hidden update parameter α to dsid^*. Finally, user and provider run the update $\psi((\mathit{dsid}, v), \alpha = \mathit{dsid}^*) = (\mathit{dsid}^*, v - k)$, embedding a new dsid^* into the successor credential. If the user tries to spend her old credential (with the old *dsid*) again, the provider will detect the duplicate *dsid*. Anonymity is still preserved because dsid^* is hidden from the provider until the credential is spent.

However, this approach requires all stores where points can be spent to be permanently online in order to check whether a given *dsid* has already been spent. As this is a problem in practice, *offline* double-spending protection is desirable. The idea is that stores that are offline and have an incomplete list of spent *dsid*s may incorrectly accept a spend transaction, but they can later (when they are online again) uncover the identities of double spenders. This allows the provider to recoup any losses due to offline double-spending by pursuing a legal solution to roll back illegal transactions. To incorporate offline double-spending protection, we additionally embed a user's secret key *usk* and a random value *dsrnd* into credentials, i.e. attributes are now $\vec{A} = (\mathit{usk}, \mathit{dsid}, \mathit{dsrnd}, v)$. The update function to earn points is unchanged. To spend points, the provider now sends a random challenge γ to the user and the user reveals $c = \mathit{usk} \cdot \gamma + \mathit{dsrnd} \mod p$ (where *usk*, *dsrnd* are values from her credential attributes). The user chooses new hidden random dsid^*, dsrnd^* for its successor credential and then runs the update for $\psi((\mathit{usk}, \mathit{dsid}, \mathit{dsrnd}, v), \alpha = (\mathit{dsid}^*, \mathit{dsrnd}^*)) = (\mathit{usk}, \mathit{dsid}^*, \mathit{dsrnd}^*, v - k)$. As long as a credential is only spent once, *usk* is perfectly hidden in c. If the user tries to spend the same credential a second time, revealing $c' = \mathit{usk} \cdot \gamma' + \mathit{dsrnd} \mod p$ for some different challenge γ', the provider can compute *usk* from c, c', γ, γ', identifying the double-spender.

This last description comes close to the scheme we present in this paper. However, one problem remains to handle: assume some user double-spends a credential on attributes (*usk*, *dsid*, *dsrnd*, v). For both spend transaction, she receives a remainder amount credential as the successor with attributes $\vec{A}^* = (usk, dsid^*, dsrnd^*, v - k)$. While both transactions will be detected as double spending and the user's key is revealed, the user can keep using both remainder amount credentials anonymously, allowing her to spend $2 \cdot (v - k) > v$ points. To prevent this, we need a mechanism that allows us to recognize remainder amount credentials that were derived from double-spending transactions. This can be achieved by forcing the user to reveal an encryption *ctrace* of $dsid^*$ under *usk* when spending points. As soon as a user double-spends, the provider can compute *usk* as above. With it, he can decrypt all *ctrace* for that user, allowing him to find out what *dsid*s have been derived from double-spending transactions of the double-spending user. Consequently, the user can be held accountable for spending remainder tokens derived from double-spending transactions.

Related work on anonymous credential systems. There is a large body of work on anonymous credential systems, extending the basic constructions [BCKL08, CL01, CL04, PS16] with additional features such as revocation [CKS10, CL01], controlled linkability and advanced policy classes [BBB$^+$18], hidden policies [DMM$^+$18], delegation [BB18, CDD17], and many others. Our notion of privacy-preserving updates on credentials, in its generality, is a new feature (although the general idea of privacy-preserving updates has been briefly sketched before [NDD06]). We show how to efficiently extend the standard blind-signature-based construction of anonymous credentials with updates, which makes our update mechanism compatible with a large part of features presented in existing work (with the exception of [DMM$^+$18], which does not rely on blind signatures).

The scheme in [CKS10] allows issuers to non-interactively update credentials they have issued. In contrast to our updatable credentials, their update cannot depend on hidden attributes and the issuer learns all attributes issued or updated. Their update mechanism is mostly aimed at providing an efficient means to update revocation information, which is controlled by the issuer. Updatable credentials in the sense of our paper allow for the functionality in [CKS10] as well (although in our system, updates are done *interactively* between user and issuer). However, beyond that, our updates can depend on hidden attributes of the user and the issuer does not learn the attributes resulting from the update.

More technically similar to our updatable credential mechanism are *stateful* anonymous credentials [CGH11, GGM14]. A stateful credential contains a state. The user can have his credential state updated to some successor state as prescribed by a public state machine model. For this, the user does not have to disclose his current credential state. Such a state transition is a special case of an update to a state attribute in an updatable credential. In this sense, our construction of updatable credentials generalizes the work of [CGH11].

Related work on incentive systems. Existing e-cash systems are related to incentive systems, but pursue different security goals [CHL05]. E-cash does not support the accumulation of points within a single token. Instead, each token corresponds to a coin and can be identified. To spend a coin, a user transfers it to another owner. In incentive systems, a number of points is accumulated into a single token (i.e. the token is like a bank account rather than a coin).

A cryptographic scheme that considers the collection of points in a practical scenario is described by Milutinovic et al. in [MDPD15]. Their scheme uCentive can be seen as a special e-cash system, where a so called uCent corresponds to a point. The user stores and spends all uCents individually, which induces storage and communication cost linear in the number of uCents (hence efficiency is restricted). Similar to our system, uCentive builds upon anonymous credentials (but without updates) and commitments, but they do not offer offline double-spending protection.

Jager and Rupp [JR16] introduce black-box accumulation (BBA) as a building-block for incentive systems. They formalize the core functionality and security of such systems based on the natural requirement that users collect and sum up values in a privacy-preserving way. In detail, they present a generic construction of BBA combining homomorphic commitments, digital signatures, and non-interactive zero-knowledge proofs of knowledge (Goth-Sahai proofs [GS08]). The BBA solution has three major shortcomings: the token creation and redemption processes are linkable, users have to redeem all of their points at once, and stores must be permanently online to detect double-spending.

Hartung et al. [HHNR17] present an improved framework of black-box accumulation (BBA+) based on the framework introduced in [JR16]. In [HHNR17], BBA is extended with offline double-spending prevention (on which we base our offline double-spending mechanism) and other desirable features. Because of efficiency reasons, users needs to reveal their point count whenever they spend points. Their efficiency problems mainly stem from the use of Groth-Sahai proofs. Note that the use of Groth-Sahai is inherent in their approach (because their proof statements are mostly about group elements). It is unclear whether their construction can be made more efficient by exchanging the proof system without changing the approach. In contrast, our incentive system can be instantiated in a Schnorr proof setting (with proof statements mostly about discrete logarithms). Because the Schnorr setting allows for very efficient proofs [BBB$^+$18, CCs08], our incentive system is also very efficient.

What prior work does not handle is the conjunction of *offline-double spending prevention* and *partial spending* (even when disregarding efficiency concerns). If a spend operation is later detected as double-spending, the remainder token still remains valid in prior constructions. Our construction solves this, allowing the provider to trace all tokens derived from double-spent transactions to a user. The price of this solution is forward and backward privacy as defined in [HHNR17], which our scheme does not offer.

Overall, with the UACS-based incentive system approach, we improve upon the current state-of-the-art [HHNR17] (from 2017 ACM CCS) in two ways: (1) efficiency (mostly because our approach allows us to avoid Groth-Sahai proofs), and (2) we enable the combination of offline double-spending prevention and partial spending. It is an interesting open question whether or not our remainder token tracing mechanism can be combined with forward and backward privacy as in [HHNR17].

A basic version of the idea of using updates on credentials for incentive systems has been informally considered in a 2005 technical report [DDD05] before.

Our contribution and structure of this paper. We introduce UACS formally in Section 3 and define its security properties. In Section 4, we construct UACS generically from blind signature schemes and in Section 5, we sketch how to efficiently instantiate UACS using the generic framework. We define formal requirements for incentive systems in Section 6, modeling our double-spend prevention mechanism and defining security. In Section 7, we construct an incentive system from a UACS. Finally, we practically evaluate our incentive system in Section 8.

2 Preliminaries

Throughout the paper, we refer to a public-parameter generation ppt $\mathcal{G}$ that outputs public parameters pp given unary security parameter 1^λ. A function $f : \mathbb{N} \to \mathbb{R}$ is negligible if for all $c > 0$ there is an x_0 such that $f(x) < 1/x^c$ for all $x > x_0$. We refer to a negligible function as *negl*. We write $\mathrm{output}_A[A \leftrightarrow B]$ for interactive algorithms A, B to denote the output of A after interacting with B. The support of a probabilistic algorithm A on input x is denoted by $[A(x)] := \{y \mid \Pr[A(x) = y] > 0\}$. The expression $\mathsf{ZKAK}[(w); (x, w) \in R]$ denotes a zero-knowledge argument of knowledge protocol where the prover proves knowledge of w such that (x, w) is in some NP relation R. The zero-knowledge argument of knowledge can be simulated perfectly given a trapdoor [Dam00] and there exists an expected polynomial-time extractor that, given black-box access to a successful prover, computes a witness w with probability 1 [Dam00].

We define security with an oracle-based notation, where an adversary $\mathcal{A}$ gets oracle access to some methods or protocols. Some oracles are interactive, i.e. they may send and receive messages during a call. We distinguish between the oracle's (local) *output*, which is generally given to the adversary, and the oracle's *sent and received messages*. The notation $\mathbf{Oracle}(\cdot)$ denotes that $\mathcal{A}$ chooses x, then oracle $\mathbf{Oracle}(x)$ is run interacting with $\mathcal{A}$. $\mathcal{A}$ is given the output of the oracle (if any). The notation $(x, y) \mapsto \mathbf{Oracle}_0(x) \leftrightarrow \mathbf{Oracle}_1(y)$ denotes that the adversary $\mathcal{A}$ chooses inputs x, y, then $\mathbf{Oracle}_0(x) \leftrightarrow \mathbf{Oracle}_1(y)$ are run, interacting with one another. $\mathcal{A}$ is given the output of both oracles, but not the messages sent or received by the oracles.

For blind signatures, we require that the blind signing protocol is of the form "commit to the message(s) to sign, then jointly compute the signature". As such, we model the commitment step and the "receive a signature on the committed value" step separately.

Definition 1. A blind signature scheme for signing committed values Π_{sig} consists of the following (ppt) algorithms:

$\mathsf{KeyGen}_{\text{sig}}(pp, 1^n) \to (pk, sk)$ generates a key pair (pk, sk) for signatures on vectors of n messages. We assume n can be efficiently derived from pk.

$\mathsf{Commit}_{\text{sig}}(pp, pk, \vec{m}, r) \to c$ given messages $\vec{m} \in \mathcal{M}^n$ and randomness r, deterministically computes a commitment c.

$\mathsf{BlindSign}_{\text{sig}}(pp, pk, sk, c) \leftrightarrow \mathsf{BlindRcv}_{\text{sig}}(pp, pk, \vec{m}, r) \to \sigma$ with common input pp, pk is an interactive protocol. The signer's input is sk, c. The receiver's input consists of the messages $\vec{m}$ and commitment randomness r. The receiver outputs a signature σ.

$\mathsf{Vrfy}_{\text{sig}}(pp, pk, \vec{m}, \sigma) \to b$ deterministically checks signature σ and outputs 0 or 1.

A blind signature scheme is *correct* if for all $\lambda, n \in \mathbb{N}$ and all $pp \in [\mathcal{G}(1^\lambda)]$, $(pk, sk) \in [\mathsf{KeyGen}(pp, 1^n)]$, all $\vec{m} \in \mathcal{M}^n$, and for every commitment randomness r, it holds that

$$\begin{aligned}\Pr[&\mathsf{BlindSign}(pp, pk, sk, \mathsf{Commit}(pp, pk, \vec{m}, r)) \\ &\leftrightarrow \mathsf{BlindRcv}(pp, pk, \vec{m}, r) \to \sigma : \\ &\mathsf{Vrfy}(pp, pk, \vec{m}, \sigma) = 1] = 1\end{aligned}$$

◇

We require for a blind signature scheme unforgeability and perfect message privacy, cf. Appendix A, Definitions 17 and 18. Definition 1 can be instantiated by Pointcheval Sanders signatures [PS16].

We furthermore need public-key encryption with the property that the key generation $\mathsf{KeyGen}_{\text{enc}}$ first generates a secret key sk, from which the corresponding public key $pk = \mathsf{ComputePK}(pp, sk)$ can be deterministically computed. For example, for ElGamal encryption with fixed base g, we have that $sk \leftarrow \mathbb{Z}_p$ and the public key is $\mathsf{ComputePK}(pp, sk) = g^{sk}$.

Definition 2. A public-key encryption scheme Π_{enc} consists of four ppt algorithms $\mathsf{KeyGen}_{\text{enc}}$, $\mathsf{ComputePK}_{\text{enc}}, \mathsf{Encrypt}_{\text{enc}}, \mathsf{Decrypt}_{\text{enc}}$ such that $\mathsf{Decrypt}_{\text{enc}}$ and $\mathsf{ComputePK}_{\text{enc}}$ are deterministic and for all $pp \in [\mathcal{G}(1^\lambda)]$, all $sk \in [\mathsf{KeyGen}_{\text{enc}}(pp)]$, and all messages m it holds that $\Pr[\mathsf{Decrypt}_{\text{enc}}(pp, sk, \mathsf{Encrypt}_{\text{enc}}(pp, \mathsf{ComputePK}_{\text{enc}}(pp, sk), m)) = m] = 1$. ◇

For the sake of privacy in our constructions, we will later demand *key-indistinguishable* CPA security. This notion requires that, in addition to CPA-security, ciphertexts cannot be linked to their public key (cf. Appendix A, Definition 19). This is the case for ElGamal encryption. Finally, we use an additively malleable commitment scheme.

Definition 3. A malleable commitment scheme Π_{cmt} consists of four ppt algorithms (KeyGen, $\mathsf{Commit}, \mathsf{Vrfy}, \mathsf{Add}$) s.t. for all $pp \in [\mathcal{G}(1^\lambda)]$, all $pk \in [\mathsf{KeyGen}(pp)]$, all $m \in M_{pp}$, and all $(c, o) \in [\mathsf{Commit}(pp, pk, m)]$

- The message space M_{pp} is an (additive) group,
- Vrfy and Add are deterministic,
- $\mathsf{Vrfy}(pp, pk, c, o, m) = 1$, and
- For $c' = \mathsf{Add}(pp, pk, c, k)$, it holds that $(c', o) \in [\mathsf{Commit}(pp, pk, m + k)]$.

We require the commitment to be perfectly binding and computationally hiding, cf. Appendix A, Definitions 20 and 21. Definition 3 can be instantiated by ElGamal encryption.

3 Updatable Anonymous Credentials

In UACS, there are three roles: issuers, users, and verifiers. Each role can be instantiated arbitrarily many times. Issuers hold keys to issue credentials to users. Credentials are certificates that are parameterized with attributes. Users can prove possession of a credential to verifiers. Users can interact with their credential's issuer to change its attributes.

3.1 Algorithms of UACS

A UACS consists of ppt algorithms $\mathsf{Setup}, \mathsf{IssuerKeyGen}$, and interactive protocols $\mathsf{Issue} \leftrightarrow \mathsf{Receive}$, $\mathsf{Update} \leftrightarrow \mathsf{UpdRcv}$, and $\mathsf{ShowPrv} \leftrightarrow \mathsf{ShowVrfy}$. We explain them in the following:

Setup We assume that some trusted party has already generated public parameters $pp \leftarrow \mathcal{G}(1^\lambda)$. pp may, for example, contain a description of a group, which can also be used for any number of other cryptographic applications. To set up a UACS, a trusted party generates UACS-specific parameters $cpp \leftarrow \mathsf{Setup}(pp)$. cpp may, for example, contain pp and parameters for a zero-knowledge proof system. The distinction between $\mathcal{G}$ and Setup is made to enable formal compatibility of UACS with other primitives, as long as they use the same pp. cpp is published and we assume that an attribute universe $\mathbb{A}$ is encoded in cpp (e.g., $\mathbb{A} = \mathbb{Z}_p$).

Key generation Whenever a new issuer wants to participate in the UACS, he first chooses an attribute vector length $n \in \mathbb{N}$ and then generates a key pair $(pk, sk) \leftarrow \mathsf{IssuerKeyGen}(cpp, 1^n)$. The secret key sk will be used to issue and update credentials, the public key pk will be used to identify the issuer and to verify credentials issued by him. Credentials by this issuer will be parameterized with a vector $\vec{A} \in \mathbb{A}^n$.

In a UACS, users do not generally need keys. This is in contrast to the usual definitions of anonymous credentials, in which users explicitly generate a secret identity. We generalize that approach for UACS and leave the implementation of user identities to the application, if desired (see Section 3.3).

Issuing and updating credentials Users have two ways to receive new credentials: receive a fresh credential from an issuer, or update an old one.

Assume the user holds a credential $cred$ with attributes $\vec{A}$ and wants to update it. User and issuer first agree on an update function $\psi : \mathbb{A}^n \times \{0,1\}^* \to \mathbb{A}^n \cup \{\perp\}$, and the user secretly chooses a hidden parameter α s.t. $\psi(\vec{A}, \alpha) \neq \perp$. The issuer and user then engage in an interactive protocol: the issuer runs $\mathsf{Update}(cpp, pk, \psi, sk)$ while the user runs $\mathsf{UpdRcv}(cpp, pk, \psi, \alpha, cred)$. Afterwards,

UpdRcv outputs a new credential *cred** with attributes $\vec{A}^* = \psi(\vec{A}, \alpha)$ or the failure symbol (e.g., if $\psi(\vec{A}, \alpha) = \perp$).

Furthermore, Update outputs a bit b to the issuer, which informally serves as an indicator whether or not the update was successful. In particular $b = 1$ guarantees that $\psi(\vec{A}, \alpha) \neq \perp$. This effectively means that, with an appropriately chosen ψ, a credential update implicitly includes a check of the old credential's attributes.

We model issuing a new credential essentially as an update of an "empty" credential: User and issuer first agree on an update function $\psi : \{\perp\} \times \{0,1\}^* \to \mathbb{A}^n \cup \{\perp\}$, and the user secretly chooses a hidden parameter α s.t. $\psi(\perp, \alpha) \neq \perp$. Then the issuer runs $\mathsf{Issue}(cpp, pk, \psi, sk)$ while the user runs $\mathsf{Receive}(cpp, pk, \psi, \alpha)$. Afterwards, $\mathsf{Receive}$ outputs a credential *cred* with attributes $\vec{A} = \psi(\perp, \alpha)$ or the failure symbol (e.g., if $\psi(\perp, \alpha) = \perp$).

In contrast to the usual definition of anonymous credentials, the issuer does not necessarily know the exact attributes he is issuing (he does not know the input to the update function ψ). To issue attributes $\vec{A}$ fully known to the issuer, the update function ψ can be set to $\psi(\perp, \alpha) = \vec{A}$ (i.e. ψ ignores α and outputs a constant). An example for an update function hiding some attributes from the issuer is $\psi(\perp, \alpha = (a, b)) = (0, a, b, a + b)$, where the user's first attribute would be 0 (which the issuer knows), but the user may freely choose a, b. To restrict the user's choice of α, the update function $\psi(\perp, \alpha)$ can output $\perp$, in which case the issuance should fail (e.g., define $\psi(\perp, \alpha = (a, b)) = \perp$ if $a + b > 20$).

Showing credentials To prove possession of a credential *cred* with attributes $\vec{A}$ from some issuer with public key *pk*, the user and the verifier first agree on a predicate $\phi : \mathbb{A}^n \times \{0,1\}^* \to \{0,1\}$. The user chooses a hidden parameter α such that $\phi(\vec{A}, \alpha) = 1$. Then the user runs $\mathsf{ShowPrv}(cpp, pk, \phi, \alpha, cred)$, interacting with the verifier running $\mathsf{ShowVrfy}(cpp, pk, \phi)$. Afterwards, $\mathsf{ShowVrfy}$ outputs a bit b. If $b = 1$, the verifier knows that the user possesses a credential, issued by *pk*, with attributes $\vec{A}$ s.t. $\exists \alpha : \phi(\vec{A}, \alpha) = 1$.

Formal definition We now formally define UACS. First, we need the notion of *valid* credentials: the predicate $\mathsf{ValidCred}(cpp, pk, cred, \vec{A})$ defines whether *cred* is considered a valid credential with attributes $\vec{A}$ for issuer *pk* under UACS public parameters *cpp*. Intuitively, we want all credentials output by $\mathsf{Receive}$ and UpdRcv to be valid with the attributes the user expects. Formally, $\mathsf{ValidCred}(cpp, pk, cred, \vec{A})$ is recursively defined as follows:

- if $\perp \neq cred \in [\mathsf{Receive}(cpp, pk, \psi, \alpha)]$, then $\mathsf{ValidCred}(cpp, pk, cred, \psi(\perp, \alpha)) = 1$.
- if $\perp \neq cred^* \in [\mathsf{UpdRcv}(cpp, pk, \psi, \alpha, cred)]$, and $\mathsf{ValidCred}(cpp, pk, cred, \vec{A}) = 1$, then it holds that $\mathsf{ValidCred}(cpp, pk, cred^*, \psi(\vec{A}, \alpha)) = 1$.

In all other cases, $\mathsf{ValidCred}(\dots) = 0$. $\mathsf{ValidCred}$ is not necessarily efficiently computable, but serves a purpose in our definitions.

Definition 4 (Updatable anonymous credential system)**.** An updatable anonymous credential system Π_{uacs} (UACS) consists of the ppt algorithms $\mathsf{Setup}, \mathsf{IssuerKeyGen}, \mathsf{Issue}, \mathsf{Receive}, \mathsf{Update}, \mathsf{UpdRcv}$, $\mathsf{ShowPrv}$, and $\mathsf{ShowVrfy}$. Let Φ be a set of supported predicates ϕ, and let Ψ be a set of supported update functions ψ (Φ and Ψ may depend on *cpp* and *pk*).
A UACS is correct, if whenever $\mathsf{ValidCred}((cpp, pk, \vec{A}, cred)) = 1$:

- if $\phi \in \Phi$ and $\phi(\vec{A}, \alpha) = 1$, then $\mathsf{ShowVrfy}(cpp, pk, \phi)$ accepts after interacting with algorithm $\mathsf{ShowPrv}(cpp, pk, \phi, \alpha, cred)$.
- if $\psi \in \Psi$, $\alpha \in \{0,1\}^*$ and $\psi(\vec{A}, \alpha) \neq \perp$, then after interacting with one another, $\mathsf{Update}(cpp, pk, \psi, sk)$ outputs 1, and $\mathsf{UpdRcv}(cpp, pk, \psi, \alpha, cred)$ does not output $\perp$. $\diamond$

3.2 Security of UACS

On a high level, a UACS has two security goals: (1) Anonymity: honest users' privacy should be protected (even against malicious issuers and verifiers), meaning that user actions should be unlinkable and hide as much data as possible. (2) Soundness: malicious users should not be able to show or update a credential they have not obtained by the issuer. These are explained next.

3.2.1 Anonymity

Our anonymity definition follows a simulation approach. This means that we require existence of simulators that can simulate the user's role of the *show*, *issue*, and *update* protocols. For this, the input for the simulators is exactly the information that the issuer/verifier should learn from the interaction (plus a trapdoor to enable simulation). The issuer/verifier cannot learn the user's private information because the protocols can be simulated without it, hence its transcripts effectively do not contain information about private information. This makes it easy to succinctly express exactly what the issuer/verifier learns and enables use of the UACS in larger contexts (for example, for our incentive system, an indistinguishability definition would not suffice).

More specifically, for an *anonymous* UACS, there exists an efficient algorithm $\mathfrak{S}_{\mathsf{Setup}}(pp)$ that outputs cpp (like Setup) and a simulation trapdoor td. Then the following simulators simulate a user's protocols: $\mathfrak{S}_{\mathsf{Receive}}(td, pk, \psi)$ simulates receiving a credential (note that this means that the issuer learns only ψ, but not α). $\mathfrak{S}_{\mathsf{UpdRcv}}(td, pk, \psi)$ simulates having a credential updated (meaning the issuer only learns ψ, but not α or the old attributes $\vec{A}$, nor any information about the specific credential-to-be-updated). Finally, $\mathfrak{S}_{\mathsf{ShowPrv}}(td, pk, \phi)$ simulates showing a credential (the verifier only learns ϕ, not $\vec{A}$ or α).

In the real world, the issuer will usually learn whether or not a credential issuing or update has worked, meaning whether or not the user's protocol side outputs a non-error value $\neq\perp$ (e.g., because the user would immediately ask to run failed protocols again). To make sure that the issuer cannot learn anything from this bit of information, we make this part of the simulation: $\mathfrak{S}_{\mathsf{Receive}}$ actually simulates a protocol $\mathsf{Receive}'$, which behaves like $\mathsf{Receive}$, but after the interaction sends a bit b to the issuer, indicating whether or not $\mathsf{Receive}$ outputs the error symbol or a credential. Analogously, $\mathfrak{S}_{\mathsf{UpdRcv}}$ simulates UpdRcv'.

Definition 5 (Simulation Anonymity)**.** A UACS Π_{uacs} has *simulation anonymity* if there exist ppt simulators $\mathfrak{S}_{\mathsf{Setup}}, \mathfrak{S}_{\mathsf{Receive}}, \mathfrak{S}_{\mathsf{ShowPrv}}, \mathfrak{S}_{\mathsf{UpdRcv}}$ such that for all (unrestricted) adversaries $\mathcal{A}$ and all $pp \in [\mathcal{G}(1^\lambda)]$: If $(cpp, td) \in [\mathfrak{S}_{\mathsf{Setup}}(pp)]$, then

- $\Pr[\mathsf{Setup}(pp) = cpp] = \Pr[\mathfrak{S}_{\mathsf{Setup}}(pp) = (cpp, \cdot)]$
- $\text{output}_{\mathcal{A}}[\mathfrak{S}_{\mathsf{Receive}}(td, pk, \psi) \leftrightarrow \mathcal{A}]$ is distributed exactly like $\text{output}_{\mathcal{A}}[\mathsf{Receive}'(cpp, pk, \psi, \alpha) \leftrightarrow \mathcal{A}]$ for all $pk, \alpha \in \{0,1\}^*$, and $\psi \in \Psi$ with $\psi(\perp, \alpha) \neq\perp$.
- $\text{output}_{\mathcal{A}}[\mathfrak{S}_{\mathsf{UpdRcv}}(td, pk, \psi) \leftrightarrow \mathcal{A}]$ is distributed exactly like $\text{output}_{\mathcal{A}}[\mathsf{UpdRcv}'(cpp, pk, \psi, \alpha, cred) \leftrightarrow \mathcal{A}]$ for all pk, $\psi \in \Psi$ and $cred, \vec{A}$ such that $\mathsf{ValidCred}(cpp, pk, cred, \vec{A}) = 1$ and $\psi(\vec{A}, \alpha) \neq\perp$.
- $\text{output}_{\mathcal{A}}[\mathfrak{S}_{\mathsf{ShowPrv}}(td, pk, \phi) \leftrightarrow \mathcal{A}]$ is distributed exactly like $\text{output}_{\mathcal{A}}[\mathsf{ShowPrv}(cpp, pk, \phi, \alpha, cred) \leftrightarrow \mathcal{A}]$ for all pk, $\phi \in \Phi$ and $cred, \vec{A}, \alpha$ such that $\mathsf{ValidCred}(cpp, pk, cred, \vec{A}) = 1$ and $\phi(\vec{A}, \alpha) = 1$ ◇

3.2.2 Soundness

Informally, *soundness* should enforce that users cannot show or update credentials they have not obtained from the issuer. Soundness protects issuers and verifiers against malicious users. The challenge in defining soundness is that because of anonymity in UACS, issuers and verifiers do not know what attributes result from Issue and Update operations, so they cannot easily check whether or not security was broken. For this reason, we intuitively say that soundness is broken if an adversary can run a series of issue/update/verify protocols, for which there is no reasonable explanation given the update functions for which the issuer has issued/updated credentials.

The soundness definition is game-based: an adversary $\mathcal{A}$ is run through an experiment (cf. Figure 1). The experiment simulates an honest issuer and honest verifier. $\mathcal{A}$ can ask to be issued credentials, to have them updated, or to show them, choosing the update functions ψ and show-predicates ϕ that the issuer/verifier shall use. Eventually, $\mathcal{A}$ halts. Now, to judge whether or not $\mathcal{A}$ won, the experiment runs an extractor $\mathcal{E}$ (whose existence we require from sound UACS). $\mathcal{E}$ outputs an *explanation list* $\mathcal{L}$, which conjectures what hidden parameters $\mathcal{A}$ used in each issue/update/verify protocol. If $\mathcal{E}$ fails to produce an explanation list $\mathcal{L}$ that is consistent with what we've observed during the experiment, then $\mathcal{A}$ wins. Consistency mainly hinges on the bit output by $\mathsf{ShowVrfy}$ and Update: If $\mathsf{ShowVrfy}(cpp, pk, \phi)$ outputs 1, then we expect that $\mathcal{L}$ shows a series of issue/update operations that explains why $\mathcal{A}$ possesses a credential with attributes $\vec{A}$ for which there exists an α that satisfies $\phi(\vec{A}, \alpha)$. Similarly, if $\mathsf{Update}(cpp, pk, \psi, sk)$ outputs 1, then $\mathcal{L}$ should show that $\mathcal{A}$ has a credential with attributes $\vec{A}$ for which there exists an α with $\psi(\vec{A}, \alpha) \neq \perp$.

Formally, an explanation list $\mathcal{L}$ contains one entry per operation that $\mathcal{A}$ requested during the experiment.

- For $\mathsf{ShowVrfy}$ or Update operations, the ith entry is a tuple $(\vec{A}_i, \alpha_i)$.
- For Issue operations, the entry is some hidden parameter α_i.

These entries naturally induce sets E_i of attribute vectors that we expect $\mathcal{A}$ to have after the ith operation. Initially, $E_0 = \emptyset$. Then inductively:

- if the ith operation is $\mathsf{ShowVrfy}$, no credentials are issued, i.e. $E_i = E_{i-1}$.
- if the ith operation is $\mathsf{Issue}(cpp, pk, \psi, sk)$, we expect $\mathcal{A}$ to now have a credential with attributes $\psi(\perp, \alpha_i)$, i.e. $E_i = E_{i-1} \cup \{\psi(\perp, \alpha_i)\}$ if $\psi(\perp, \alpha_i) \neq \perp$.
- if the ith operation is $\mathsf{Update}(cpp, pk, \psi, sk)$ (and Update output 1), we expect $\mathcal{A}$ to now have a credential with attributes $\psi(\vec{A}_i, \alpha_i)$, i.e. $E_i = E_{i-1} \cup \{\psi(\vec{A}_i, \alpha_i)\}$ if $\psi(\vec{A}_i, \alpha_i) \neq \perp$. If Update output 0, we expect no new credential to have been issued, i.e. $E_i = E_{i-1}$.

We say that an explanation list $\mathcal{L}$ is *consistent* if it explains all the instances where $\mathsf{ShowVrfy}$ or Update output 1:

- if the ith operation was $\mathsf{ShowVrfy}(cpp, pk, \phi)$ with output 1, then the list's $(\vec{A}_i, \alpha_i)$ fulfills ϕ (i.e. $\phi(\vec{A}_i, \alpha_i) = 1$) and $\vec{A}_i$ is the result of an earlier issue/update operation (i.e. $\vec{A}_i \in E_{i-1}$).
- if the ith operation was $\mathsf{Update}(cpp, pk, \psi, sk)$ with output 1, then the list's $(\vec{A}_i, \alpha_i)$ fulfills ψ (i.e. $\psi(\vec{A}_i, \alpha_i) \neq \perp$) and $\vec{A}_i$ is the result of an earlier issue/update operation (i.e. $\vec{A}_i \in E_{i-1}$).

Definition 6 (Soundness)**.** We say that Π is *sound* if there exists an expected polynomial time algorithm $\mathcal{E}$ (probability for runtime is over $\mathcal{A}$'s randomness r_A and $\mathcal{E}$'s randomness), such that for all ppt adversaries $\mathcal{A}$, there exists a negligible function *negl* with

$$\Pr[\mathrm{Exp}^{\mathrm{sound}}(\Pi, \mathcal{A}, \mathcal{E}, \lambda) = 1] \leq \mathit{negl}(\lambda)$$

for all λ. ◇

```
$\text{Exp}^{\text{sound}}(\Pi, \mathcal{A}, \mathcal{E}, \lambda)$:
    $cpp \leftarrow \mathsf{Setup}(\mathcal{G}(1^\lambda))$, $(1^n, st) \leftarrow \mathcal{A}(cpp)$, for some $n \in \mathbb{N}$
    $(pk, sk) \leftarrow \mathsf{IssuerKeyGen}(cpp, 1^n)$
    **Run** $\mathcal{A}^{\mathsf{Issue}(cpp,pk,\cdot,sk),\mathsf{Update}(cpp,pk,\cdot,sk),\mathsf{ShowVrfy}(cpp,pk,\cdot)}(pk, st)$
    Let $r_\mathcal{A}$ be the randomness of $\mathcal{A}$
    Let $r_{\mathsf{Issue}}$, $r_{\mathsf{Update}}$, $r_{\mathsf{ShowVrfy}}$ be the oracles' randomness.
    **Run** $\mathcal{L} \leftarrow \mathcal{E}^\mathcal{A}(cpp, r_\mathcal{A}, r_{\mathsf{Issue}}, r_{\mathsf{Update}}, r_{\mathsf{ShowVrfy}})$.
    **Output** 0 **if** $\mathcal{L}$ is consistent, **otherwise** output 1
```

Figure 1: Soundness experiment for UACS

A few technical notes: the randomness $r_\mathcal{A}, r_{\mathsf{Issue}}, r_{\mathsf{Update}}$, and r_{ShowVrfy} together with oracle access to $\mathcal{A}$ can be used by $\mathcal{E}$ to effectively re-run the experiment exactly as it happened before. It can then use, for example, forking techniques to extract relevant witnesses from $\mathcal{A}$. The requirement that $\mathcal{E}$ must be efficient (expected polynomial time) is somewhat arbitrary at this point: the definition would still make sense if $\mathcal{E}$'s runtime were unrestricted, since $\mathcal{E}$ is just a way to express that there must *exist* a consistent explanation. However, for constructions that use UACS as a primitive (such as in Section 3.3 and in our incentive system later), $\mathcal{E}$ must often be efficient so that an efficient reduction can run $\mathcal{E}$ to obtain $\mathcal{A}$'s hidden values. This effectively implies that in a sound UACS, $\mathcal{A}$ must *know* (in the sense of an argument of knowledge) the values $\vec{A}, \alpha$ it uses for issue/update/show.

3.3 A Note on User Secrets and Pseudonyms

Usually, users in a credential system have a personal key *usk* that is embedded in their credentials. They can derive any number of unlinkable pseudonyms N from *usk*. UACS generalize this: *usk* and pseudonyms are not immediate part of the definition, but because UACS naturally supports hidden attribute issuing, *usk* can be seen as just another UACS attribute.

To implement user keys and pseudonyms in UACS, one can use the following template: The user chooses *usk* randomly from a superpolynomial-size domain. Pseudonyms N are commitments to *usk*, i.e. $(N, o) \leftarrow \mathsf{Commit}(pp, pk, usk)$. The user privately stores the open value o for the pseudonym.

Assume the user identified himself with the pseudonym N. To receive a credential on attributes $\vec{A}$, the user sets his hidden parameter to $\alpha = (usk, o)$ and the update function (1) checks if N opens to *usk* using o and then (2) embeds *usk* as an attribute into the credential:

$$\psi(\perp, \alpha) = \begin{cases} (usk, \vec{A}) \text{ if } \mathsf{Vrfy}(pp, pk, N, o, usk) = 1 \\ \perp \text{ otherwise} \end{cases}$$

This ensures that only the user who created N can receive the credential and that the *usk* embedded into it is consistent with N.

Similarly, show predicates ϕ can be modified such that the user supplies the additional hidden parameter $\alpha = o$ and ϕ additionally checks that the commitment N opens to the user secret embedded in the credential (ensuring that the credential actually belongs to the user behind N). When updating a credential, the update function ψ should always leave the *usk* attribute intact.

As a technical note on security, if the commitment is computationally hiding, then simulation anonymity can be used to argue that anonymity is preserved: the protocols can be simulated without *usk* or o. This also motivates why we chose a simulation-based anonymity definition: a reduction to the commitment hiding property would have to embed a challenge commitment N into some UACS protocol, which it then needs to be able to simulate because without an open value, it cannot run the protocol honestly. If the commitment is computationally binding, then soundness implies that $\mathcal{E}$ can *extract* an open value o. For example, in a scenario where a user can use the same pseudonym for two different *usk*, a consistent explanation would contain two open values to break the binding property of the commitment. This motivates the choice of restricting the UACS extractor $\mathcal{E}$ to (expected) polynomial time, as otherwise, the reduction to the commitment binding property would not be efficient.

4 Generic Construction of UACS

An UACS can be generically constructed from any blind signature scheme $\Pi_{\mathrm{sig}} = (\mathsf{KeyGen}_{\mathrm{sig}}, \mathsf{Commit}_{\mathrm{sig}}, \mathsf{BlindSign}_{\mathrm{sig}}, \mathsf{BlindRcv}_{\mathrm{sig}})$ (Definition 1) as follows:

Keys The public parameters *cpp* of the UACS are the public parameters of Π_{sig} plus a zero-knowledge argument common reference string (this will later allow us to simulate zero-knowledge arguments). For the issuer, $\mathsf{IssuerKeyGen}(cpp, 1^n)]$ generates the key pair (pk, sk) by running the blind signature scheme's key generation $\mathsf{KeyGen}_{\mathrm{sig}}(pp, 1^n)$ to get a key for blocks of n messages.

Showing credentials A credential *cred* with attributes $\vec{A}$ is simply a signature σ on $\vec{A}$ under the issuer's *pk*. Showing a credential $(\mathsf{ShowPrv}(cpp, pk, \phi, \alpha, cred = \sigma) \leftrightarrow \mathsf{ShowVrfy}(cpp, pk, \phi))$ simply has the user run a zero-knowledge argument of knowledge

$$\mathsf{ZKAK}[(\vec{A}, \alpha, \sigma); \mathsf{Vrfy}_{\mathrm{sig}}(pp, pk, \vec{A}, \sigma) = 1 \wedge \phi(\vec{A}, \alpha) = 1]$$

proving that he is in possession of a valid signature on hidden attributes $\vec{A}$ and knows α such that ϕ is satisfied. $\mathsf{ShowVrfy}$ outputs 1 if and only if the proof is accepted.

Updating and issuing credentials For $\mathsf{Update}(cpp, pk, \psi, sk) \leftrightarrow \mathsf{UpdRcv}(cpp, pk, \psi, \alpha, cred)$, the user has a credential $cred = \sigma$, which is a signature on $\vec{A}$, and wants a signature on $\vec{A}^* := \psi(\vec{A}, \alpha)$. He computes a commitment c to $\vec{A}^*$. He proves that c is well-formed and that he possesses a signature σ on $\vec{A}$:

$$\begin{aligned}\mathsf{ZKAK}[(\vec{A}, \sigma, \alpha, r); c = \mathsf{Commit}_{\mathrm{sig}}(pp, pk, \psi(\vec{A}, \alpha), r)\\ \wedge \mathsf{Vrfy}_{\mathrm{sig}}(pp, pk, \vec{A}, \sigma) = 1 \wedge \psi(\vec{A}, \alpha) \neq \perp].\end{aligned}$$

If the proof is rejected, the issuer outputs 0 and aborts (this ensures that the user can only update if he possesses an old credential with $\psi(\vec{A}, \alpha) \neq \perp$). Otherwise, the issuer will output 1 after the rest of the protocol. The issuer runs $\mathsf{BlindSign}_{\mathrm{sig}}(pp, pk, sk, c)$, while the user runs $\mathsf{BlindRcv}_{\mathrm{sig}}(pp, pk, \vec{A}^*, r)$. For the user, $\mathsf{BlindRcv}_{\mathrm{sig}}$ outputs a new signature σ^*. The user checks that σ^* is valid signature $\mathsf{Vrfy}_{\mathrm{sig}}(pp, pk, \vec{A}^*, \sigma^*) \overset{!}{=} 1$. If so, he knows that σ^* is a valid credential on his expected attributes and outputs $cred^* = \sigma^*$. Otherwise, he outputs $\perp$.

Issuing a credential $(\mathsf{Issue}(cpp, pk, \psi, sk) \leftrightarrow \mathsf{Receive}(cpp, pk, \psi, \alpha))$ works similarly, but the user commits to $\psi(\perp, \alpha)$ and he omits the part about σ in the ZKAK (only proves that c is well-formed and $\psi(\perp, \alpha) \neq \perp$).

Construction 7. Let Π_{sig} be a blind signature scheme. We define an updatable credential system Π_{uacs} as described above.

A full formal description can be found in Appendix B.

Correctness and security Correctness of the above construction follows immediately from correctness of the underlying blind signature scheme Π_{sig}. For security, we have the following two theorems:

Theorem 8. If the underlying blind signature scheme has perfect message privacy (Definition 18), then Construction 7 has simulation anonymity (Definition 5).

Theorem 9. If the underlying blind signature scheme is unforgeable (Definition 17), then Construction 7 is sound (Definition 6).

The proofs of the above theorems are straight-forward reductions to the corresponding blind signature properties. They are presented in Appendix C.

5 Efficient Instantiation of UACS

Since there exist zero-knowledge arguments of knowledge for all of NP, almost arbitrary update functions are supported by this construction. Because those generic zero-knowledge arguments are not necessarily considered practically efficient, in practice one usually wants to restrict the class of update functions. For example, a large class of statements is supported by Sigma protocols (such as generalizations of Schnorr's protocol), which are very efficient (see, for example, [BBB+18]). The blind signature scheme by Pointcheval and Sanders [PS16] is a good candidate to use in conjunction with Sigma protocols. If the update function is sufficiently "simple" (i.e. the check $\psi(\vec{A}, \alpha) \overset{!}{=} \vec{A}^*$ can be efficiently implemented as a Sigma protocol), our construction is efficient.

6 Incentive Systems

In an incentive system, there are two roles: users and the provider. The provider operates a point collection system in order to incentivize certain user behavior. Users gain points for certain actions (e.g., buying something), which they later want to redeem for some bonus item (e.g., a frying pan). A user privately stores his points in a *token*. We will usually talk about multiple users and a single provider.

6.1 Structure of an Incentive System

An incentive system Π_{insy} consists of the following ppt algorithms $\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{IssuerKeyGen}$, $\mathsf{Link}, \mathsf{VrfyDs}, \mathsf{Trace}$, as well as interactive protocols $\mathsf{Issue} \leftrightarrow \mathsf{Join}$, $\mathsf{Credit} \leftrightarrow \mathsf{Earn}$, and $\mathsf{Spend} \leftrightarrow \mathsf{Deduct}$. We explain them in the following.

Setup and key generation We assume that a trusted party has already generated public parameters $pp \leftarrow \mathcal{G}(1^\lambda)$ (like in UACS). To set up an incentive system, a trusted party generates incentive-system-specific parameters $ispp \leftarrow \mathsf{Setup}(pp)$. We assume that some maximum point score v_{max} is encoded in *ispp* and that this limit is large enough never to be hit in practice.

To join the system, a provider runs $\mathsf{IssuerKeyGen}(ispp)$ to obtain a key pair (pk, sk). He publishes *pk* and distributes *sk* to all store terminals that can issue points to users (these can be, for example, distributed over multiple physical stores). For the sake of this explanation, we will distinguish the provider and individual store terminals.

When users want to join the system, they run $\mathsf{KeyGen}(ispp)$ and store the resulting key pair (upk, usk) (e.g., on their smartphone).

Obtaining a token To obtain a token with balance 0 from the provider (or store terminal), the user sends his *upk* to the provider and identifies himself (this is out of the scope of the incentive system. For example, the user could sign *upk* with some signature key stored on their digital passport). The provider associates *upk* to the user's real identity (so that in case of dispute, the user can be identified from *upk*). Then, the provider runs $\mathsf{Issue}(\mathit{ispp}, \mathit{pk}, \mathit{upk}, \mathit{sk})$ interacting with the user running $\mathsf{Join}(\mathit{ispp}, \mathit{pk}, \mathit{upk}, \mathit{usk})$. Afterwards, Join outputs a token *token* and a double-spending identifier *dsid* to the user (these are hidden from the provider). The user stores his current token *token*, its current *dsid* and value v (for this fresh token, $v = 0$, i.e. no points have been collected yet). *dsid* can be seen as a random ID for the token, which will play a role in preventing double-spending.

Earning and spending points After obtaining a token *token* from the provider, the user can start collecting points. Assume the user buys something in a store, for which he should receive k points. To grant the points, the store terminal runs $\mathsf{Credit}(\mathit{ispp}, \mathit{pk}, k, \mathit{sk})$ interacting with the user running $\mathsf{Earn}(\mathit{ispp}, \mathit{pk}, k, \mathit{usk}, \mathit{token})$. Afterwards, Earn outputs a new token token^* to the user. If $\mathit{token}^* \neq \perp$, the user deletes his old token *token* and replaces it with the new token token^* (the *dsid* does not change between *token* and token^*). If the old token *token* had the value v, then the new token has value $v + k$.

After earning enough points, the user may want to spend some of them in exchange for some reward (e.g., spend 100 points to obtain a frying pan). For this, user and the store terminal agree on a number $k \leq v$ of points to spend. Then the user reveals his *dsid* to the store terminal. The terminal keeps a local database $\mathcal{DB}_{\text{local}}$ of *dsid* it has already seen (more details later). If *dsid* is present in $\mathcal{DB}_{\text{local}}$ (meaning the user is trying to spend a token that has already been spent before), the terminal rejects the transaction. Otherwise, the store terminal runs $\mathsf{Deduct}(\mathit{ispp}, \mathit{pk}, k, \mathit{dsid}, \mathit{sk})$ interacting with the user running $\mathsf{Spend}(\mathit{ispp}, \mathit{pk}, k, \mathit{dsid}, \mathit{usk}, \mathit{token})$.

Spend outputs a new token token^* and a new dsid^* for the new token (since the old *dsid* has been revealed, token^* needs a new one). The new token holds the remainder amount $v - k$ of points left after the spend operation. The user updates its current *token*, *dsid*, and v with the new values (and deletes the old values).

Deduct outputs a bit b to the terminal and, if $b = 1$, a double-spend tag *dstag*. If $b = 0$, the transaction has failed (e.g., the user does not have enough points). In that case, the store terminal does nothing. If $b = 1$, the transaction is considered successful and the reward is given to the user. The terminal stores the transaction data together with *dsid* and *dstag* in its local database $\mathcal{DB}_{\text{local}}$. This data will be used to handle offline double-spending.

Handling offline double-spending Because the local databases $\mathcal{DB}_{\text{local}}$ of each store are not necessarily in sync (stores are not required to be always online), users can (potentially) spend the same token in two offline stores. This is because if the first store is offline, it cannot (in time) communicate to the second store that the token's *dsid* has already been spent. This way, the user may receive rewards for which he does not have sufficient points.

To deal with this, an incentive system offers the following mechanism: assume there was a spend transaction t in which a user spends his token. Associated with t are the token's id *dsid* and a tag *dstag* (as described above). If there is another spend transactions $t^\times$ that is associated with *the same dsid* and some tag $\mathit{dstag}^\times$, then double spending occurred. $t^\times$ should be considered invalid and the provider should try to undo all consequences of $t^\times$. To undo rewards gained fraudulently by the user as the result of $t^\times$, the provider would first run the algorithm $\mathsf{Link}(\mathit{ispp}, \mathit{dstag}, \mathit{dstag}^\times)$, which outputs (1) the double-spending user's public key *upk*, and (2) linking information *dslink*. With *upk*, the provider can identify the user, while *dslink* serves as publicly verifiable proof that the user has indeed double-spent. This can be verified by anyone using $\mathsf{VrfyDs}(\mathit{ispp}, \mathit{dslink}, \mathit{upk})$, which outputs a bit indicating whether or not *dslink* is a valid proof of double-spending for user *upk*. With

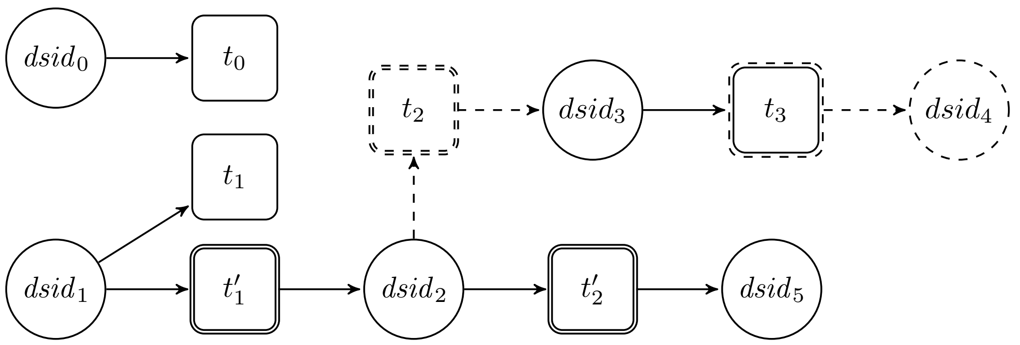


Figure 2: Example $\mathcal{DB}$. Double-struck spend operations are invalid. All dashed lines are added when t_2 is synchronized into $\mathcal{DB}$. The user has double-spent $dsid_1$ (and t_1' is marked invalid because of this). When t_2 is synchronized into $\mathcal{DB}$, it is immediately marked invalid, $dsid_3$ is revealed to be its successor and as a consequence, t_3 is marked invalid and its successor $dsid_4$ is computed.

this mechanism, the provider can recoup any losses (e.g., through a legal process).

The second consequence of transaction $t^\times$ is that the user gained a remainder token from it, which also never should have happened. *dslink* as computed above can also be used to deal with this: the incentive system provides the method $\mathsf{Trace}(ispp, dslink, dstag^\times)$ which outputs the *dsid* of the remainder token that resulted from $t^\times$. This can be iterated: if $\mathsf{Trace}(ispp, dslink, dstag^\times)$ uncovers $dsid_0$ and $dsid_0$ has also been already spent in transaction t_0 with tag $dstag_0$, then $\mathsf{Trace}(ispp, dslink, dstag_0)$ will uncover some $dsid_1$, etc.

To be more concrete, we imagine the provider sets up a central database $\mathcal{DB}$. The database is a directed bipartite graph, which contains (1) one token node *dsid* for each *dsid* the provider received, and (2) one transaction node t_i for each spend transaction. Edges establish known consume/produce relations for transactions: every transaction t_i effectively consumes exactly one *dsid* (which the user reveals), inducing an edge $dsid \to t_i$. If double-spending occurs, Trace may uncover the remainder token's $dsid^*$ produced by t_i, in which case the graph would contain an edge $t_i \to dsid^*$. Figure 2 depicts an example database $\mathcal{DB}$.

Stores periodically send their observed transactions t_i (together with their k, *dsid*, and *dstag*) to the central database. If *dsid* is not yet in the database, the database simply adds $dsid \to t_i$ to the graph. If *dsid* was already in the database, then it has already been spent in a transaction t_j. t_i is marked invalid and $\mathsf{Link}, \mathsf{Trace}$ are used to find t_i's successor $dsid^*$. If $dsid^*$ is already in the database, any transaction descendants of $dsid^*$ are marked invalid and the process repeats. The exact algorithm $\mathsf{DBsync}(k, dsid, dstag, \mathcal{DB})$ to add a transaction t_i to the graph is given in Figure 3.

If desired, the provider can periodically send all or some spent *dsid*s known in $\mathcal{DB}$ to the store terminals so that they can immediately reject future double-spending transactions. This is especially useful for $dsid^*$s that were revealed through Trace and have not been spent yet (but we already know that they will ultimately be revealed as double-spending after the next database synchronization).

6.2 Formal definition

We now formally define incentive systems. For this, we define a set of oracles with shared state that formally represent the behavior of honest parties in the processes explained above. These oracles will allow us to the correctness and security definitions (we will allow an adversary $\mathcal{A}$ to query a selection of these oracles in subsequent security games). For these definitions, we assume that *ispp* has been generated honestly.

DBsync(k, *dsid*, *dstag*, $\mathcal{DB}$):

- Add new spend operation node t_i to $\mathcal{DB}$, associate it with k, *dstag*.
- If *dsid* is not in $\mathcal{DB}$, add the node *dsid* and an edge from *dsid* to t_i.
- Otherwise, add the edge from *dsid* to t_i, and:
  - If *dsid* has no (*upk*, *dslink*) associated with it, then there exist two outgoing edges from *dsid* to transactions t_i, t_j. In this case, compute (*upk*, *dslink*) $=$ $\mathsf{Link}(\mathit{ispp}, \mathit{dstag}'_i, \mathit{dstag}'_j)$ using the two tags $\mathit{dstag}'_i, \mathit{dstag}'_j$ associated with t_i and t_j, respectively. Associate (*upk*, *dslink*) with *dsid*.
  - Mark t_i invalid (this triggers the steps below).
- Whenever some node t_i with incoming edge from some *dsid* is marked invalid
  - Use (*upk*, *dslink*) associated with *dsid* and *dstag* associated to t_i to compute $\mathit{dsid}^* = \mathsf{Trace}(\mathit{ispp}, \mathit{dslink}, \mathit{dstag})$. Add dsid^* to the graph (if it does not already exist), associate (*upk*, *dslink*) with dsid^*, and add an edge from t_i to dsid^*. If there is an edge from dsid^* to some t_j, mark t_j invalid (if it was not already marked). This triggers this routine again.

Figure 3: **DBsync** algorithm

Honest users To model honest users, we define the following oracles:

- **Keygen**() chooses a new user handle u, generates key pair $(\mathit{upk}, \mathit{usk}) \leftarrow \mathsf{KeyGen}(\mathit{ispp})$, and stores for reference $(\mathit{upk}_u, \mathit{usk}_u, v_u, \mathit{pk}_u, \mathit{token}_u, \mathit{dsid}_u) \leftarrow (\mathit{upk}, \mathit{usk}, 0, \perp, \perp, \perp)$. It outputs u, *upk*.
- **Join**(u, pk) given handle u runs $(\mathit{token}, \mathit{dsid}) \leftarrow \mathsf{Join}(\mathit{ispp}, \mathit{pk}, \mathit{upk}_u, \mathit{usk}_u)$. If $\mathit{token} = \perp$, the oracle outputs $\perp$. Otherwise, it stores $\mathit{pk}_u \leftarrow \mathit{pk}$, $\mathit{token}_u \leftarrow \mathit{token}$, and $\mathit{dsid}_u \leftarrow \mathit{dsid}$. This oracle can only be called once for each u. It must be called before any calls to **Earn**$(u, \cdot)$ and **Spend**$(u, \cdot)$.
- **Earn**(u, k) given handle u and $k \in \mathbb{N}$ with $v_u + k \leq v_{max}$, the oracle runs $\mathit{token}^* \leftarrow \mathsf{Earn}(\mathit{ispp}, \mathit{pk}_u, k, \mathit{usk}_u, \mathit{token}_u)$. If $\mathit{token}^* = \perp$, the oracle outputs $\perp$. Otherwise, it updates $\mathit{token}_u \leftarrow \mathit{token}^*$ and $v_u \leftarrow v_u + k$.
- **Spend**(u, k) given handle u and $k \in \mathbb{N}$ with $v_u \geq k$, the oracle first sends dsid_u to its communication partner and then runs $(\mathit{token}^*, \mathit{dsid}^*) \leftarrow \mathsf{Spend}(\mathit{ispp}, \mathit{pk}_u, k, \mathit{dsid}_u, \mathit{usk}_u, \mathit{token}_u)$. It updates $\mathit{token}_u \leftarrow \mathit{token}^*$, $\mathit{dsid}_u \leftarrow \mathit{dsid}^*$ and $v_u \leftarrow v_u - k$. If $\mathit{token}^* = \perp$, the oracle outputs $\perp$ and all further calls to any oracles concerning u are ignored.[1]

Honest Provider To model an honest provider, we define the following oracles:

- **IssuerKeyGen**() generates $(\mathit{pk}, \mathit{sk}) \leftarrow \mathsf{IssuerKeyGen}(\mathit{ispp})$. It stores *pk* and *sk* for further use. It initially sets the set of users $\mathcal{U} \leftarrow \emptyset$ and sets the double-spend database $\mathcal{DB}$ to the empty graph. Furthermore, initially $v_{\text{earned}}, v_{\text{spent}} \leftarrow 0$. Further calls to this oracle are ignored. This oracle must be called before any of the other provider-related oracles. The oracle outputs *pk*.
- **Issue**(*upk*) if $\mathit{upk} \in \mathcal{U}$, the request is ignored. Otherwise, the oracle runs $\mathsf{Issue}(\mathit{ispp}, \mathit{pk}, \mathit{upk}, \mathit{sk})$ and adds *upk* to $\mathcal{U}$.
- **Credit**(k) for $k \in \mathbb{N}$, runs $\mathsf{Credit}(\mathit{ispp}, \mathit{pk}, k, \mathit{sk})$ and sets value $v_{\text{earned}} \leftarrow v_{\text{earned}} + k$.

[1]Spending the same *token* twice would be considered double-spending, even if one of the Spend operations fails. Hence after a failed Spend operation, the user must not attempt to use her old *token*.

- **Deduct**(k) for $k \in \mathbb{N}$, waits to receive *dsid*. It then runs algorithm $\mathsf{Deduct}(\mathit{ispp}, \mathit{pk}, k, \mathit{dsid}, \mathit{sk}) \to (b, \mathit{dstag})$. If $b = 0$, it outputs $\perp$. Otherwise, it chooses a fresh spend handle s and stores $(\mathit{dsid}_s, \mathit{dstag}_s, k_s) \leftarrow (\mathit{dsid}, \mathit{dstag}, k)$. Then it outputs s and increments $v_{\text{spent}} \leftarrow v_{\text{spent}} + k$.
- **DBsync**(s) runs $\mathcal{DB}' \leftarrow \mathsf{DBsync}((\mathit{dsid}_s, \mathit{dstag}_s, k_s), \mathcal{DB})$. Then, it updates $\mathcal{DB} \leftarrow \mathcal{DB}'$ and recomputes v_{invalid} as the sum of values k associated with invalid transactions within $\mathcal{DB}'$.

Syntax and correctness We are now prepared to define incentive systems and their correctness. For this, consider an experiment $\mathrm{Exp}^{\mathrm{correct}}(\Pi_{\mathrm{insy}}, \mathcal{A}, \lambda)$, where $\mathcal{A}$ is an algorithm. The experiment first runs **IssuerKeyGen** to receive *pk*. The adversary $\mathcal{A}$ is then given *pk* and access to the following oracles (see Section 2 for oracle notation): **Keygen**(), $u \mapsto$ **Issue**$(\mathit{upk}_u) \leftrightarrow$ **Join**(u, pk), $(u, k) \mapsto$ **Earn**$(u, k) \leftrightarrow$ **Credit**(k), $(u, k) \mapsto$ **Spend**$(u, k) \leftrightarrow$ **Deduct**(k), and $s \mapsto$ **DBsync**(s). The experiment outputs `fail` if something goes wrong, i.e. if one of the oracles outputs $\perp$ or if $\mathcal{DB}$ contains a transaction marked as invalid. Note that in this experiment, all protocols are followed honestly and $\mathcal{A}$ effectively just chooses a polynomial-length sequence of actions that users or the provider take.

Definition 10 (Incentive System)**.** An incentive system Π_{insy} consists of ppt algorithms Setup, KeyGen, IssuerKeyGen, Issue, Join, Credit, Earn, Spend, Deduct, Link, VrfyDs, Trace, where Link, VrfyDs, and Trace are deterministic. It is *correct* if for all ppt $\mathcal{A}$, there exists a negligible function *negl* such that

$$\Pr[\mathrm{Exp}^{\mathrm{correct}}(\Pi_{\mathrm{insy}}, \mathcal{A}, \lambda) = \mathtt{fail}] \leq \mathit{negl}(\lambda). \qquad \diamond$$

6.3 Security Definitions of Incentive Systems

With regards to security, an incentive system should protect honest users' privacy. The provider wants to be sure that users cannot spend more points than the provider issued. If they do (e.g., in offline stores), the provider needs to be able to uncover all illegal transactions and prove the double-spending user's guilt. We will now define these properties formally.

6.3.1 Anonymity

For anonymity, we want that a malicious provider is unable to learn which user belongs to which earn/spend transaction. In reality, this protects users, for example, from having their shopping history linked to their identity. Users are *not* anonymous when registering for the incentive system (Join) because the provider needs to learn their real identity to identify double-spending users. However, if users are honest and do not double-spend, the provider should not be able to link a user's registration to any other action they do.

More formally, a malicious provider should not be able to distinguish two users running the Earn or the Spend protocol. We define this with a game-based approach: we define two experiments, $\mathrm{Exp}^{\mathsf{ano\text{-}Earn}}$ and $\mathrm{Exp}^{\mathsf{ano\text{-}Spend}}$, which treat anonymity for the Earn and Spend operation, respectively (cf. Figure 4). In the first phase of the experiments, the adversary $\mathcal{A}$ plays the role of a malicious provider: $\mathcal{A}$ publishes some public key *pk* and interacts with honest users. Note that by design of the honest user oracles, honest users never double-spend. $\mathcal{A}$ then chooses two users u_0, u_1. The experiment makes one of the users run Earn (or Spend) with $\mathcal{A}$. $\mathcal{A}$ should not be able to distinguish u_0 running Earn (or Spend) from u_1 running the protocol.

There are two exceptions to this: First, if u_0 has a valid token $\mathit{token}_{u_0} \neq \perp$, while u_1 does not (e.g., because the provider sabotaged an earlier spend operation), then certainly u_0 can be distinguished from u_1. Second, if u_0 or u_1 do not have sufficiently many points to spend k, or, analogously, if u_0 or u_1 has too many points to receive k additional points without hitting the v_{max} limit. Since it is functionally desired to be able to distinguish users in these cases, the experiment accounts for this.

```
$\mathrm{Exp}_b^{\text{ano-X}}(\Pi_{\text{insy}}, \mathcal{A}, \lambda)$:
  $ispp \leftarrow \mathsf{Setup}(\mathcal{G}(1^\lambda))$
  $(pk, st) \leftarrow \mathcal{A}(ispp)$
  $(u_0, u_1, k, st) \leftarrow \mathcal{A}^{\mathbf{Keygen}(),\mathbf{Join}(\cdot,pk),\mathbf{Earn}(\cdot,\cdot),\mathbf{Spend}(\cdot,\cdot)}(st)$
  **If** $\bot \in \{token_{u_0}, token_{u_1}\}$, **output** 0
  **If** $X = \mathsf{Earn}$ **and** $v_{u_0}, v_{u_1} \leq v_{max} - k$
    **output** $\hat{b} \leftarrow \mathcal{A}^{\mathbf{Earn}(u_b,k)}(st)$, where $\mathcal{A}$ may only query once
  **Else if** $X = \mathsf{Spend}$ **and** $v_{u_0}, v_{u_1} \geq k$
    **output** $\hat{b} \leftarrow \mathcal{A}^{\mathbf{Spend}(u_b,k)}(st)$, where $\mathcal{A}$ may only query once
  **Else**
    **output** 0
```

Figure 4: Anonymity experiments

```
$\mathrm{Exp}^{\text{fram-res}}(\Pi_{\text{insy}}, \mathcal{A}, \lambda)$:
  $ispp \leftarrow \mathsf{Setup}(\mathcal{G}(1^\lambda))$
  $(pk, st) \leftarrow \mathcal{A}(ispp)$
  $(u, dslink) \leftarrow \mathcal{A}^{\mathbf{Keygen}(),\mathbf{Join}(\cdot,pk),\mathbf{Earn}(\cdot,\cdot),\mathbf{Spend}(\cdot,\cdot)}(st)$
  **If** $\mathsf{VrfyDs}(ispp, dslink, upk_u) = 1$
    **output** 1
  **Else output** 0
```

Figure 5: Framing resistance experiment

Definition 11 (Anonymity)**.** The experiment $\mathrm{Exp}^{\text{ano-X}}$ is presented in Fig. 4 and defined for $X \in \{\mathsf{Earn}, \mathsf{Spend}\}$. We say that an incentive system Π_{insy} is *anonymous* if for both $X \in \{\mathsf{Earn}, \mathsf{Spend}\}$ and for all ppt $\mathcal{A}$ it holds that

$$|\Pr[\mathrm{Exp}_0^{\text{ano-X}}(\Pi, \mathcal{A}, \lambda) = 1] - \Pr[\mathrm{Exp}_1^{\text{ano-X}}(\Pi, \mathcal{A}, \lambda) = 1]| \leq negl(\lambda)$$

for all λ. ◇

6.3.2 Framing resistance

To deal with double-spending users, the provider wants to be able to convincingly accuse users of double-spending. Framing resistance guarantees that honest users cannot be falsely accused of double-spending by a malicious provider. This is a positive for honest users (as they can repudiate double-spending claims) and for the provider (a double-spending proof holds more weight in court if the provider cannot possibly frame innocent users).

We define framing resistance with an experiment, in which the adversary $\mathcal{A}$ plays the role of a malicious provider, who publishes some pk and interacts with honest users (which by definition do not double-spend). Ultimately, $\mathcal{A}$ tries to compute a value $dslink$ that is accepted by VrfyDs as proof of double-spending for some honest user. The chances of him succeeding must be negligible.

Definition 12 (Framing resistance)**.** We define experiment $\mathrm{Exp}^{\text{fram-res}}$ in Fig. 5. We say that incentive system Π_{insy} is *framing resistant* if for all ppt $\mathcal{A}$, there exists a negligible function $negl$ s.t.

```
$\mathrm{Exp}^{\mathrm{sound}}(\Pi_{\mathrm{insy}}, \mathcal{A}, \lambda)$:
    $ispp \leftarrow \mathsf{Setup}(\mathcal{G}(1^\lambda))$
    $pk \leftarrow$ **IssuerKeyGen**()
    **Run** $\mathcal{A}^{\mathbf{Issue}(\cdot),\mathbf{Credit}(\cdot),\mathbf{Deduct}(\cdot),\mathbf{DBsync}(\cdot)}(ispp, pk)$
    **If** $v_{\mathrm{spent}} - v_{\mathrm{invalid}} > v_{\mathrm{earned}}$ **and** $\mathcal{A}$ has queried **DBsync**($s$) for all
    spending record handles $s$ output by the **Deduct** oracle
        **output** 1
    **If** $\mathcal{DB}$ contains some $(upk, dslink)$ associated with some $dsid$
    such that $\mathsf{VrfyDs}(ispp, dslink, upk) \neq 1$ **or** $upk \notin \mathcal{U}$,
        **output** 1
    **Else output** 0
```

Figure 6: Soundness experiment

$\Pr[\mathrm{Exp}^{\text{fram-res}}(\Pi, \mathcal{A}, \lambda) = 1] \leq negl(\lambda)$ for all λ. ◇

6.3.3 Soundness

For soundness, we ideally want to ensure that malicious users cannot spend more points than the honest provider has issued. Of course, in the presence of offline double-spending, this statement cannot be true: users can certainly spend their tokens twice in offline stores. Hence we need to be more precise. We keep count of three kinds of points (as can be seen in the oracle definitions on page 16): v_{earned} counts how many points the provider has issued. v_{spent} counts how many points users have spent (in the sense of successful Deduct runs). v_{invalid} counts how many points were spent in transactions that have been marked invalid in the provider's database $\mathcal{DB}$. Soundness will guarantee that $v_{\mathrm{spent}} - v_{\mathrm{invalid}} \leq v_{\mathrm{earned}}$, i.e. users cannot spend more then they have earned *if you deduct transactions the provider discovers to be invalid.* This means that while users may be able to double-spend, they cannot do so undetected.

Furthermore, soundness guarantees that double-spending transactions can be traced to users, i.e. whenever $\mathcal{DB}$ contains some upk, $dslink$ annotated to some double-spent token node $dsid$, then upk is one of the registered users and $dslink$ is valid proof of double-spending.

The experiment has an adversary $\mathcal{A}$ play the role of an arbitrary number of malicious users, while the experiment simulates an honest provider. $\mathcal{A}$ can interact with the honest provider for the usual user operations. Additionally, $\mathcal{A}$ can control the order in which his transaction data is added to the central database $\mathcal{DB}$. $\mathcal{A}$ wins if either $v_{\mathrm{spent}} - v_{\mathrm{invalid}} > v_{\mathrm{earned}}$ (even though all transactions have been synchronized to $\mathcal{DB}$, i.e. all offline double-spending detection should have already happened) or if the database holds some invalid tracing data upk, $dslink$.

Definition 13 (Soundness)**.** We define the experiment $\mathrm{Exp}^{\mathrm{sound}}$ in *Fig.* 6. We say that incentive system Π_{insy} is *sound* if for all ppt $\mathcal{A}$, there exists a negligible function *negl* with $\Pr[\mathrm{Exp}^{\mathrm{sound}}(\Pi, \mathcal{A}, \lambda) = 1] \leq negl(\lambda)$ for all λ. ◇

7 Construction of an Incentive System from UACS

For our construction of an incentive system, we use a UACS Π_{uacs} (Definition 4), a public-key encryption scheme Π_{enc} (Definition 2), and an additively malleable commitment scheme Π_{cmt} (Definition 3). At its core, the users' tokens will be credentials encoding their points. They are updated whenever the user earns or spends points. Most of the other mechanisms in place deal with double-spending prevention, as we'll explain below.

Key generation The system is set up using $\mathsf{Setup}(pp)$, which outputs $ispp = (pp, cpp, pk_{\text{cmt}})$ consisting of public parameters pp (e.g., the elliptic curve group), credential public parameters $cpp \leftarrow \mathsf{Setup}_{\text{uacs}}(pp)$, and a commitment key $pk_{\text{cmt}} \leftarrow \mathsf{KeyGen}_{\text{cmt}}(pp)$. pp fixes an attribute space $\mathbb{A}$ for the credential system and a message space $\mathcal{M}_{\text{enc}}$ for the encryption scheme. We assume $\mathbb{A} = \mathbb{Z}_p$ for some super-poly p and set the point maximum to $v_{max} = p - 1$.

The key pair $(pk, sk) \leftarrow \mathsf{IssuerKeyGen}(ispp)$ for a provider is simply a credential key pair $(pk, sk) \leftarrow \mathsf{IssuerKeyGen}_{\text{uacs}}(cpp, 1^n)$ for $n = 4$ (i.e. all our attribute vectors will have length 4). They will use sk to issue and update credentials.

Users generate a key pair $(upk, usk) \leftarrow \mathsf{KeyGen}(ispp)$, which is simply an encryption key pair, i.e. $usk \leftarrow \mathsf{KeyGen}_{\text{enc}}(pp)$ and $upk = \mathsf{ComputePK}_{\text{enc}}(pp, usk)$. As a rough idea, the user's key will be used to (1) identify the user, and (2) encrypt tracing data. If double-spending occurs, our mechanisms will ensure that usk is revealed, allowing the provider to access the tracing data.

Obtaining a token A token $token = (dsid, dsrnd, v, cred)$ consists of its identifier $dsid$, some randomness $dsrnd$ used for double-spending protection, its current value v, and a credential $cred$ with attributes $(usk, dsid, dsrnd, v)$.

The provider wants the $dsid$ for each token to be uniformly random, so that users cannot maliciously provoke $dsid$ collisions (which, at first glance, would not actually benefit the user. However, $dsid$ collisions between otherwise unrelated tokens would hinder tracing). Furthermore, the provider should not be able to learn $dsid$, because he would otherwise be able to recognize the user when he spends the token and reveals $dsid$. For this reason, the first step of the token-obtaining protocol $\mathsf{Issue}(ispp, pk, upk, sk) \leftrightarrow \mathsf{Join}(ispp, pk, upk, usk) \rightarrow (token, dsid)$ has the user and provider compute a commitment to $dsid$ such that $dsid$ is guaranteed to be uniformly random in $\mathbb{Z}_p$ if either the user or the provider is honest, and only the user knows $dsid$. To ensure this, both parties contribute a random share for $dsid$. The user privately picks a random share $dsid_{\text{usr}} \leftarrow \mathbb{Z}_p$, and the provider does the same for his share $dsid_{\text{prvdr}} \leftarrow \mathbb{Z}_p$. Then the user commits to his share $(C_{\text{usr}}, open) \leftarrow \mathsf{Commit}_{\text{cmt}}(pp, pk_{\text{cmt}}, dsid_{\text{usr}})$ and sends the commitment C_{usr} to the provider. The provider replies with his share $dsid_{\text{prvdr}} \leftarrow \mathbb{Z}_p$ (in plain). Using additive malleability of the commitment scheme, both parties can compute the commitment $C_{\text{dsid}} = \mathsf{Add}_{\text{cmt}}(pp, pk_{\text{cmt}}, C_{\text{usr}}, dsid_{\text{prvdr}})$ to $dsid := dsid_{\text{usr}} + dsid_{\text{prvdr}}$. Intuitively, if the provider is honest, then $dsid_{\text{prvdr}}$ is uniformly random, and hence $dsid$ is random. If the user is honest, $dsid_{\text{usr}}$ is uniformly random and hidden within the commitment, so the provider will not be able to choose $dsid_{\text{prvdr}}$ adaptively, hence overall, $dsid$ in that case should also be uniformly random (and hidden from the provider).

Now the provider issues a credential to the user. For this, the user's hidden parameter is $\alpha = (usk, dsid, dsrnd, open)$, where the user privately chooses $dsrnd \leftarrow \mathbb{Z}_p$. The update function is

$$\psi(\perp, \alpha) = \begin{cases} (usk, dsid, dsrnd, 0) & \text{if } \psi_{\text{chk}} \\ \perp & \text{otherwise} \end{cases}$$

where ψ_{chk} is true if and only if

- The user secret to be written into the credential is consistent with the user's public key ($upk = \mathsf{ComputePK}_{\text{enc}}(pp, usk)$), and
- $dsid$ is committed ($\mathsf{Vrfy}_{\text{cmt}}(pp, pk_{\text{cmt}}, C_{\text{dsid}}, dsid, open) = 1$).

The two parties run $\mathsf{Issue}_{\text{uacs}}(cpp, pk, \psi, sk) \leftrightarrow \mathsf{Receive}_{\text{uacs}}(cpp, pk, \psi, \alpha) \rightarrow cred$, where the user receives his credential $cred$. The user outputs his token $token = (dsid, dsrnd, v = 0, cred)$.

Earning points The protocol $\mathsf{Credit}(\mathit{ispp}, \mathit{pk}, k, \mathit{sk}) \leftrightarrow \mathsf{Earn}(\mathit{ispp}, \mathit{pk}, k, \mathit{usk}, \mathit{token})$, where the user receives a new token token^* with value $v + k$ is very simple: the provider and the user simply run a credential update that adds k to v, i.e. with update function $\psi((\mathit{usk}, \mathit{dsid}, \mathit{dsrnd}, v), \cdot) = (\mathit{usk}, \mathit{dsid}, \mathit{dsrnd}, v + k)$. The user stores the new token $\mathit{token}^* = (\mathit{dsid}, \mathit{dsrnd}, v + k, \mathit{cred}^*)$, where cred^* is the result of the credential update.

Spending points Spending $k \leq v$ points of a token $\mathit{token} = (\mathit{dsid}, \mathit{dsrnd}, v, \mathit{cred})$ is the most complicated operation, as most of the double-spending protection happens here. Before the protocol, the user reveals dsid, which (with overwhelming probability) uniquely identifies token. Then, the parties run $\mathsf{Spend}(\mathit{ispp}, \mathit{pk}, k, \mathit{dsid}, \mathit{usk}, \mathit{token}) \leftrightarrow \mathsf{Deduct}(\mathit{ispp}, \mathit{pk}, k, \mathit{dsid}, \mathit{sk})$, which works as follows:

For the remainder token, the user and the provider jointly compute a commitment ${C_{\mathrm{dsid}}}^*$ on a guaranteed random dsid^* as in the "Obtaining a token" protocol. From this, the user obtains ${C_{\mathrm{dsid}}}^*$, dsid^*, open^* and the provider obtains ${C_{\mathrm{dsid}}}^*$.

To enable the provider to reveal the user's identity in case of double-spending, the provider sends a random challenge $\gamma \leftarrow \mathbb{Z}_p$ to the user and the user replies with $c = \mathit{usk} \cdot \gamma + \mathit{dsrnd}$ (using dsrnd of the token he's spending). Intuitively, if this is the first time this token is spent, dsrnd is uniformly random and the provider has never seen the value, hence to the provider, c is just some uniformly random value. The idea is that if the user tries to spend the same token a second time, he will be forced to reveal $c' = \mathit{usk} \cdot \gamma' + \mathit{dsrnd}$ for some (likely different) challenge γ', from which the provider would be able to compute $\mathit{usk} = (c - c')/(\gamma - \gamma')$, clearly identifying the user.

In case the user double-spends, the provider needs to be able to find the dsid^* of the remainder token that is going to be issued. To enable this, the user encrypts dsid^* under his own public key: $\mathit{ctrace} \leftarrow \mathsf{Encrypt}_{\mathrm{enc}}(\mathit{pp}, \mathit{upk}, \mathit{dsid}^*)$ and sends ctrace to the provider. The idea is that if usk is ever revealed because the user double-spends, then the provider can use it to decrypt ctrace and uncover the remainder token's identifier dsid^*.

Finally, user and provider run a credential update on the user's current token to become the remainder token. This includes a check whether or not the data sent by the user is formed correctly. The user's hidden parameter is $\alpha = (\mathit{dsid}^*, \mathit{dsrnd}^*, \mathit{open}^*)$, where dsid^*, and open^* are as above and the user secretly chooses $\mathit{dsrnd}^* \leftarrow \mathbb{Z}_p$. The update function is

$$\psi((\mathit{usk}, \mathit{dsid}, \mathit{dsrnd}, v), \alpha) = \begin{cases} (\mathit{usk}, \mathit{dsid}^*, \mathit{dsrnd}^*, v - k) \text{ if } \psi_{\mathrm{chk}} \\ \perp \text{ otherwise} \end{cases}$$

where ψ_{chk} is true if and only if

- dsid in the credential is the same as the user has revealed to the provider,
- the user has sufficient points, i.e. $v \geq k$,
- the commitment to dsid^* is well-formed, i.e. $1 = \mathsf{Vrfy}_{\mathrm{cmt}}(\mathit{pp}, \mathit{pk}_{\mathrm{cmt}}, {C_{\mathrm{dsid}}}^*, \mathit{dsid}^*, \mathit{open}^*)$,
- c is well-formed to reveal usk upon double-spending, i.e. $c = \mathit{usk} \cdot \gamma + \mathit{dsrnd}$, and
- ctrace can be decrypted with usk, i.e. $\mathit{dsid}^* = \mathsf{Decrypt}_{\mathrm{enc}}(\mathit{pp}, \mathit{usk}, \mathit{ctrace})$.

From the credential update, the user receives a credential cred^*. If $\mathit{cred}^* \neq \perp$, Spend outputs $\mathit{token}^* = (\mathit{dsid}^*, \mathit{dsrnd}^*, v - k, \mathit{cred}^*)$ and its dsid^*. The provider receives a bit b from the update, intuitively indicating whether or not the user had sufficient points and that the data he sent will enable correct tracing of the user and this transaction. Deduct outputs b and, if $b = 1$, the double-spending tag $\mathit{dstag} = (c, \gamma, \mathit{ctrace})$.

Handling offline double-spending Tracing double-spending users and their *dsid*s works as follows:

For two tags $\mathit{dstag} = (c, \gamma, \mathit{ctrace} = \mathit{usk} \cdot \gamma + \mathit{dsrnd})$ and $\mathit{dstag}' = (c', \gamma', \mathit{ctrace}' = \mathit{usk} \cdot \gamma' + \mathit{dsrnd})$, we can compute $\mathsf{Link}(\mathit{ispp}, \mathit{dstag}, \mathit{dstag}') = (\mathit{upk}, \mathit{dslink})$ as $\mathit{dslink} = \mathit{usk} = (c - c')/(\gamma - \gamma')$, and $\mathit{upk} = \mathsf{ComputePK}(\mathit{pp}, \mathit{dslink})$.

Then, *dslink* can be used to trace that user's transactions by decrypting *ctrace* as $\mathsf{Trace}(\mathit{ispp}, \mathit{dslink}, \mathit{dstag}) := \mathsf{Decrypt}_{\mathrm{enc}}(\mathit{pp}, \mathit{dslink}, \mathit{ctrace}) = \mathit{dsid}^*$.

Finally, clearly we can establish a user's guilt by revealing the secret key $\mathit{usk} = \mathit{dslink}$ to his public key *upk*. To verify a user's key, $\mathsf{VrfyDs}(\mathit{ispp}, \mathit{dslink}, \mathit{upk})$ checks that $\mathsf{ComputePK}(\mathit{pp}, \mathit{dslink}) = \mathit{upk}$.

Correctness A more compact representation of this construction can be found in Appendix D. It is easy to check correctness given that *dsid*s are by definition uniformly random in $\mathbb{Z}_p$ if both user and provider behave honestly.

7.1 Security

We state the following theorems:

Theorem 14. If Π_{uacs} has simulation anonymity (Definition 5), Π_{enc} is key-ind. CPA secure (Definition 19), Π_{cmt} is computational hiding, then Π_{insy} (Construction 23) guarantees *anonymity* (Definition 11).

Proof sketch. The adversary $\mathcal{A}$ is asked to distinguish if it talks to user u_0 or u_1 in the challenge phase. Both users are determined by $\mathcal{A}$. We will first handle the easy case of $\mathrm{Exp}_b^{\text{ano-X}}(\Pi_{\mathrm{insy}}, \mathcal{A}, \lambda)$ for $X = \mathsf{Earn}$: everything that the adversary $\mathcal{A}$ sees perfectly hides the user's secret *usk* and *dsid*. For the case $X = \mathsf{Spend}$ and user u_b, let i be the spend operation in the challenge phase and $i-1$ the previous spend operation in the setup phase. During spend $i-1$, the adversary $\mathcal{A}$ gets $\mathsf{Encrypt}_{\mathrm{enc}}(\mathit{pp}, \mathit{upk}_b, \mathit{dsid}_i)$ and can compute $\mathsf{Commit}_{\mathrm{cmt}}(\mathit{pp}, \mathit{pk}_{\mathrm{cmt}}, \mathit{dsid}_i)$ from the commitment to $\mathit{dsid}_{\mathrm{usr}}$ that he receives. In spend i, $\mathcal{A}$ gets (1) $\mathsf{Encrypt}_{\mathrm{enc}}(\mathit{pp}, \mathit{upk}_b, \mathit{dsid}_{i+1})$, (2) $\mathsf{Commit}_{\mathrm{cmt}}(\mathit{pp}, \mathit{pk}_{\mathrm{cmt}}, \mathit{dsid}_{i+1})$, and (3) dsid_i. For (2), observe that $\mathsf{Commit}_{\mathrm{cmt}}(\mathit{pp}, \mathit{pk}_{\mathrm{cmt}}, \mathit{dsid}_{i+1})$ has no influence on $\mathcal{A}$'s advantage since it is independent of b. If we look at (1), we observe that the encryption is generated under upk_b. Therefore, in addition to CPA security, we need that the keys of the users are indistinguishable. Considering (3), observe that the commitment to dsid_i (in spend $i-1$) is computationally hiding. Furthermore, to link $\mathsf{Encrypt}_{\mathrm{enc}}(\mathit{pp}, \mathit{upk}_b, \mathit{dsid}_i)$ or $\mathsf{Commit}_{\mathrm{cmt}}(\mathit{pp}, \mathit{pk}_{\mathrm{cmt}}, \mathit{dsid}_i)$ from spend $i-1$ to dsid_i revealed in spend i, $\mathcal{A}$ has to break (key-ind.) CPA security of Π_{enc} or comp. hiding of Π_{cmt}.

The full proof can be found in Appendix F.2.

Theorem 15. If Π_{uacs} is sound (Definition 6), Π_{cmt} is perfectly binding (Definition 20), and $\mathsf{ComputePK}_{\mathrm{enc}}(\mathit{pp}, \cdot)$ is injective, then the incentive system Π_{insy} (Construction 23) is *sound* (Definition 13).

Proof sketch. The proof is a reduction to soundness of the underlying updatable credential system. Let $\mathcal{A}$ be an attacker against incentive system soundness. We construct $\mathcal{B}$. $\mathcal{B}$ simulates $\mathcal{A}$'s view perfectly by replacing $\mathsf{Issue}_{\mathrm{uacs}}$ and $\mathsf{Update}_{\mathrm{uacs}}$ calls with calls to the corresponding UACS oracles. Let error be the event that (1) $\mathcal{B}$ has output the same challenge δ in two different **Deduct** runs, or (2) there were two commitments $C_{\mathrm{dsid}}, {C_{\mathrm{dsid}}}'$ in two runs of **Deduct** or **Issue** such that the commitments can be opened to two different messages. (1) happens with negligible probability ($\delta \leftarrow \mathbb{Z}_p$), so does (2) because $\mathit{dsid}_{\mathrm{prvdr}}, \mathit{dsid}^*_{\mathrm{prvdr}} \leftarrow \mathbb{Z}_p$ and the commitment scheme Π_{cmt} is perfectly binding. It then remains to show that if error does not happen and there exists a consistent explanation list $\mathcal{L}$, then

Table 1: Avg. performance of our implementation over 100 runs in milliseconds. Emphasized: typical execution platform for each algorithm.

Device	Issue	Join	Credit	Earn	Deduct	Spend
Google Pixel (Phone, Snapdragon 821)	56	**76**	122	**110**	353	**390**
Surface Book 2 (Laptop, i7-8650U)	**10**	13	**17**	18	**64**	69

$\mathcal{A}$ cannot win (implying that unless error, if $\mathcal{A}$ wins, then $\mathcal{B}$ wins as there is no $\mathcal{L}$ that would make it lose). The proof of this is somewhat technical, but essentially, we look at each user individually. For this user, there exists some "canonical" sequence of spend and earn operations in $\mathcal{L}$ that does not involve any spend operations marked as invalid (in the double-spending database $\mathcal{DB}$). From the design of update functions and consistency of $\mathcal{L}$, it is clear that in such a sequence, the value accumulated by earn operations cannot be smaller than the value spent through spend operations, i.e. the desired property $v_{\text{earned}} \geq v_{\text{spent}} - v_{\text{invalid}}$ holds if we only consider these canonical operations. The rest of the proof deals with ensuring that spend operations that are not part of the canonical sequence have all been marked invalid in $\mathcal{DB}$ (such that removing all non-canonical operations from consideration does not change $v_{\text{spent}} - v_{\text{invalid}}$ and only decreases v_{earned}). Because of $\neg$error, challenges γ do not repeat and any two attribute-vectors that share the same *dsid* have the same *usk*, *dsrnd*. This implies that extracting *usk* from two transactions with the same *dsid* works without error (given $c = \mathit{usk} \cdot \gamma + \mathit{dsrnd}$ and the definition of Link). Since any extracted *usk* is correct in this sense, the tracing of *dsid* as in DBsync works as intended, i.e. all invalid transactions will be marked as such in $\mathcal{DB}$ as required.

The full proof can be found in Appendix F.3.

Theorem 16. If encryption scheme Π_{enc} is CPA-secure and Π_{uacs} has simulation anonymity, then Π_{insy} (Construction 23) is framing resistant.

Framing resistance follows easily via reduction to Π_{enc}'s (key-ind.) CPA security: An adversary who can frame an honest user needs to be able to compute the secret key *usk* for the user's $\mathit{upk} = \mathsf{ComputePK}_{\text{enc}}(\mathit{pp}, \mathit{usk})$. The proof can be found in Appendix F.4

8 Instantiation and Performance of our Incentive System

We instantiated Construction 23 using the signature scheme by Pointcheval and Sanders [PS16] for the UACS, and ElGamal as the public-key encryption scheme and malleable commitment. A concrete description of the instantiated scheme can be found in Appendix E. Using the open-source Java library `upb.crypto` and the bilinear group provided by `mcl` (bn256)[2] we implemented this instantiation and ran it on a phone (typical user device) and a laptop (approximate provider device). In Table 1 we focus on the execution time (in ms) of the protocols, excluding communication cost. The numbers illustrate that our scheme is practical. They are better or comparable to the BBA+ performance [HHNR17], who do not offer partial spending (the user needs to reveal point count when spending) and hence can avoid expensive range proofs.

References

[Ame19] American Express Company. American express membership rewards. `https://global.americanexpress.com/rewards`, January 2019.

[2]upb.crypto: `https://github.com/upbcuk`. mcl: `https://github.com/herumi/mcl`

[BB18] Johannes Blömer and Jan Bobolz. Delegatable attribute-based anonymous credentials from dynamically malleable signatures. In Bart Preneel and Frederik Vercauteren, editors, *ACNS 18*, volume 10892 of *LNCS*, pages 221–239. Springer, Heidelberg, July 2018.

[BBB$^+$18] Kai Bemmann, Johannes Blömer, Jan Bobolz, Henrik Bröcher, Denis Diemert, Fabian Eidens, Lukas Eilers, Jan Haltermann, Jakob Juhnke, Burhan Otour, Laurens Porzenheim, Simon Pukrop, Erik Schilling, Michael Schlichtig, and Marcel Stienemeier. Fully-featured anonymous credentials with reputation system. In *ARES*, pages 42:1–42:10. ACM, 2018.

[BCKL08] Mira Belenkiy, Melissa Chase, Markulf Kohlweiss, and Anna Lysyanskaya. P-signatures and noninteractive anonymous credentials. In Ran Canetti, editor, *TCC 2008*, volume 4948 of *LNCS*, pages 356–374. Springer, Heidelberg, March 2008.

[CCs08] Jan Camenisch, Rafik Chaabouni, and abhi shelat. Efficient protocols for set membership and range proofs. In Josef Pieprzyk, editor, *ASIACRYPT 2008*, volume 5350 of *LNCS*, pages 234–252. Springer, Heidelberg, December 2008.

[CDD17] Jan Camenisch, Manu Drijvers, and Maria Dubovitskaya. Practical UC-secure delegatable credentials with attributes and their application to blockchain. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 683–699. ACM Press, October / November 2017.

[CGH11] Scott E. Coull, Matthew Green, and Susan Hohenberger. Access controls for oblivious and anonymous systems. *ACM Trans. Inf. Syst. Secur.*, 14(1):10:1–10:28, 2011.

[CHL05] Jan Camenisch, Susan Hohenberger, and Anna Lysyanskaya. Compact e-cash. In Ronald Cramer, editor, *EUROCRYPT 2005*, volume 3494 of *LNCS*, pages 302–321. Springer, Heidelberg, May 2005.

[CKS10] Jan Camenisch, Markulf Kohlweiss, and Claudio Soriente. Solving revocation with efficient update of anonymous credentials. In Juan A. Garay and Roberto De Prisco, editors, *SCN 10*, volume 6280 of *LNCS*, pages 454–471. Springer, Heidelberg, September 2010.

[CL01] Jan Camenisch and Anna Lysyanskaya. An efficient system for non-transferable anonymous credentials with optional anonymity revocation. In Birgit Pfitzmann, editor, *EUROCRYPT 2001*, volume 2045 of *LNCS*, pages 93–118. Springer, Heidelberg, May 2001.

[CL04] Jan Camenisch and Anna Lysyanskaya. Signature schemes and anonymous credentials from bilinear maps. In Matthew Franklin, editor, *CRYPTO 2004*, volume 3152 of *LNCS*, pages 56–72. Springer, Heidelberg, August 2004.

[Dam00] Ivan Damgård. Efficient concurrent zero-knowledge in the auxiliary string model. In Bart Preneel, editor, *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 418–430. Springer, Heidelberg, May 2000.

[DDD05] Liesje Demuynck and Bart De Decker. Anonymous updating of credentials. Technical report, December 2005.

[DMM$^+$18] Dominic Deuber, Matteo Maffei, Giulio Malavolta, Max Rabkin, Dominique Schröder, and Mark Simkin. Functional credentials. *PoPETs*, 2018(2):64–84, 2018.

[GGM14] Christina Garman, Matthew Green, and Ian Miers. Decentralized anonymous credentials. In *NDSS 2014*. The Internet Society, February 2014.

[GS08] Jens Groth and Amit Sahai. Efficient non-interactive proof systems for bilinear groups. In Nigel P. Smart, editor, *EUROCRYPT 2008*, volume 4965 of *LNCS*, pages 415–432. Springer, Heidelberg, April 2008.

[HHNR17] Gunnar Hartung, Max Hoffmann, Matthias Nagel, and Andy Rupp. BBA+: improving the security and applicability of privacy-preserving point collection. In *ACM Conference on Computer and Communications Security*, pages 1925–1942. ACM, 2017.

[JR16] Tibor Jager and Andy Rupp. Black-box accumulation: Collecting incentives in a privacy-preserving way. *PoPETs*, 2016(3):62–82, 2016.

[MDPD15] Milica Milutinovic, Italo Dacosta, Andreas Put, and Bart De Decker. ucentive: An efficient, anonymous and unlinkable incentives scheme. In *TrustCom/BigDataSE/ISPA (1)*, pages 588–595. IEEE, 2015.

[NDD06] Vincent Naessens, Liesje Demuynck, and Bart De Decker. A fair anonymous submission and review system. In Herbert Leitold and Evangelos P. Markatos, editors, *Communications and Multimedia Security, 10th IFIP TC-6 TC-11 International Conference, CMS 2006, Heraklion, Crete, Greece, October 19-21, 2006, Proceedings*, volume 4237 of *Lecture Notes in Computer Science*, pages 43–53. Springer, 2006.

[PAY19] PAYBACK GmbH. Payback. `https://www.payback.net/`, January 2019.

[PS16] David Pointcheval and Olivier Sanders. Short randomizable signatures. In Kazue Sako, editor, *CT-RSA 2016*, volume 9610 of *LNCS*, pages 111–126. Springer, Heidelberg, February / March 2016.

A Security Definitions for Building Blocks

Definition 17 (Unforgeability)**.** Consider the following unforgeability game $\mathrm{Exp}^{\text{blind-uf}}(\Pi, \mathcal{A}, \lambda)$ for a blind signature scheme Π:

- The experiment runs $pp \leftarrow \mathcal{G}(1^\lambda)$ and hands pp to $\mathcal{A}$. $\mathcal{A}$ responds with 1^n for some $n \in \mathbb{N}$. The experiment generates $(pk, sk) \leftarrow \mathsf{KeyGen}(pp, 1^n)$ and hands pk to $\mathcal{A}$.
- $\mathcal{A}$ can query signatures by announcing c, $\vec{m} \in \mathcal{M}^n$ and r such that $c = \mathsf{Commit}(pp, pk, \vec{m}, r)$. The experiment then runs $\mathsf{BlindSign}(pp, pk, sk, c)$ interacting with $\mathcal{A}$ and records $\vec{m}$.
- Eventually, $\mathcal{A}$ outputs $\vec{m}^*$ and σ^*. The experiment outputs 1 iff $\mathsf{Vrfy}(pp, pk, \vec{m}^*, \sigma^*) = 1$ and $\vec{m}^*$ was not recorded in any query.

Π has *blind unforgeability* if for all ppt $\mathcal{A}$ there exists *negl* such that $\Pr[\mathrm{Exp}^{\text{blind-uf}}(\Pi, \mathcal{A}, \lambda) = 1] \leq \mathit{negl}(\lambda)$ for all λ. $\diamond$

Definition 18 (Perfect msg privacy)**.** We say that a blind signature scheme has perfect message privacy if

- *"the commitment scheme is perfectly hiding"*: For all $\vec{m}_0, \vec{m}_1 \in M^n$, $\mathsf{Commit}(pp, pk, \vec{m}_0, r_0)$ is distributed exactly the same as $\mathsf{Commit}(pp, pk, \vec{m}_1, r_1)$ over the random choice of r_0, r_1.

- *"$\mathsf{BlindRcv}$ does not reveal the message"*: for any two messages $\vec{m}_0, \vec{m}_1 \in M^n$ and all (unrestricted) $\mathcal{A}$:

$$\begin{aligned}&(\mathrm{output}_{\mathcal{A}}[\mathcal{A}(C_0) \leftrightarrow \mathsf{BlindRcv}(pp, pk, \vec{m}_0, r_0)], \chi_0)\\ &\text{is distributed exactly like}\\ &(\mathrm{output}_{\mathcal{A}}[\mathcal{A}(C_1) \leftrightarrow \mathsf{BlindRcv}(pp, pk, \vec{m}_1, r_1)], \chi_1)\end{aligned}$$

  where r_0, r_1 is chosen uniformly at random, $C_j = \mathsf{Commit}(pp, pk, \vec{m}_j, r_j)$ and χ_j is an indicator variable with $\chi_j = 1$ if and only if $\mathsf{Vrfy}(pp, pk, \vec{m}_j, \sigma_j) = 1$ for the local output σ_j of $\mathsf{BlindRcv}$ in either case. $\diamond$

While this definition may seem strong, it is satisfied, for example, by the Pointcheval Sanders blind signature scheme [PS16], where Commit is a effectively a (perfectly hiding) Pedersen commitment, Their $\mathsf{BlindRcv}$ (in our formulation without zero-knowledge proof) does not send any messages (meaning the output of $\mathcal{A}$ is clearly independent of $\vec{m}$), and the χ_j bit (validity of the resulting signature) is also independent of the committed message.

Definition 19 (Key-indistinguishable CPA)**.** Let Π_{enc} be a public-key encryption scheme. Consider the following experiments $\mathrm{Exp}_b^{\text{key-ind}}(\Pi_{\mathrm{enc}}, \mathcal{A}, \lambda)$ for $b \in \{0,1\}$.

- The experiment generates public parameters $pp \leftarrow \mathcal{G}(1^\lambda)$ and two keys $\mathsf{KeyGen}_{\mathrm{enc}}(pp) \to sk_0, sk_1$, hands $\mathcal{A}$ the pp and the two public keys $(pk_0, pk_1) = (\mathsf{ComputePK}_{\mathrm{enc}}(pp, sk_0), \mathsf{ComputePK}_{\mathrm{enc}}(pp, sk_1))$.
- $\mathcal{A}$ outputs two messages $m_0, m_1 \in M_{pp}$.
- $\mathcal{A}$ gets $\mathsf{Encrypt}_{\mathrm{enc}}(pp, \underline{pk_b}, m_b)$ from the experiment and outputs a bit $\hat{b}$.

We say that Π_{enc} is key-ind. CPA secure if for all ppt $\mathcal{A}$, there exists a negligible function *negl* s.t. $|\Pr[\mathrm{Exp}_0^{\text{key-ind}}(\Pi_{\mathrm{enc}}, \mathcal{A}, \lambda) = 1] - \Pr[\mathrm{Exp}_1^{\text{key-ind}}(\Pi_{\mathrm{enc}}, \mathcal{A}, \lambda) = 1]| \leq negl(\lambda)$

Definition 20 (Perfectly binding commitment)**.** A (malleable) commitment scheme is *perfectly binding* if for all $pp \in [\mathcal{G}(1^\lambda)]$, $pk \in [\mathsf{KeyGen}(pp)]$ and all $(c, o) \in [\mathsf{Commit}(pp, pk, m)]$, there exists no m', o' such that $m' \neq m$ and $\mathsf{Vrfy}(pp, pk, c, o', m') = 1$.

Definition 21 (Comp. hiding commitment)**.** Let Π_{cmt} be a malleable commitment scheme. Consider the following experiment $\mathrm{Exp}_b^{\mathrm{hid}}(\Pi_{\mathrm{cmt}}, \mathcal{A}, \lambda)$.

- $pp \leftarrow \mathcal{G}(1^\lambda)$, $pk \leftarrow \mathsf{KeyGen}(pp)$, $(m_0, m_1, st) \leftarrow \mathcal{A}(pp, pk)$, $m_0, m_1 \in M_{pp}$
- $\hat{b} \leftarrow \mathcal{A}(c, st)$ where $c = \mathsf{Commit}(pp, pk, m_b)$

We say that Π_{cmt} is computational hiding if for all ppt $\mathcal{A}$, there exists a negligible function *negl* s.t. $|\Pr[\mathrm{Exp}_0^{\mathrm{hid}}(\Pi_{\mathrm{cmt}}, \mathcal{A}, \lambda) = 1] - \Pr[\mathrm{Exp}_1^{\mathrm{hid}}(\Pi_{\mathrm{cmt}}, \mathcal{A}, \lambda) = 1]| \leq negl(\lambda)$

B Formal Description of the Generic Construction of UACS

Construction 22. Let Π_{sig} be a blind signature (Definition 1). We construct UACS:

$\mathsf{Setup}(pp) \to cpp$ generates public parameters cpp consisting of pp and a zero-knowledge argument common reference string. The attribute space $\mathbb{A}$ is the signature scheme's message space $\mathcal{M}_{\mathrm{sig}}$.

$\mathsf{IssuerKeyGen}(cpp, 1^n) \to (pk, sk)$ generates keys by running algorithm $\mathsf{KeyGen}_{\mathrm{sig}}(pp, 1^n) \to (pk, sk)$. The update function universe Ψ consists of all $\psi : (\mathcal{M}_{\mathrm{sig}}^n \cup \{\perp\}) \times \{0,1\}^* \to \mathcal{M}_{\mathrm{sig}}^n \cup \{\perp\}$ that are supported by the zero-knowledge arguments below.

$\mathsf{Issue}(cpp, pk, \psi, sk) \leftrightarrow \mathsf{Receive}(cpp, pk, \psi, \alpha) \rightarrow cred$ for $\psi \in \Psi$ works as follows:

- The receiver computes $\vec{A} = \psi(\perp, \alpha)$ and commits to $\vec{A}$ by computing $c = \mathsf{Commit}_{\mathrm{sig}}(pp, pk, \vec{A}, r)$ for random r and sends c to the issuer.
- Receiver proves $\mathsf{ZKAK}[(\alpha, r); c = \mathsf{Commit}_{\mathrm{sig}}(pp, pk, \psi(\perp, \alpha), r)]$
- If the proof accepts, issuer runs $\mathsf{BlindSign}_{\mathrm{sig}}(pp, pk, sk, c)$ and receiver runs $\mathsf{BlindRcv}_{\mathrm{sig}}(pp, pk, \vec{A}, r) \rightarrow \sigma$.
- The receiver checks if $\mathsf{Vrfy}_{\mathrm{sig}}(pp, pk, \vec{A}, \sigma) = 1$. If so, it outputs $cred = (\vec{A}, \sigma)$, otherwise it outputs $\perp$.

$b \leftarrow \mathsf{Update}(cpp, pk, \psi, sk) \leftrightarrow \mathsf{UpdRcv}(cpp, pk, \psi, \alpha, cred) \rightarrow cred^*$ works as follows:

- The receiver parses $cred = (\vec{A}, \sigma)$ and computes $\vec{A}^* = \psi(\vec{A}, \alpha)$.
- The receiver commits to $\vec{A}^*$ by computing the commitment $c = \mathsf{Commit}_{\mathrm{sig}}(pp, pk, \vec{A}^*, r)$ for random r and sends c to the issuer.
- Then, the receiver proves $\mathsf{ZKAK}[(\vec{A}, \sigma, \alpha, r); \mathsf{Vrfy}_{\mathrm{sig}}(pp, pk, \vec{A}, \sigma) = 1 \wedge c = \mathsf{Commit}_{\mathrm{sig}}(pp, pk, \psi(\vec{A}, \alpha), r)]$.
- If the proof rejects, the issuer outputs 0 and aborts.
- Otherwise, issuer runs $\mathsf{BlindSign}_{\mathrm{sig}}(pp, pk, sk, c)$ while receiver runs $\mathsf{BlindRcv}_{\mathrm{sig}}(pp, pk, \vec{A}^*, r) \rightarrow \sigma^*$.
- The receiver checks if $\mathsf{Vrfy}_{\mathrm{sig}}(pp, pk, \vec{A}^*, \sigma^*) = 1$. If so, it outputs $cred^* = (\vec{A}^*, \sigma^*)$, otherwise it outputs $\perp$. The issuer outputs 1.

$\mathsf{ShowPrv}(cpp, pk, \phi, \alpha, cred) \leftrightarrow \mathsf{ShowVrfy}(cpp, pk, \phi) \rightarrow b$ works as follows: the prover parses $cred = (\vec{A}, \sigma)$. If $\phi(\vec{A}, \alpha) = 0$, the prover aborts and the verifier outputs 0. Otherwise, the prover runs the proof $\mathsf{ZKAK}[(\vec{A}, \alpha, \sigma); \mathsf{Vrfy}_{\mathrm{sig}}(pp, pk, \vec{A}, \sigma) = 1 \wedge \phi(\vec{A}, \alpha) = 1]$. If the proof succeeds, the verifier outputs 1, otherwise 0.

C Security Proof for Updatable Credentials

In this section, we sketch the security proofs for Construction 7.

Theorem 8: Anonymity. We define the simulators as follows:

- $\mathfrak{S}_{\mathsf{Setup}}(pp)$ runs the trapdoor generator of the ZKAK and outputs cpp (which contains pp and the ZKAK common reference string from the trapdoor generator), and the simulation trapdoor td.
- $\mathfrak{S}_{\mathsf{Receive}}(td, pk, \psi)$ and $\mathfrak{S}_{\mathsf{UpdRcv}}(td, pk, \psi)$ work very similarly to one another: they both commit to $\vec{0}$ as $c = \mathsf{Commit}_{\mathrm{sig}}(pp, pk, \vec{0}, r)$ with random r and send c to $\mathcal{A}$. They then simulate the ZKAK proof (in Receive or UpdRcv) using td. Finally, they both run $\mathsf{BlindRcv}_{\mathrm{sig}}(pp, pk, \vec{0}, r) \rightarrow \sigma$ and compute the bit $b = \mathsf{Vrfy}_{\mathrm{sig}}(pp, pk, \vec{0}, \sigma)$. They send b to $\mathcal{A}$.
- $\mathfrak{S}_{\mathsf{ShowPrv}}(td, pk, \phi)$ simulates the ZKAK.

Given that Π_{sig} has perfect message privacy by assumption, the commitment c and the bit b computed by the simulator have the same distribution as in Receive or UpdRcv. Simulation of the zero-knowledge arguments produces the correct view for $\mathcal{A}$ by assumption. □

Theorem 9: Soundness. We define the algorithm $\mathcal{E}$ that is supposed to extract an explanation list $\mathcal{L}$ as follows:

- On input $(cpp, r_{\mathcal{A}}, r_{\mathsf{Issue}}, r_{\mathsf{Update}})$, the extractor $\mathcal{E}^{\mathcal{A}}$ first runs $\mathcal{A}$ with randomness $r_{\mathcal{A}}$ and *cpp* until $\mathcal{A}$ halts.
- For the ith query to Issue, Update, or ShowVrfy in this run, $\mathcal{E}$ does the following:
  - if it is a query to Issue and the proof of knowledge within Issue is accepting, then $\mathcal{E}$ uses the proof of knowledge extractor to obtain a witness (α, r). It stores $\alpha_i := \alpha$ on $\mathcal{L}$. If the proof of knowledge is not accepting, it stores some arbitrary $\alpha_i \in \{0,1\}^*$ on $\mathcal{L}$.
  - if it is a query to Update and the proof of knowledge within Update is accepting, then $\mathcal{E}$ uses the proof of knowledge extractor to obtain a witness $(\vec{A}, \sigma, \alpha, r)$. It stores $(\vec{A}_i, \alpha_i) := (\vec{A}, \alpha)$ on $\mathcal{L}$.
  - if it is a query to ShowVrfy and the proof of knowledge within ShowVrfy is accepting, then $\mathcal{E}$ uses the proof of knowledge extractor to obtain a witness $(\vec{A}, \sigma)$. It stores $\vec{A}_i := \vec{A}$ on $\mathcal{L}$.
- $\mathcal{E}$ outputs $\mathcal{L}$.

Since the argument of knowledge extractor runs in expected polynomial time, $\mathcal{E}$ runs in expected polynomial time, too (probability over $r_{\mathcal{A}}$ and $\mathcal{E}$'s random coins).

With this $\mathcal{E}$, the soundness of our updatable credential construction can be reduced to unforgeability of the underlying blind signature scheme sig (Definition 17). Let $\mathcal{E}$ be as above. Let $\mathcal{A}$ be an attacker against $\mathrm{Exp}^{\mathrm{sound}}$. We construct $\mathcal{B}$ against $\mathrm{Exp}^{\text{blind-uf}}$:

- $\mathcal{B}$ runs $\mathcal{A}$ with randomness $r_{\mathcal{A}}$
- $\mathcal{B}$ receives *pp* from the unforgeability experiment. $\mathcal{B}$ generates *cpp* from *pp* and hands *cpp* to $\mathcal{A}$. $\mathcal{A}$ responds with 1^n for some $n \in \mathbb{N}$. $\mathcal{B}$ hands 1^n to its challenger, receiving *pk*. $\mathcal{B}$ hands *pk* to $\mathcal{A}$.
- Whenever $\mathcal{A}$ queries Issue with update function ψ, $\mathcal{B}$ checks the proof of knowledge. If it accepts, $\mathcal{B}$ uses the proof of knowledge extractor to obtain a witness (α, r). $\mathcal{B}$ submits $\vec{m} := \psi(\perp, \alpha)$, r, and $c := \mathsf{Commit}_{\mathrm{sig}}(pp, pk, \vec{m}, r)$ to its challenger, who starts running $\mathsf{BlindSign}_{\mathrm{sig}}(pp, pksk, c)$. $\mathcal{B}$ relays the messages for $\mathsf{BlindSign}_{\mathrm{sig}}$ between its challenger and $\mathcal{A}$.
- Whenever $\mathcal{A}$ queries Update with update function ψ, $\mathcal{B}$ checks the proof of knowledge. If it accepts, $\mathcal{B}$ uses the proof of knowledge extractor to obtain a witness $(\vec{A}, \sigma, \alpha, r)$. If $\mathcal{B}$ has not queried its challenger for $\vec{A}$ before, it outputs $\vec{m} := \vec{A}$ and σ as a forgery. Otherwise, $\mathcal{B}$ submits $\vec{m} := \psi(\vec{A}, \alpha)$, r, and $c := \mathsf{Commit}_{\mathrm{sig}}(pp, pk, \vec{m}, r)$ to its challenger, who starts running $\mathsf{BlindSign}_{\mathrm{sig}}(pp, pk, sk, c)$. $\mathcal{B}$ relays the messages for BlindSignsig between its challenger and $\mathcal{A}$.
- Whenever $\mathcal{A}$ queries ShowVrfy with predicate ϕ, $\mathcal{B}$ checks the proof of knowledge. If it accepts, $\mathcal{B}$ uses the proof of knowledge extractor to obtain a witness $(\vec{A}, \alpha, \sigma)$. If $\mathcal{B}$ has not queried its challenger for $\vec{A}$ before, it outputs $\vec{m} := \vec{A}$ and σ as a forgery.
- Eventually, $\mathcal{A}$ and halts. $\mathcal{B}$ runs $\mathcal{E}^{\mathcal{A}}(cpp, r_{\mathcal{A}}, r_{\mathsf{Issue}}, r_{\mathsf{Update}})$ (using the same random coins for $\mathcal{E}$ that $\mathcal{B}$ used for its extraction of proofs of knowledge, ensuring that the output of $\mathcal{E}$ will be consistent with the values extracted by $\mathcal{B}$ before) to obtain $\mathcal{L}$.
- Then $\mathcal{B}$ halts.

Analysis: Whenever $\mathcal{B}$ outputs a signature forgery, it is guaranteed that the signature is valid (since they are valid witnesses in a proof of knowledge for a relation that requires signature validity). If $\mathcal{B}$ outputs a forgery during an Update or ShowVrfy query, by construction it has never asked for the message to be signed before.

It is easy to see that the simulation is perfect. If $\mathcal{B}$ does not halt before $\mathcal{A}$ halts, the output $\mathcal{L}$ of $\mathcal{E}$ necessarily fulfills argument consistency: Suppose for contradiction that $\mathcal{L}$ is not consistent, i.e. there is some index i such that $\mathcal{L}$ is inconsistent for that index. Let E_i be as prescribed in the soundness experiment given $\mathcal{L}$. Note that before the ith query, $\mathcal{B}$ has only queried its oracle for signatures on messages $\vec{A} \in E_{i-1}$.

- Assume i belongs to an Issue query. By definition i cannot have caused the inconsistency.
- Assume i belongs to an Update query with update function ψ_i. Then the entry on $\mathcal{L}$ is some $(\vec{A}_i, \alpha_i)$. Because i caused the inconsistency, Update has output 1 (implying that $\mathcal{B}$ runs the proof of knowledge extractor and obtained the witness $(\vec{A}, \sigma, \alpha, r)$) and (1) $\psi_i(\vec{A}_i, \alpha_i) = \perp$ or (2) $\vec{A}_i \notin E_{i-1}$. (1) can be ruled out since $\psi_i(\vec{A}_i, \alpha_i) \neq \perp$ is guaranteed by the proof of knowledge statement and hence by its extractor. If (2) happens, then $\mathcal{B}$ halts and claims a forgery (as it has not queried $\vec{A}_i$ to its oracle before), contradicting that $\mathcal{B}$ does not halt before $\mathcal{A}$ halts.
- Assume i belongs to a $\mathsf{ShowVrfy}$ query with predicate ϕ_i. This case is handled analogously to Update.

So we know that if $\mathcal{B}$ does not halt before $\mathcal{A}$ halts, then $\mathcal{E}$ outputs a consistent $\mathcal{L}$, implying that $\Pr[\mathrm{Exp}^{\text{blind-uf}}(\Pi_{\text{sig}}, \mathcal{B}, \lambda)] \geq \Pr[\mathrm{Exp}^{\text{sound}}(\Pi, \mathcal{A}, \mathcal{E}, \lambda) = 1]$. So if for $\mathcal{E}$ as defined above, there exists an adversary $\mathcal{A}$ with non-negligible success probability, then there exists $\mathcal{B}$ (as defined above) with non-negligible success probability against the blind signature scheme. By assumption, such a $\mathcal{B}$ does not exist, hence the updatable credential system is sound. (Note that $\mathcal{B}$ runs in expected polynomial time. This can be converted to polynomial time by trading off success probability using Markov's inequality.) □

D Compact Description of the Incentive System Construction

We list the generic incentive system construction, as explained in Section 7, in a more compact and formal manner.

Construction 23. Let Π_{uacs} be an UACS, Π_{enc} be a public-key encryption scheme, and let Π_{cmt} be an additively malleable commitment scheme. We define the incentive system Π_{insy} as follows:

$\mathsf{Setup}(pp) \to ispp$ runs $cpp \leftarrow \mathsf{Setup}_{\text{uacs}}(pp)$. pp fixes an attribute space $\mathbb{A}$ and message space $\mathcal{M}_{\text{enc}}$ for the encryption scheme. We assume $\mathbb{A} = \mathbb{Z}_p$ for some super-poly p and set $v_{max} = p - 1$. Setup chooses a commitment key $pk_{\text{cmt}} \leftarrow \mathsf{KeyGen}_{\text{cmt}}(pp)$. It outputs $ispp = (pp, cpp, pk_{\text{cmt}})$.

$\mathsf{KeyGen}(ispp) \to (upk, usk)$ generates an encryption key $usk_{\text{enc}} \leftarrow \mathsf{KeyGen}_{\text{enc}}(pp)$ and $upk_{\text{enc}} = \mathsf{ComputePK}_{\text{enc}}(pp, usk_{\text{enc}})$. It outputs $upk = upk_{\text{enc}}$ and $usk = usk_{\text{enc}}$.

$\mathsf{IssuerKeyGen}(ispp) \to (pk, sk)$ outputs a credential issuer key pair $(pk, sk) \leftarrow \mathsf{IssuerKeyGen}_{\text{uacs}}(cpp, 1^n)$ for $n = 4$.

$\mathsf{Issue}(ispp, pk, upk, sk) \leftrightarrow \mathsf{Join}(ispp, pk, upk, usk) \to (token, dsid)$
the user picks $dsid_{\text{usr}} \leftarrow \mathbb{Z}_p$ and computes commitment and open value $(C_{\text{usr}}, open) \leftarrow \mathsf{Commit}_{\text{cmt}}(pp, pk_{\text{cmt}}, dsid_{\text{usr}})$. The user sends C_{usr} to the provider. The provider replies with $dsid_{\text{prvdr}} \leftarrow \mathbb{Z}_p$. Both parties compute $C_{\text{dsid}} = \mathsf{Add}_{\text{cmt}}(pp, pk_{\text{cmt}}, C_{\text{usr}}, dsid_{\text{prvdr}})$. Then the user sets $dsid = dsid_{\text{usr}} + dsid_{\text{prvdr}}$, chooses $dsrnd \leftarrow \mathbb{Z}_p$, and sets $\alpha = (usk, dsid, dsrnd, open)$. Then the provider runs $\mathsf{Issue}_{\text{uacs}}(cpp, pk, \psi, sk)$ and the user runs $\mathsf{Receive}_{\text{uacs}}(cpp, pk, \psi, \alpha) \to cred$. Here, the update function is set to $\psi(\perp, (usk, dsid, dsrnd, open)) = (usk, dsid, dsrnd, 0)$, if user public key $upk = \mathsf{ComputePK}_{\text{enc}}(pp, usk)$ and it holds that $\mathsf{Vrfy}_{\text{cmt}}(pp, pk_{\text{cmt}}, C_{\text{dsid}}, dsid, open) = 1$. Otherwise, set $\psi(\perp, \alpha) = \perp$. If $cred \neq \perp$, the user outputs $token = (dsid, dsrnd, v = 0, cred)$ and $dsid$. Otherwise, the user outputs $\perp$.

$\mathsf{Credit}(ispp, pk, k, sk) \leftrightarrow \mathsf{Earn}(ispp, pk, k, usk, token) \to token^*$ the user parses $token = (dsid, dsrnd, v, cred)$ and checks that $v + k \leq v_{max}$. The protocol then works as follows: The provider runs $\mathsf{Update}_{\mathrm{uacs}}(cpp, pk, \psi, sk)$, interacting with the user running $\mathsf{UpdRcv}_{\mathrm{uacs}}(cpp, pk, \psi, \alpha, cred) \to cred^*$ with $\alpha = \perp$. Here, the update function is set to $\psi((usk, dsid, dsrnd, v), \cdot) = (usk, dsid, dsrnd, v + k)$. If $cred^* \neq \perp$, the user outputs $token^* = (dsid, dsrnd, v + k, cred^*)$.

$(token^*, dsid^*) \leftarrow \mathsf{Spend}(ispp, pk, k, dsid, usk, token) \leftrightarrow \mathsf{Deduct}($
$ispp, pk, k, dsid, sk) \to (b, dstag)$ first has the user parse $token$ as $token = (dsid, dsrnd, v, cred)$ and check that $v \geq k$. Then:

- The user chooses $dsid^*_{\mathrm{usr}} \leftarrow \mathbb{Z}_p$ and generates $(C_{\mathrm{usr}}{}^*, open^*) \leftarrow \mathsf{Commit}_{\mathrm{cmt}}(pp, pk_{\mathrm{cmt}}, dsid^*_{\mathrm{usr}})$. He sends $C_{\mathrm{usr}}{}^*$ to the provider.
- The provider chooses a random challenge $\gamma \leftarrow \mathbb{Z}_p$ and a random $dsid^*_{\mathrm{prvdr}} \leftarrow \mathbb{Z}_p$, and sends both to the user.
- Issuer and user each compute $C_{\mathrm{dsid}}{}^* = \mathsf{Add}_{\mathrm{cmt}}(pp, pk_{\mathrm{cmt}}, C_{\mathrm{usr}}{}^*, dsid^*_{\mathrm{prvdr}})$.
- The user prepares values $dsid^* = dsid^*_{\mathrm{usr}} + dsid^*_{\mathrm{prvdr}}$ and $dsrnd^* \leftarrow \mathbb{Z}_p$ for his next token and sets $\alpha = (dsid^*, dsrnd^*, open^*)$.
- The user computes $c = usk \cdot \gamma + dsrnd$.
- The user encrypts $dsid^*$ as $ctrace \leftarrow \mathsf{Encrypt}_{\mathrm{enc}}(pp, upk, dsid^*)$.
- The user sends c, $ctrace$ to the provider.
- The provider runs $b \leftarrow \mathsf{Update}_{\mathrm{uacs}}(cpp, pk, \psi, sk)$ and the user runs algorithm $cred^* \leftarrow \mathsf{UpdRcv}_{\mathrm{uacs}}(cpp, pk, \psi, \alpha, cred)$. Here, the update function is $\psi((usk, dsid, dsrnd, v), (dsid^*, dsrnd^*, open^*)) = (usk, dsid^*, dsrnd^*, v - k)$ if
  - $dsid$ is the same as in the Deduct input,
  - $v \geq k$,
  - $\mathsf{Vrfy}_{\mathrm{cmt}}(pp, pk_{\mathrm{cmt}}, C_{\mathrm{dsid}}{}^*, dsid^*, open^*)$,
  - $c = usk \cdot \gamma + dsrnd$, and
  - $\mathsf{Decrypt}_{\mathrm{enc}}(pp, usk, ctrace) = dsid^*$.

  Otherwise, $\psi(\dots) = \perp$.
- If $cred^* \neq \perp$, the user outputs $(token^* = (dsid^*, dsrnd^*, v - k, cred^*), dsid^*)$.
- The provider outputs b and, if $b = 1$, $dstag = (c, \gamma, ctrace)$.

$\mathsf{Link}(ispp, dstag, dstag') \to (upk, dslink)$ with $dstag = (c, \gamma, ctrace)$, $dstag' = (c', \gamma', ctrace')$, outputs $dslink = (c - c')/(\gamma - \gamma')$ (the intent is that $dslink = usk$) and $upk = \mathsf{ComputePK}(pp, dslink)$.

$\mathsf{Trace}(ispp, dslink, dstag) \to dsid^*$ for $dstag = (c, \gamma, ctrace)$ retrieves $dsid^*$ by decrypting $ctrace$, i.e. $dsid^* = \mathsf{Decrypt}_{\mathrm{enc}}(pp, dslink, ctrace)$.

$\mathsf{VrfyDs}(ispp, dslink, upk) \to b$ outputs 1 iff $\mathsf{ComputePK}(pp, dslink) = upk$.

E Concrete Construction from Pointcheval Sanders Blind Signatures

We present a concrete construction based on Pointcheval Sanders blind signatures [PS16] for the UACS and ElGamal encryption for the public-key encryption scheme and the additively malleable commitment. For this, we follow the generic construction of UACS (Construction 7) and the incentive system (Construction 23) closely with one change: We sign $dsid \in \mathbb{Z}_p$, but we *encrypt* $Dsid = w^{dsid} \in \mathbb{G}_1$. Hence, when tracing double-spent transactions, one only learns $Dsid^* = w^{dsid^*}$ instead of $dsid^*$. This is not a restriction since the output of Trace is only needed to quickly find the corresponding transaction to $dsid^*$. So in practice, the issuer would store $Dsid$ instead of $dsid$ for every transaction that he observes, and then use $Dsid$ to quickly find the transaction pointed at by Trace. Note that the security of the construction is not impacted (the same security proofs still apply almost verbatim).

Construction 24 (Inc. system from Pointcheval Sanders signatures)**.** Let ($\mathsf{KeyGen}_{\mathrm{PS}}, \mathsf{Commit}_{\mathrm{PS}}, \mathsf{BlindSign}_{\mathrm{PS}}, \mathsf{BlindRcv}_{\mathrm{PS}}, \mathsf{Vrfy}_{\mathrm{PS}}$) denote the Pointcheval Sanders signature scheme [PS16].

$\mathcal{G}(1^\lambda) \to pp$ generates and outputs a type 3 bilinear group $pp = (\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_\mathrm{T}, p, e)$ of prime order p.

$\mathsf{Setup}(pp) \to ispp$ and chooses a random $w \leftarrow \mathbb{G}_1$ as a shared base for ElGamal encryption Π_{enc} and $g, h \leftarrow \mathbb{G}_1$ for the malleable commitment Π_{cmt} (also ElGamal). Setup also generates a Pedersen commitment key $g_{\mathrm{Damgård}}, h_{\mathrm{Damgård}} \leftarrow \mathbb{G}_1$ and a collision-resistant hash function H, both for Damgård's technique [Dam00], enabling efficient simulation of ZKAK protocols. We omit these values in the following. It outputs $ispp = (pp, w, g, h)$. The maximum point score is $v_{max} = p - 1$.

$\mathsf{KeyGen}(ispp) \to (upk, usk)$ generates an encryption key pair by choosing a random $usk \leftarrow \mathbb{Z}_p$ and computing $upk = w^{usk}$. It outputs (upk, usk).

$\mathsf{IssuerKeyGen}(ispp) \to (pk, sk)$ generates $(pk_{\mathrm{PS}}, sk_{\mathrm{PS}}) \leftarrow \mathsf{KeyGen}_{\mathrm{PS}}(pp, 1^{n=4})$. We write the keys as $sk_{\mathrm{PS}} = (x, y_1, \dots, y_4)$ and $pk_{\mathrm{PS}} = (g, g^{y_1}, \dots, g^{y_4}, \tilde{g}, \tilde{g}^x, \tilde{g}^{y_1}, \dots, \tilde{g}^{y_4})$. $\mathsf{IssuerKeyGen}$ outputs $pk = pk_{\mathrm{PS}}$ and $sk = sk_{\mathrm{PS}}$.

$\mathsf{Issue}(ispp, pk, upk, sk) \leftrightarrow \mathsf{Join}(ispp, pk, upk, usk) \to (token, dsid)$ works as follows:

- The user chooses random $dsid_{\mathrm{usr}} \leftarrow \mathbb{Z}_p$ and computes the commitment $C_{\mathrm{usr}} = (g_{\mathrm{usr}}^{dsid} \cdot h^{open}, g^{open})$ for a random $open \leftarrow \mathbb{Z}_p$. It sends C_{usr} to the issuer.
- The issuer replies with a random $dsid_{\mathrm{prvdr}} \leftarrow \mathbb{Z}_p$. Both issuer and user compute $C_{\mathrm{dsid}} = (g_{\mathrm{usr}}^{dsid} \cdot h^{open} \cdot g_{\mathrm{prvdr}}^{dsid}, g^{open})$.
- The user sets $dsid = dsid_{\mathrm{usr}} + dsid_{\mathrm{prvdr}}$ and chooses random $dsrnd, r \leftarrow \mathbb{Z}_p$, computes $Dsid = w^{dsid}$ and sends $c = (g^{y_1})^{usk} \cdot (g^{y_2})^{dsid} \cdot (g^{y_3})^{dsrnd} \cdot g^r$ to the issuer.
- The user proves $\mathsf{ZKAK}[(usk, dsid, dsrnd, r, open); c = (g^{y_1})^{usk} \cdot (g^{y_2})^{dsid} \cdot (g^{y_3})^{dsrnd} \cdot g^r \wedge upk = w^{usk} \wedge C_{\mathrm{dsid}} = (g^{dsid} \cdot h^{open}, g^{open})]$.
- If the proof is accepted, the issuer sends $\sigma'_{\mathrm{PS}} = (\sigma'_0, \sigma'_1) = (g^{r'}, (c \cdot g^x)^{r'})$ for a random $r' \leftarrow \mathbb{Z}_p^*$ to the user.
- The user unblinds the signature as $\sigma_{\mathrm{PS}} = (\sigma'_0, \sigma'_1 \cdot (\sigma'_0)^{-r})$.
- The user checks $\mathsf{Vrfy}_{\mathrm{PS}}(pp, pk_{\mathrm{PS}}, (usk, dsid, dsrnd, 0), \sigma_{\mathrm{PS}}) \overset{!}{=} 1$. If the checks succeed, it outputs $token = (dsid, dsrnd, v = 0, \sigma_{\mathrm{PS}})$ and $Dsid$, otherwise it outputs $\perp$.

$\mathsf{Credit}(ispp, pk, k, sk) \leftrightarrow \mathsf{Earn}(ispp, pk, k, usk, token) \to token^*$ where $token = (dsid, dsrnd, v, \sigma_{\mathrm{PS}} = (\sigma_0, \sigma_1))$ works as follows:

- The user computes randomized signatures $(\sigma'_0, \sigma'_1) = (\sigma_0^r, (\sigma_1 \cdot \sigma_0^{r'})^r)$ for $r \leftarrow \mathbb{Z}_p^*$, $r' \leftarrow \mathbb{Z}_p$. He sends σ'_0, σ'_1 to the issuer.
- The user proves $\mathsf{ZKAK}[(usk, dsid, dsrnd, v, r'); \mathsf{Vrfy}_{\mathrm{PS}}(pp, pk_{\mathrm{PS}}, (usk, dsid, dsrnd, v), (\sigma'_0, \sigma'_1 \cdot (\sigma'_0)^{-r'})) = 1]$

- If the proof accepts, the issuer sends $(\sigma_0'', \sigma_1'') = ((\sigma_0')^{r''}, ((\sigma_1') \cdot (\sigma_0')^{y_4 \cdot k})^{r''})$ for a random $r'' \leftarrow \mathbb{Z}_p^*$ to the user.
- The user unblinds the signature as $\sigma = (\sigma_0^*, \sigma_1^*) = (\sigma_0'', \sigma_1'' \cdot (\sigma_0'')^{-r'})$ and checks $\mathsf{Vrfy}_{\mathrm{PS}}(pp, pk_{\mathrm{PS}}, (usk, dsid, dsrnd, v + k), \sigma_{\mathrm{PS}}^*) \overset{!}{=} 1$. If the check succeeds, it outputs $token^* = (dsid, dsrnd, v + k, \sigma_{\mathrm{PS}}^*)$, otherwise it outputs $\perp$.

$(token^*, Dsid^*) \leftarrow \mathsf{Spend}(ispp, pk, k, dsid, usk, token) \leftrightarrow \mathsf{Deduct}(ispp, pk, k, dsid, sk) \rightarrow (b, dstag)$ where $token = (dsid, dsrnd, v, cred)$ works as follows:

- The user chooses random $dsid_{\mathrm{usr}}^* \leftarrow \mathbb{Z}_p$ and computes the commitment $C_{\mathrm{usr}}{}^* = (g^{dsid_{\mathrm{usr}}^*} \cdot h^{open^*}, g^{open^*})$ for a random $open^* \leftarrow \mathbb{Z}_p$. It sends $C_{\mathrm{usr}}{}^*$ to the issuer.
- The issuer replies with a random $dsid_{\mathrm{prvdr}}^* \leftarrow \mathbb{Z}_p$ and a random challenge $\gamma \leftarrow \mathbb{Z}_p$. Both issuer and user compute $C_{\mathrm{dsid}}{}^* = (g^{dsid_{\mathrm{usr}}^*} \cdot h^{open^*} \cdot g^{dsid_{\mathrm{prvdr}}^*}, g^{open^*})$.
- The user prepares new values $dsid^* = dsid_{\mathrm{usr}}^* + dsid_{\mathrm{prvdr}}^*$ and $dsrnd^* \leftarrow \mathbb{Z}_p$ for his next token and computes $Dsid^* = w^{dsid^*}$ and $C = (g^{y_1})^{usk} \cdot (g^{y_2})^{dsid^*} \cdot (g^{y_3})^{dsrnd^*} \cdot (g^{y_4})^{v-k} \cdot g^{r_C}$ for a random $r_C \leftarrow \mathbb{Z}_p$.
- The user computes $c = usk \cdot \gamma + dsrnd$.
- The user encrypts $Dsid^*$ as $ctrace = (w^r, (w^r)^{usk} \cdot Dsid^*)$ for a random $r \leftarrow \mathbb{Z}_p$.
- The user randomizes his credential $(\sigma_0', \sigma_1') = (\sigma_0^{r''}, (\sigma_1 \cdot \sigma_0^{r'})^{r''})$ for $r'' \leftarrow \mathbb{Z}_p^*$, $r' \leftarrow \mathbb{Z}_p$
- The user sends $C, c, ctrace, \sigma_0', \sigma_1'$ to the issuer and proves

$$\begin{aligned}\mathsf{ZKAK}[(usk, dsrnd, v, dsid^*, dsrnd^*, r', r, r_C, open^*);\\ c = usk \cdot \gamma + dsrnd\\ \wedge \mathsf{Vrfy}_{\mathrm{PS}}(pp, pk_{\mathrm{PS}}, (usk, dsid, dsrnd, v), (\sigma_0', \sigma_1' \cdot (\sigma_0')^{-r'})) = 1\\ \wedge v \geq k\\ \wedge ctrace = (w^r, (w^r)^{usk} \cdot w^{dsid^*})\\ \wedge C = (g^{y_1})^{usk} \cdot (g^{y_2})^{dsid^*} \cdot (g^{y_3})^{dsrnd^*} \cdot (g^{y_4})^{v-k} \cdot g^{r_C}\\ \wedge C_{\mathrm{dsid}}{}^* = (g^{dsid^*} \cdot h^{open^*}, g^{open^*})]\end{aligned}$$

  If the proof fails, the issuer aborts and outputs $(0, \perp)$.
- If the proof accepts, the issuer sends $\sigma_{\mathrm{PS}}'' = (\sigma_0'', \sigma_1'') = (g^{r''''}, (C \cdot g^x)^{r''''})$ for a random $r'''' \leftarrow \mathbb{Z}_p^*$ to the user and outputs $(1, dstag = (c, \gamma, ctrace))$.
- The user unblinds the signature as $\sigma_{\mathrm{PS}}^* = (\sigma_0'', \sigma_1'' \cdot (\sigma_0'')^{-r_C})$.
- The user checks $\mathsf{Vrfy}_{\mathrm{PS}}(pp, pk_{\mathrm{PS}}, (usk, dsid^*, dsrnd^*, v-k), \sigma_{\mathrm{PS}}^*) \overset{!}{=} 1$. If the check succeeds, it outputs $token^* = (dsid^*, dsrnd^*, v - k, \sigma_{\mathrm{PS}}^*)$ and $Dsid^*$, otherwise it outputs $\perp$.

$\mathsf{Link}(ispp, dstag, dstag') \rightarrow (upk, dslink)$ given $dstag = (c, \gamma, ctrace)$ and $dstag' = (c', \gamma', ctrace')$, outputs $dslink = (c - c')/(\gamma - \gamma')$ and $upk = w^{dslink}$.

$\mathsf{Trace}(ispp, dslink, dstag) \rightarrow Dsid^*$ for $dstag = (c, \gamma, (ctrace_0, ctrace_1))$ computes $Dsid^* = ctrace_1 \cdot ctrace_0^{-dslink}$.

$\mathsf{VrfyDs}(ispp, dslink, upk) \rightarrow b$ outputs 1 iff $w^{dslink} = upk$.

F Security Proofs for the Incentive System

F.1 Correctness in the Presence of Adversarial Users

For completeness, we define correctness in the presence of adversarial users, which rules out that adversarial users can interfere with operations between honest users and an honest provider.

Definition 25 (Correctness in the presence of adversarial users)**.** Let Π be an incentive system. Consider the following experiment $\mathrm{Exp}^{\text{adv-corr}}(\Pi, \mathcal{A}, \lambda)$:

- The experiment sets up $\mathit{ispp} \leftarrow \mathsf{Setup}(\mathcal{G}(1^\lambda))$ and calls the oracle $\mathit{pk} \leftarrow \mathbf{IssuerKeyGen}()$. It hands ispp and pk to $\mathcal{A}$.
- $\mathcal{A}$ may query the following oracles
  - $\mathbf{Issue}(\cdot)$
  - $\mathbf{Credit}(\cdot)$
  - $\mathbf{Deduct}(\cdot)$
  - $\mathbf{DBsync}(\cdot)$
  - $\mathbf{Keygen}()$
  - $u \mapsto \mathbf{Join}(u, \mathit{pk}) \leftrightarrow \mathbf{Issue}(\mathit{upk}_u)$
  - $(u, k) \mapsto \mathbf{Earn}(u, k) \leftrightarrow \mathbf{Credit}(k)$
  - $(u, k) \mapsto \mathbf{Spend}(u, k) \leftrightarrow \mathbf{Deduct}(k)$
- Eventually, $\mathcal{A}$ halts.
- The experiment outputs 1 iff $\mathcal{DB}$ contains some current dsid_u of some honest user u (i.e. user u's next spend operation would be detected double-spending as dsid_u is already in $\mathcal{DB}$).

We say that Π has *correctness in the presence of adversarial users* if for all ppt $\mathcal{A}$, there exists a negligible function negl s.t. $\Pr[\mathrm{Exp}^{\text{adv-corr}}(\Pi, \mathcal{A}, \lambda) = 1] \leq \mathit{negl}(\lambda)$ for all λ.

Note that correctness in the presence of adversarial users is not implied by correctness, soundness and framing resistance. Correctness does not imply anything for the case in which there are adversarial users. Framing resistance implies that u cannot be blamed for the double-spending (it may still happen that the online double-spending prevention prevents u from spending his coins). Soundness implies that after u spends his coin, *someone* can be blamed for it. This does not rule out that a corrupted user is able to inject u's dsid into $\mathcal{DB}$ while taking the blame. However, this would essentially constitute a denial of service attack on u, which is why correctness in the presence of adversarial users is a desirable property.

Theorem 26. If $\mathbb{Z}_p$ is super-poly, then Π_{insy} (Construction 23) is correct in the presence of adversarial users (Definition 25).

Proof. Assume there are k dsid entries in $\mathcal{DB}$ and ℓ honest users u at the point where $\mathcal{A}$ halts. For honest users, dsid_u is uniformly random in $\mathbb{Z}_p$ by construction. Furthermore, $\mathcal{A}$'s view is independent of the current $(\mathit{dsid}_u)_{u \text{ honest}}$ as none of the oracles output any information about them. So the probability that some dsid_u is one of the k dsid in $\mathcal{DB}$ is at most $\ell \cdot k / |\mathbb{Z}_p|$, which is negligible as ℓ and k are polynomial and $|\mathbb{Z}_p|$ is super-poly. □

F.2 Incentive System Anonymity

In the following we proof Theorem 14. For the proof of the theorem we have to look at the experiment $\mathrm{Exp}^{\text{ano-X}}$ (Fig. 4) instantiated for the incentive system Π_{insy}. On a high level, in Π_{insy} the important information for anonymity are the user specific values. Ignoring the commitments and ciphertexts, we could solely rely on the simulatability of the protocols to proof the theorem. However, the commitment and encryption scheme only guarantees computationally hiding and key-ind. CPA security.

Lemma 27. If Π_{uacs} has simulation anonymity (Definition 5), then for all ppt algorithms $\mathcal{A}$ it holds that $|\Pr[\text{Exp}_0^{\textsf{ano-Earn}}(\Pi_{\text{insy}}, \mathcal{A}, \lambda) = 1] - \Pr[\text{Exp}_1^{\textsf{ano-Earn}}(\Pi_{\text{insy}}, \mathcal{A}, \lambda) = 1]| = 0$.

Proof. We have to show that we can simulate the experiment and especially the challenge phase independent of b. Since Π_{uacs} satisfies simulation anonymity (Definition 5), there are ppt algorithms $\mathfrak{S}_{\textsf{Setup}}, \mathfrak{S}_{\textsf{Receive}}, \mathfrak{S}_{\textsf{ShowPrv}}, \mathfrak{S}_{\textsf{UpdRcv}}$. Therefore, we can perfectly simulate the setup by running $\mathfrak{S}_{\textsf{Setup}}$. Next, observe that we can honestly execute the oracles as in the experiment, since we know all inputs of the users. In the challenge phase the experiment executes **Earn** $\leftrightarrow \mathcal{A}$, where **Earn** is an execution of $\textsf{UpdRcv}_{\text{uacs}} \leftrightarrow \mathcal{A}$. We can perfectly simulate **Earn** in the challenge phase independent of b by running $\mathfrak{S}_{\textsf{UpdRcv}} \leftrightarrow \mathcal{A}$. □

In the $\textsf{Spend}$ case of experiment $\text{Exp}_b^{\textsf{ano-Spend}}(\Pi_{\text{insy}}, \mathcal{A}, \lambda)$ we have to look at the $\textsf{Spend} \leftrightarrow \textsf{Deduct}$ protocol of Π_{insy} (Construction 23), since the setup and challenge phase of $\text{Exp}_b^{\textsf{ano-Spend}}$ executes $\textsf{Spend}(u_b, k) \leftrightarrow \mathcal{A}$. In the challenge phase the adversary $\mathcal{A}$ is asked to guess which of the users u_0, u_1 that he picked before executed the $\textsf{Spend}$ protocol.

Let us first state where Π_{cmt} and Π_{enc} are used. During $\textsf{Spend} \leftrightarrow \textsf{Deduct}$ the provider (in $\text{Exp}_b^{\textsf{ano-Spend}}$ the adversary) gets commitments (generated with Π_{cmt}) from the users during the combined generation of a fresh $\mathit{dsid}^* = \mathit{dsid}^*_{\text{usr}} + \mathit{dsid}^*_{\text{prvdr}}$, where the user commits to a $\mathit{dsid}^*_{\text{usr}} \leftarrow \mathbb{Z}_p$ and the provider provides his $\mathit{dsid}^*_{\text{prvdr}} \in \mathbb{Z}_p$.

Also during $\textsf{Spend} \leftrightarrow \textsf{Deduct}$, the user encrypts dsid^* under his user public key $\mathit{upk}_{\text{enc}}$ as $\mathit{ctrace} \leftarrow \textsf{Encrypt}_{\text{enc}}(\mathit{pp}, \mathit{upk}_{\text{enc}}, \mathit{dsid}^*)$. Here the adversary could break anonymity by distinguishing which user public key was used to encrypt or the breaks CPA security.

Remember that the adversary $\mathcal{A}$ in $\text{Exp}_b^{\textsf{ano-Spend}}(\Pi_{\text{insy}}, \mathcal{A}, \lambda)$ can query the spend oracle **Spend**(u, k) for user u and spend value k in the setup and challenge phase. In each of the oracle executions he learns the dsid of the token that the user spends. This means that $\textsf{Spend}$ executions in the setup phase and the execution in the challenge phase are implicitly linked. In detail, $\mathcal{A}$ chooses users u_0, u_1 in the challenge phase. Then in the challenge phase $\textsf{Spend}$, $\mathcal{A}$ learns the dsid^*_b to a commitment C^*_b and encryption ctrace_b he received during the last $\textsf{Spend}$ execution in the setup phase with either u_0 or u_1. If he could link the information, he would break anonymity. Let us quickly deal with the easy case where $\mathcal{A}$ never triggered a spend operation during the setup phase, then the dsid^* that he gets during the challenge $\textsf{Spend}$ is a fresh random value from $\mathbb{Z}_p$ w.h.p..

For the rest of the proof we will change the challenge phase. In detail, we change in the challenge $\textsf{Spend}$ execution which dsid^*_b the adversary $\mathcal{A}$ gets (index i in the following) and how the encryption ctrace that $\mathcal{A}$ receives is generated (index j in the following). Therefore, we define experiments $H_{i,j}$ where $i, j \in \{0, 1\}$.

Let $H_{0,0} = \text{Exp}_0^{\textsf{ano-Spend}}(\Pi_{\text{insy}}, \mathcal{A}, \lambda)$ and $H_{1,1} = \text{Exp}_1^{\textsf{ano-Spend}}(\Pi_{\text{insy}}, \mathcal{A}, \lambda)$. In $H_{0,0}$ the adversary gets in the challenge phase one execution of $\textsf{Spend}$ with user u_0 where $\mathcal{A}$ receives dsid^*_0 ($j = 0$). Therefore, the only important $\textsf{Spend}$ execution of the setup phase is the last execution with user u_0 ($i = 0$) where $\mathcal{A}$ gets the commitment $C^*_0 = \textsf{Commit}_{\text{cmt}}(\mathit{pp}, \mathit{pk}_{\text{cmt}}, \mathit{dsid}^*_{\text{usr}})$ and encryption $\mathit{ctrace}_0 = \textsf{Encrypt}_{\text{enc}}(\mathit{pp}, \mathit{upk}_{\text{enc},0}, \mathit{dsid}^*_0)$, where $\mathit{dsid}^*_0 = \mathit{dsid}^*_{\text{usr}} + \mathit{dsid}^*_{\text{prvdr}}$. $H_{1,1}$ is analogous. To show $|[\Pr[H_{0,0} = 1] - \Pr[H_{1,1} = 1]]| \leq \mathit{negl}$ we also define an intermediate experiment $H_{0,1}$ and prove that $|\Pr[H_{0,0} = 1] - \Pr[H_{0,1} = 1]| \leq \mathit{negl}$ and $|\Pr[H_{0,1} = 1] - \Pr[H_{1,1} = 1]| \leq \mathit{negl}$. In $H_{0,1}$ we output dsid^*_0 in the challenge $\textsf{Spend}$ execution, where dsid^*_0 was determined and used in the previous $\textsf{Spend}$ execution (part of the setup phase) with user u_0. The change is that we no longer also output an encryption of dsid^*_0 under $\mathit{upk}_{\text{enc},0}$. Instead, we output $\mathit{ctrace}' = \textsf{Encrypt}_{\text{enc}}(\mathit{pp}, \mathit{upk}_{\text{enc},1}, \mathit{dsid}^*_1)$, where dsid^*_1 was determined and used in the previous $\textsf{Spend}$ execution (part of the setup phase) with user u_1. The public key $\mathit{upk}_{\text{enc},1}$ is also the one of user u_1.

Lemma 28. If Π_{uacs} has simulation anonymity and Π_{enc} is key-ind. CPA secure, then for all ppt $\mathcal{A}$, $|(\Pr[H_{0,0}(\mathcal{A}) = 1] - \Pr[H_{0,1}(\mathcal{A}) = 1])| = \mathit{negl}(\lambda)$ for all $\lambda \in \mathbb{N}$.

Proof. Assume that adversary $\mathcal{A}$ distinguishes $H_{0,0}, H_{0,1}$ with non-negligible probability. We give a reduction B using $\mathcal{A}$ to key-indistinguishable CPA security (Definition 19) of Π_{enc}. In the reduction B we get from $\text{Exp}_b^{\text{key-ind}}(\Pi_{\text{enc}}, \mathcal{A}, \lambda)$ $(b \in \{0,1\})$ two public keys that B injects as two user public keys $upk_{\text{enc},0}$ and $upk_{\text{enc},1}$ by guessing one pair of the users that $\mathcal{A}$ can choose in the challenge phase of $H_{0,b}$. Since Π_{uacs} guarantees simulation anonymity (Definition 5) and $c = usk \cdot \gamma + dsrnd$ is perfectly hiding, the reduction B can simulate the setup of the incentive system and the oracles $\textbf{Keygen}, \textbf{Join}, \textbf{Earn}, \textbf{Spend}$ of $H_{0,b}$ for two users u_0, u_1 that we choose before. For all other users B executes the oracles honestly as in the experiment. If $\mathcal{A}$ outputs two users that are not our guess u_0, u_1, then B aborts. This happens with probability $1 - \frac{1}{poly(\lambda)^2}$. Otherwise, in the challenge phase with $\mathcal{A}$, Spend is changed in B as described above. In detail, B gives the $\text{Exp}_b^{\text{key-ind}}$ challenger $dsid_0^*$ and $dsid_1^*$ (both from the latest token of the users u_0, u_1 from the setup phase) and outputs the answer of the challenger as the encryption for $\mathcal{A}$. Eventually, $\mathcal{A}$ outputs his guess $\hat{b}$ which B outputs to the $\text{Exp}_b^{\text{key-ind}}$ challenger. Consequently, $|\Pr[\text{Exp}_0^{\text{key-ind}}(\Pi_{\text{enc}}, B, \lambda) = 1] - \Pr[\text{Exp}_1^{\text{key-ind}}(\Pi_{\text{enc}}, B, \lambda) = 1]| = \frac{1}{poly(\lambda)^2} \cdot |(\Pr[H_{0,0}(\mathcal{A}) = 1] - \Pr[H_{0,1}(\mathcal{A}) = 1])|$. □

Next, we look at $|\Pr[H_{0,1} = 1] - \Pr[H_{1,1} = 1]|$. From $H_{0,1}$ to $H_{1,1}$ we change which $dsid_b^*$ the adversary receives during the challenge Spend execution. Either $dsid_0^*$ that is part of the latest token of the user u_0 or $dsid_1^*$ from the latest token of user u_1. As described above the adversary receives commitments and encryptions for $dsid_b^*$ corresponding to the latest token of the users. Remember, $H_{1,1} = \text{Exp}_1^{\text{ano-}\mathsf{Spend}}(\Pi_{\text{insy}}, \mathcal{A}, \lambda)$.

Lemma 29. If Π_{uacs} guarantees simulation anonymity, Π_{enc} is key-indistinguishable CPA secure, Π_{cmt} is computational hiding, then for all ppt adversaries $\mathcal{A}$ it holds that $|(\Pr[H_{0,1}(\mathcal{A}) = 1] - \Pr[H_{1,1}(\mathcal{A}) = 1])| = negl$

Lemma 29 follows from the following lemmas. First, we define a helper experiment $G_{u,v,x,y}^b(D, \lambda)$ for $u, v, x, y \in \{0,1\}$ that we will use in the following lemmas.

$\mathbf{G^b_{u,v,x,y}(D}, \lambda)$:
- $pp \leftarrow \mathcal{G}(1^\lambda)$
- $pk_{\text{cmt}} \leftarrow \mathsf{KeyGen}_{\text{cmt}}(pp)$
- $sk_0, sk_1 \leftarrow \mathsf{KeyGen}_{\text{enc}}()$ and $pk_0, pk_1 \leftarrow \mathsf{ComputePK}_{\text{enc}}(pp)$
- Hand D $pp, pk_{\text{cmt}}, pk_0$, and pk_1
- Choose two messages $m_0, m_1 \leftarrow M_{pp}$

Phase 1:
- Hand D the commitment C_u where $\mathsf{Commit}_{\text{cmt}}(pp, pk_{\text{cmt}}, m_u) \rightarrow (C_u, \mathsf{Open})$
- Receive $\mathsf{share} \in M_{pp}$ from D
- Hand D the encryption $S_v \leftarrow \mathsf{Encrypt}_{\text{enc}}(pp, pk_v, m_v + \mathsf{share})$

Phase 2:
- Hand D the commitment C_x where $\mathsf{Commit}_{\text{cmt}}(pp, pk_{\text{cmt}}, m_x) \rightarrow (C_x, \mathsf{Open})$
- Receive $\mathsf{share}' \in M_{pp}$ from D
- Hand D the encryption $S_y \leftarrow \mathsf{Encrypt}_{\text{enc}}(pp, pk_y, m_y + \mathsf{share}')$

Challenge:
- Hand D message m_b
- Receive $\hat{b}$ from D
- Output 1 iff $\hat{b} = b$

Lemma 30. If Π_{uacs} guarantees simulation anonymity, Π_{enc} is key-indistinguishable CPA secure, Π_{cmt} is computational hiding, then there is an ppt reduction D such that for all ppt adversaries $\mathcal{A}$ it holds that $|\Pr[G^0_{0,0,1,1}(D,\lambda)=1]-\Pr[G^1_{0,0,1,1}(D,\lambda)=1]|=\frac{1}{poly\lambda}\cdot|\Pr[H_{0,1}(\mathcal{A})=1]-\Pr[H_{1,1}(\mathcal{A})=1]|$.

Proof. Assume that an adversary $\mathcal{A}$ distinguishes $H_{0,1}$ and $H_{1,1}$, then we can give an reduction D that distinguishes $G^0_{0,0,1,1}(D,\lambda)$ and $G^1_{0,0,1,1}(D,\lambda)$.

In the following we define D against $G^b_{0,0,1,1}(D,\lambda)$ using $\mathcal{A}$. To shorten the proof, in D the guessing of two users u_0, u_1 to inject the public keys given by $G^b_{0,0,1,1}(D,\lambda)$ and the handling of the oracle queries is analogous to B. The last **Spend** query by $\mathcal{A}$ for user u_0 is answered with the help of Phase 1 in $G^b_{0,0,1,1}(D,\lambda)$ and the rest of Spend simulated. From Phase 1 D uses the commitment C_u instead of generating a commitment to a fresh $dsid_{\text{usr}}$. In Spend, $\mathcal{A}$ hands D (acting as the user) a $dsid_{\text{prvdr}}$ that D hands itself to $G^b_{0,0,1,1}(D,\lambda)$ (Phase 1) as share. The encryption that D gets from Phase 1 is used as the encryption $ctrace$ in Spend. For the last **Spend** query by $\mathcal{A}$ for user u_1 reduction D acts analogous with the difference that D uses Phase 2. Eventually $\mathcal{A}$ enters the challenge phase and outputs two user handles. If the handles are not the one that D guessed, then abort. Otherwise, D simulates a Spend with $\mathcal{A}$ where D is supposed to send $\mathcal{A}$ the $dsid$ of the latest token of the challenged user. Hence, D sends the message m_b that D received in the challenge phase of $G^b_{0,0,1,1}(D,\lambda)$ instead. If $\mathcal{A}$ outputs $\hat{b}$, D also outputs $\hat{b}$ to $G^b_{0,0,1,1}(D,\lambda)$. Overall, $|\Pr[G^0_{0,0,1,1}(D,\lambda)=1]-\Pr[G^1_{0,0,1,1}(D,\lambda)=1]|=\frac{1}{poly\lambda}\cdot|\Pr[H_{0,1}(\mathcal{A})=1]-\Pr[H_{1,1}(\mathcal{A})=1]|$. □

It is left to show that for all ppt algorithms E it holds that $|\Pr[G^0_{0,0,1,1}(E,\lambda)=1]-\Pr[G^1_{0,0,1,1}(E,\lambda)=1]|\leq negl$. Remember, in $G^b_{u,v,x,y}(D,\lambda)$ the bits (u,v) determine Phase 1, (x,y) Phase 2, and b determines the output message m_b at the end of the experiment. In detail, the bits u and x determine the messages for the commitments; the bits v and y determine the messages and public keys for the encryption. In Figure 7 we show an overview of the following proof steps, where Phase 1 and Phase 2 points to the point where we introduce a change to the previous game and "key-ind. CPA" respectively "comp. hiding" is the security guarantee that we use in the reduction.

Lemma 31. It holds that $G^0_{1,1,0,0}=G^1_{0,0,1,1}$.

Proof. This is the last step presented in Figure 7. The lemma follows from the following observation. Since the experiment $G_{u,v,x,y}$ chooses the challenge messages itself, the order of the Phases can be switched without changing the game while also changing the challenge message from m_0 to m_1. Changing order of the Phases is the same as replacing m_u, m_v, m_x, and m_y by m_{1-u}, m_{1-v}, m_{1-x}, and m_{1-y}. □

Lemma 32. If Π_{enc} is key-indistinguishable CPA secure, then for all ppt adversaries E it holds that $|\Pr[G^0_{0,0,1,1}(E,\lambda)=1]-\Pr[G^0_{0,1,1,1}(E,\lambda)=1]|=negl(\lambda)$.

Proof. We show that if there is an adversary E s.t. $|\Pr[G^0_{0,0,1,1}(E,\lambda)=1]-\Pr[G^0_{0,1,1,1}(E,\lambda)=1]|$ is non-negligible, than we can give an reduction $R^{\text{Phase 1}}_{\text{ki-cpa}}$ that breaks key-ind. CPA (Definition 19, $\text{Exp}^{\text{key-ind}}_b(\Pi_{\text{enc}}, R^{\text{Phase 1}}_{\text{ki-cpa}}, \lambda)$) using E.

$R^{\text{Phase 1}}_{\text{ki-cpa}}$ gets from its experiment $\text{Exp}^{\text{key-ind}}_b$ public parameters pp and two encryption scheme public keys pk_0, pk_1. $R^{\text{Phase 1}}_{\text{ki-cpa}}$ generates honestly a commitment public key pk_{cmt} and hands E pk_{cmt} and the received pp, pk_0, pk_1 as in $G^0_{0,v,1,1}$. Next, $R^{\text{Phase 1}}_{\text{ki-cpa}}$ chooses two messages $m_0, m_1 \leftarrow M_{pp}$,

$$\begin{tikzcd}
G^0_{0,0,1,1} \arrow[r, leftrightarrow, "\text{Phase 1}", "\text{key-ind. CPA}"'] & G^0_{0,1,1,1} \arrow[r, leftrightarrow, "\text{Phase 2}", "\text{key-ind. CPA}"'] & G^0_{0,1,1,0} \arrow[r, leftrightarrow, "\text{Phase 1}", "\text{comp. hiding}"'] & G^0_{1,1,1,0} \arrow[r, leftrightarrow, "\text{Phase 2}", "\text{comp. hiding}"'] & G^0_{1,1,0,0} \arrow[r, leftrightarrow, "", "\equiv"'] & G^1_{0,0,1,1}
\end{tikzcd}$$

Figure 7: Sequence of games for anonymity proof

generates a commitment to m_0 for Phase 1 as in $G^0_{0,v,1,1}$. $R^{\text{Phase 1}}_{\text{ki-cpa}}$ sends the Phase 1 commitment to E and gets $\mathsf{share} \in M_{pp}$ back. Next, $R^{\text{Phase 1}}_{\text{ki-cpa}}$ hands $m^*_0 = m_0 + \mathsf{share}$ and $m^*_1 = m_1 + \mathsf{share}$ to $\text{Exp}^{\text{key-ind}}_b$ and gets $S \leftarrow \mathsf{Encrypt}_{\text{enc}}(pp, pk_b, m^*_b)$ back. Phase 2 is executed by $R^{\text{Phase 1}}_{\text{ki-cpa}}$ as in the experiment $G^0_{0,v,1,1}$ (commitment to m_1 and encryption of $m_1 + \mathsf{share}'$). In the challenge phase, $R^{\text{Phase 1}}_{\text{ki-cpa}}$ hands the message m_0 to E and receives E's guess $\hat{b}$ that $R^{\text{Phase 1}}_{\text{ki-cpa}}$ also outputs to $\text{Exp}^{\text{key-ind}}_b$.

Observe that $R^{\text{Phase 1}}_{\text{ki-cpa}}$ perfectly simulates the view of E in $G^0_{0,0,1,1}$ if $R^{\text{Phase 1}}_{\text{ki-cpa}}$ acts in the experiment $\text{Exp}^{\text{key-ind}}_0$. The same holds for the view of E in $G^0_{0,1,1,1}$ if $R^{\text{Phase 1}}_{\text{ki-cpa}}$ acts in the experiment $\text{Exp}^{\text{key-ind}}_1$. Consequently, it holds that $|\Pr[\text{Exp}^{\text{key-ind}}_0(\Pi_{\text{enc}}, R^{\text{Phase 1}}_{\text{ki-cpa}}, \lambda) = 1] - \Pr[\text{Exp}^{\text{key-ind}}_1(\Pi_{\text{enc}}, R^{\text{Phase 1}}_{\text{ki-cpa}}, \lambda) = 1]| = |\Pr[G^0_{0,0,1,1}(\mathcal{E}, \lambda) = 1] - \Pr[G^0_{0,1,1,1}(\mathcal{E}, \lambda) = 1]|$. □

Lemma 33. If Π_{enc} is key-indistinguishable CPA secure, then for all ppt adversaries E it holds that $|\Pr[G^0_{0,1,1,1}(E, \lambda) = 1] - \Pr[G^0_{0,1,1,0}(E, \lambda) = 1]| = \mathit{negl}(\lambda)$.

The reduction $R^{\text{Phase 2}}_{\text{ki-cpa}}$ to show the above lemma works analogous to $R^{\text{Phase 1}}_{\text{ki-cpa}}$ with the difference that in $R^{\text{Phase 2}}_{\text{ki-cpa}}$ we use the encryption challenge of $\text{Exp}^{\text{key-ind}}_b(\Pi_{\text{enc}}, R^{\text{Phase 1}}_{\text{ki-cpa}}, \lambda)$ in Phase 2.

Lemma 34. If Π_{cmt} is comp. hiding, then for all ppt adversaries F it holds that $|\Pr[G^0_{0,1,1,0}(F, \lambda) = 1] - \Pr[G^0_{1,1,1,0}(F, \lambda) = 1]| = \mathit{negl}$.

Proof. In the following we show that if there is an adversary F such that $|\Pr[G^0_{0,1,1,0}(F, \lambda) = 1] - \Pr[G^0_{1,1,1,0}(F, \lambda) = 1]|$ is non-negligible, than we can give an reduction $R^{\text{Phase 1}}_{\text{hiding}}$ that breaks comp. hiding of the commitment scheme Π_{cmt} (Definition 21, $\text{Exp}^{\text{hid}}_b(\Pi_{\text{cmt}}, R^{\text{Phase 1}}_{\text{hiding}}, \lambda)$) using F.

$R^{\text{Phase 1}}_{\text{hiding}}$ gets from its experiment $\text{Exp}^{\text{hid}}_b$ public parameters pp and a commitment scheme public key pk_{cmt}. $R^{\text{Phase 1}}_{\text{hiding}}$ generates honestly two encryption public keys pk_0, pk_1 as in $G^0_{u,1,1,0}$. hands F pk_0, pk_1 and the received pp, pk as in $G^0_{u,1,1,0}$. Next, $R^{\text{Phase 1}}_{\text{hiding}}$ chooses two messages $m_0, m_1 \leftarrow M_{pp}$, hands both to $\text{Exp}^{\text{hid}}_b$, and gets $C_b \leftarrow \mathsf{Commit}_{\text{cmt}}(pp, pk_{\text{cmt}}, m_b)$ back. $R^{\text{Phase 1}}_{\text{hiding}}$ outputs C_b as the Phase 1 commitment to F. The rest of Phase 1 and Phase 2 are executes by $R^{\text{Phase 1}}_{\text{hiding}}$ honestly as in the experiment $G^0_{u,1,1,0}$. In the challenge phase, $R^{\text{Phase 1}}_{\text{hiding}}$ hands F the message m_b and receives F's guess $\hat{b}$. $R^{\text{Phase 1}}_{\text{hiding}}$ also outputs $\hat{b}$ to $\text{Exp}^{\text{hid}}_b$. Observe that $R^{\text{Phase 1}}_{\text{hiding}}$ perfectly simulates the view of F in $\text{Exp}^{\text{hid}}_b$. Consequently, $|\Pr[\text{Exp}^{\text{hid}}_0(\Pi_{\text{cmt}}, R^{\text{Phase 1}}_{\text{hiding}}, \lambda) = 1] - \Pr[\text{Exp}^{\text{hid}}_1(\Pi_{\text{cmt}}, R^{\text{Phase 1}}_{\text{hiding}}, \lambda) = 1]| = |\Pr[G^0_{0,1,1,0}(F, \lambda) = 1] - \Pr[G^0_{1,1,1,0}(F, \lambda) = 1]|$. □

Lemma 35. If Π_{cmt} is comp. hiding, then for all ppt adversaries F it holds that $|\Pr[G^0_{1,1,1,0}(F, \lambda) = 1] - \Pr[G^0_{1,1,0,0}(F, \lambda) = 1]| = \mathit{negl}$.

The reduction $R^{\text{Phase 2}}_{\text{hiding}}$ to show the above lemma works analogous to $R^{\text{Phase 1}}_{\text{hiding}}$ with the difference that in $R^{\text{Phase 2}}_{\text{hiding}}$ we use the commitment challenge of $\text{Exp}^{\text{hid}}_b(\Pi_{\text{enc}}, R^{\text{Phase 2}}_{\text{hiding}}, \lambda)$ in Phase 2.

This concludes the proof of Lemma 29 and therefore of Theorem 14.

F.3 Incentive System Soundness

Theorem 15. Let $\mathcal{A}$ be an attacker against incentive system soundness of Construction 23. We construct $\mathcal{B}$ against updatable credential soundness of Π_{uacs}.

- $\mathcal{B}$ receives cpp from its challenger. $\mathcal{B}$ replies with 1^4 to receive pk. It completes the setup by choosing $pk_{\text{cmt}} \leftarrow \mathsf{KeyGen}_{\text{cmt}}(pp)$. Then $\mathcal{B}$ simulates the query to **IssuerKeyGen**(): instead of running $\mathsf{IssuerKeyGen}$, $\mathcal{B}$ uses its challenger's key pk as the query result. $\mathcal{B}$ outputs $ispp = (pp, cpp, pk_{\text{cmt}})$ and pk to $\mathcal{A}$.

- Oracle queries by $\mathcal{A}$ are simulated by $\mathcal{B}$ as prescribed by the protocol with one exception: whenever the original protocol would run $\mathsf{Issue}_{\text{uacs}}$ or $\mathsf{Update}_{\text{uacs}}$, $\mathcal{B}$ instead queries its challenger for the corresponding operation and relays protocol messages between the challenger and $\mathcal{A}$.

- Eventually $\mathcal{A}$ halts. Then $\mathcal{B}$ halts as well.

Obviously, the view of $\mathcal{A}$ is the same whether it interacts with the incentive system soundness challenger or with $\mathcal{B}$.

Let error be the event that (1) $\mathcal{B}$ has output the same challenge δ in two different **Deduct** runs, or (2) there were two commitments $C_{\text{dsid}}, C_{\text{dsid}}'$ in two runs of **Deduct** or **Issue** such that the commitments can be opened to two different messages. Note that cmt is perfectly binding by assumption and so every commitment opens to at most one value (which $\mathcal{B}$ cannot necessarily efficiently compute, but as an event, this is well-defined).

It holds that $\Pr[\text{error}] \leq \mathit{negl}(\lambda)$ because (1) in each **Deduct** query, δ is chosen uniformly random by $\mathcal{B}$ from the super-poly size set $\mathbb{Z}_p$, and (2) C_{dsid} is the result of an $\mathsf{Add}_{\text{cmt}}$ operation with a uniformly random value $\mathit{dsid}_{\text{prvdr}} \leftarrow \mathbb{Z}_p$ chosen by $\mathcal{B}$, hence it opens (only) to a uniformly random value.

Let $\mathcal{A}\text{wins}_{\text{trace}}$ be the event that $\mathcal{DB}$ contains some $(\mathit{upk}, \mathit{dslink})$ s.t. $\mathsf{VrfyDs}(\mathit{ispp}, \mathit{dslink}, \mathit{upk}) \neq 1$ or $\mathit{upk} \notin \mathcal{U}$. Let $\mathcal{A}\text{wins}_{\text{overspend}}$ be the event that $v_{\text{spent}} - v_{\text{invalid}} > v_{\text{earned}}$ and $\mathsf{DBsync}(s)$ has been queried for all spend handles s. Let $\mathcal{A}$wins be the event that $\mathcal{A}$ wins the game, $\mathcal{A}\text{wins}_{\text{trace}} \vee \mathcal{A}\text{wins}_{\text{overspend}}$. Lemma 36 will show that if $\mathcal{A}\text{wins} \wedge \neg\text{error}$ occurs, then there *exists* no explanation list $\mathcal{L}$ that is consistent, implying $\Pr[\text{Exp}^{\text{sound}}(\Pi, \mathcal{B}, \mathcal{E}, \lambda) = 1 \mid \mathcal{A}\text{wins} \wedge \neg\text{error}] = 1$. Overall, let $\mathcal{E}$ be an algorithm, then

$$\begin{aligned}
&\Pr[\text{Exp}^{\text{sound}}(\Pi_{\text{uacs}}, \mathcal{B}, \mathcal{E}, \lambda) = 1] \\
\geq &\Pr[\text{Exp}^{\text{sound}}(\Pi_{\text{uacs}}, \mathcal{B}, \mathcal{E}, \lambda) = 1 \mid \mathcal{A}\text{wins} \wedge \neg\text{error}] \\
&\quad \cdot \Pr[\mathcal{A}\text{wins} \wedge \neg\text{error}] \\
= &1 \cdot \Pr[\mathcal{A}\text{wins} \wedge \neg\text{error}] \geq \Pr[\mathcal{A}\text{wins}] - \Pr[\text{error}] \\
= &\Pr[\text{Exp}^{\text{sound}}(\Pi_{\text{insy}}, \mathcal{A}, \lambda) = 1] - \Pr[\text{error}].
\end{aligned}$$

Consequently, because $\Pr[\text{Exp}^{\text{sound}}(\Pi_{\text{uacs}}, \mathcal{B}, \mathcal{E}, \lambda) = 1]$ is negligible by assumption, it follows that $\Pr[\text{Exp}^{\text{sound}}(\Pi_{\text{insy}}, \mathcal{A}, \lambda) = 1]$ must be negligible. □

Lemma 36. If $\mathcal{A}\text{wins} \wedge \neg\text{error}$, then no explanation list is consistent (cf. Theorem 15 and Definition 6).

Proof. We prove the statement by showing that if $\neg$error and there exists a consistent explanation list $\mathcal{L}$, then $\neg\mathcal{A}$wins. Let $\mathcal{L}$ be a consistent explanation list and let E_i be the corresponding sets of explained attribute vectors (cf. Definition 6).

For ease of reasoning in all the following lemmas, we represent the explanation list as a bipartite directed graph G (cf. Figure 8). The graph contains (1) one node $\vec{A}$ for every explained attribute vector $\vec{A} \in \bigcup_i E_i$ and (2) nodes for **Issue**, **Credit**, **Deduct** queries: If the ith query is an **Issue**(upk) query, there is a node i. If the ith query is a **Credit**(k) query for which the $\mathsf{Update}_{\text{uacs}}$ operation outputs 1 for the provider, there is a node i. If the ith query is an $s \leftarrow$ **Deduct**(k) query for which the $\mathsf{Update}_{\text{uacs}}$ operation outputs 1 for the provider, there is a node i.

An **Issue** node i has an outgoing edge to the attribute vector $\psi_i(\perp, \alpha_i)$, where ψ_i is the update function used within the ith query and α_i is as supplied by $\mathcal{L}$. A **Credit** or **Deduct** node i has an incoming edge from attribute vector $\vec{A}_i$ and an outgoing edge to $\psi_i(\vec{A}_i, \alpha_i)$, where ψ_i is the update function used within this query and $\vec{A}_i, \alpha_i$ are as supplied by $\mathcal{L}$. We call $\vec{A}_i$ the *predecessor* and $\psi_i(\vec{A}_i, \alpha_i)$ the *successor* of a **Credit Deduct** node i.

We say that a **Deduct** node i is marked *invalid* if its corresponding transaction in the double-spend database $\mathcal{DB}$ is marked invalid. Otherwise, the node is *valid*.

Lemma 37 shows that $\neg\mathcal{A}\text{wins}_{\text{trace}}$ and Lemma 38 shows that $\neg\mathcal{A}\text{wins}_{\text{overspend}}$. Hence $\neg\mathcal{A}\text{wins}$. ☐

For all of the following lemmas, we are in the setting of Lemma 36, i.e. we assume that $\neg$error happens and $\mathcal{L}$ is consistent.

Lemma 37. $\neg\mathcal{A}\text{wins}_{\text{trace}}$ holds (i.e. $\mathcal{DB}$ contains no $(upk, dslink)$ with $\mathsf{VrfyDs}(ispp, dslink, upk) \neq 1$ or $upk \notin \mathcal{U}$)

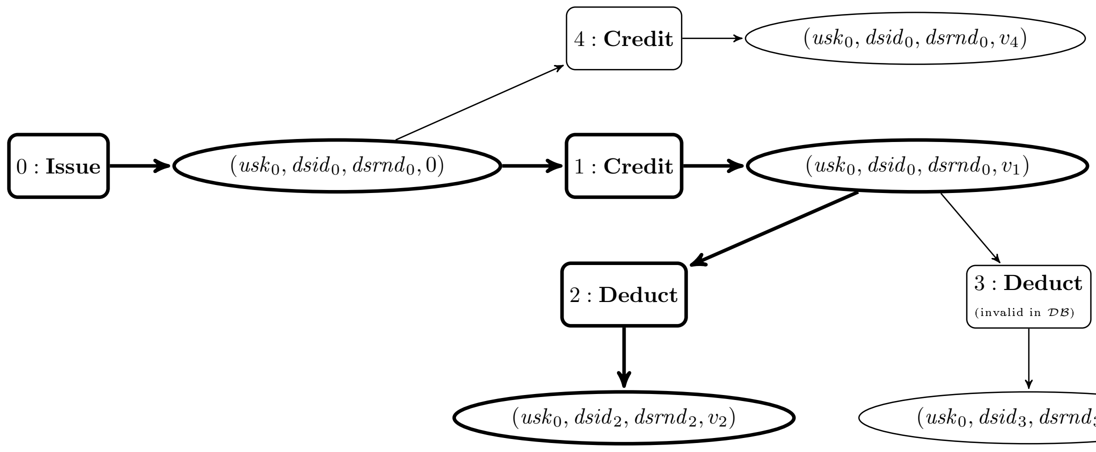
Proof. First, note that any output $(upk, dslink)$ of Link by definition fulfills the VrfyDs check. The "$upk \in \mathcal{U}$" part remains to be shown. Assume that at some point, $(upk, dslink)$ was associated with $dsid$ in $\mathcal{DB}$. Lemma 46 states there exists a node $\vec{A} = (usk, dsid, dsrnd, \cdot)$ in G with $usk = dslink$.

Lemma 42 implies that $\vec{A}$ is reachable from some **Issue** node. Let upk' be the public key added to $\mathcal{U}$ by the **Issue** oracle call. Let $\vec{A}' = (usk', dsid', dsrnd', 0)$ be the successor to that **Issue** node. Since $\vec{A}$ is reachable from $\vec{A}'$, we have $usk' = usk$ (no update function ever changes the user secret). Because the Issue update function checks $\mathsf{ComputePK}_{\text{enc}}(pp, usk') \overset{!}{=} upk'$, and $upk = \mathsf{ComputePK}(pp, dslink) = \mathsf{ComputePK}(pp, usk)$ by definition of DBsync we have $upk = upk'$. So it holds that upk was added to $\mathcal{U}$. ☐

Lemma 38. $\neg\mathcal{A}\text{wins}_{\text{overspend}}$ holds (i.e. $v_{\text{spent}} - v_{\text{invalid}} \leq v_{\text{earned}}$).

Proof. Assume that **DBsync**(s) has been queried for all spend handles s. For any subgraph H of G, we first define $v_{\text{spent}}(H) = \sum_{(i,\textbf{Deduct})\in H} k_i$, $v_{\text{earned}}(H) = \sum_{(i,\textbf{Credit})\in H} k_i$, and $v_{\text{invalid}}(H) = \sum_{(i,\textbf{Credit})\in H; i \text{ invalid}} k_i$. Note that these are consistent with $v_{\text{spent}}, v_{\text{earned}}, v_{\text{invalid}}$ in the incentive system soundness game, i.e. $v_{\text{spent}} = v_{\text{spent}}(G), v_{\text{earned}} = v_{\text{earned}}(G)$, and $v_{\text{invalid}} = v_{\text{invalid}}(G)$.

Every weakly connected component of G contains a simple path starting at an **Issue** node that contains all *valid* **Deduct** nodes within that component and no *invalid* **Deduct** nodes (Lemma 39). We obtain the subgraph G' of G as the (disjoint) union of these paths (one for each weakly connected component). As we have removed all invalid **Deduct** nodes but preserved all valid ones, we have $v_{\text{spent}}(G') = v_{\text{spent}}(G) - v_{\text{invalid}}(G)$. Because every weakly connected component G'' in G' is a path starting at an **Issue** node, we have that $v_{\text{spent}}(G'') \leq v_{\text{earned}}(G'')$ (Lemma 47). Because this holds for every weakly connected component G'' of G', we have $v_{\text{spent}}(G') \leq v_{\text{earned}}(G')$. Also, obviously $v_{\text{earned}}(G') \leq v_{\text{earned}}(G)$ by the subgraph property.



4 : Credit
$(usk_0, dsid_0, dsrnd_0, v_4)$
0 : Issue
$(usk_0, dsid_0, dsrnd_0, 0)$
1 : Credit
$(usk_0, dsid_0, dsrnd_0, v_1)$
2 : Deduct
3 : Deduct
(invalid in $\mathcal{DB}$)
$(usk_0, dsid_2, dsrnd_2, v_2)$
$(usk_0, dsid_3, dsrnd_3, v_3)$

Figure 8: Example explanation graph G as in Lemma 36 (but with only one user).
The bold graph elements form the "canonical" path (Lemma 39) containing all valid deduct operations; all other nodes are removed in Lemma 38, ensuring $v_{\text{spent}}(G') = v_{\text{spent}}(G) - v_{\text{invalid}}(G)$ and $v_{\text{earned}}(G') \leq v_{\text{earned}}(G)$.

Overall, $v_{\text{spent}}(G) - v_{\text{invalid}}(G) = v_{\text{spent}}(G') \leq v_{\text{earned}}(G') \leq v_{\text{earned}}(G)$ □

Lemma 39. Every weakly connected component of G contains a simple path containing all valid **Deduct** nodes within that component and no invalid **Deduct** nodes.

Proof. Let G' be a weakly connected component in G. G' contains a single **Issue** node by Lemma 42. Let j be the numerically largest index such that $j \in G'$ is a valid **Deduct** node (if no such j exists, the lemma's statement holds trivially). Because of Lemma 42, there exists a path P from the **Issue** node to j. We show that P contains all valid **Deduct** nodes and no invalid nodes.

Assume for contradiction that P contains an invalid node. Then j would be invalid as well (Lemma 43), as it is reachable from that invalid node. Hence P does not contain invalid nodes.

Assume for contradiction that some **Deduct** node j' is valid but not on P. j' is reachable from the **Issue** node (Lemma 42) via some path P'. P and P' start at the same node but j is not on P' (because $j' < j$ by maximal choice of j and operation indices are monotonously increasing on any path (Lemma 40). Because j' is not on P and j is not on P', neither path is a prefix of the other, so there exists a node that differs on the two paths. Let $\vec{A}$ be the last node on P and P' *before* the first node that differs (note that this must be an attribute vector node as the operation nodes have out-degree 1 by definition). Let i be the first **Deduct** node after $\vec{A}$ on P and let i' be the first **Deduct** node after $\vec{A}$ on P'. Because i and i' are the first **Deduct** operation on each path (i.e. only **Credit** operations happen between $\vec{A}$ and i or i'), we have that $dsid_i = dsid_{i'}$ (where $dsid_\ell$ is the $dsid$ that was revealed during the ℓth query). From the definition of DBsync, it is easy to see that i or i' must have been marked invalid (at most one transaction per $dsid$ is valid). Since i is on P, it is valid. Hence i' must be invalid. Because j' is reachable from i' (via P'), j' must be invalid (Lemma 43) □

Lemma 40. For any path in G, the indices of **Issue** and **Deduct** nodes on the path are strictly monotonously increasing.

Proof. Let P be a path and let j and i be **Deduct** nodes on the path in that order (in the following, j could also be an **Issue** node) so that there are no other **Deduct** nodes between them on the path. Assume for contradiction that $i \leq j$. Let $\psi_i, \vec{A}_i = (usk_i, dsid_i, dsrnd_i, v_i), \alpha_i$ be the "input" values for query i (as defined by $\mathcal{L}$). Because $\mathcal{L}$ is consistent, there is some **Issue** or **Deduct** node $i' < i$ that creates attribute vectors with $dsid_i$. However, there is a path from j to i that involves only **Credit** and attribute vector nodes. This implies that the $dsid$ in j's successor node is $dsid_i$. This means that $j \neq i'$ are associated with the same $dsid$, contradicting $\neg$error (cf. Lemma 36). □

Lemma 41. G is acyclic.

Proof. Assume there exists a cycle C. C cannot contain **Issue** nodes as they have in-degree 0. Because of Lemma 40, C cannot not contain any **Deduct** nodes. This means that the only oracle nodes on the cycle are **Credit** nodes. In turn, this implies that all $\vec{A} = (usk, dsid, dsrnd, v)$ nodes on the cycle share the same usk, $dsid$, $dsrnd$ (as those are not changed by **Credit**). **Credit** strictly increases v, but on a cycle we would have to see a **Credit** node that decreases v or leaves it unchanged. Hence there are also no **Credit** on the cycle. Overall, there are only attribute vector nodes on the cycle, but there are no edges between attribute vector nodes, contradicting the existence of the cycle. □

Lemma 42. Every weakly connected component of G contains exactly one **Issue** node. Furthermore, every node in G is reachable from (exactly one) **Issue** node.

Proof. Let v be a node in G. Because G is acyclic (Lemma 41), the process of walking edges backwards from v eventually stops. It cannot stop at an attribute vector node (since every attribute vector node has in-degree at least 1 by consistency of $\mathcal{L}$) and it cannot stop at a **Credit** or **Deduct** node (as those have in-degree 1), hence it must stop at an **Issue** node. So v can be reached from some **Issue** node.

Assume for contradiction that some weakly connected component contains two **Issue** nodes v_0, v_1. By choice of our update functions, all attribute vector nodes $\vec{A} = (\mathit{usk}, \mathit{dsid}, \mathit{dsrnd}, v)$ reachable from a **Issue** node have the same usk (because no update ever changes usk). Furthermore, there are no two **Issue** nodes with the same usk (since **Issue**(upk) can only be called once per upk and $\mathsf{ComputePK}$ is injective). As a consequence, every node is reachable from exactly one **Issue** node.

If we partition the attribute vector nodes in the weakly connected component into those that are reachable from v_0 and those that are reachable from v_1, there must be some path (of the form $\vec{A}_0 \to i \to \vec{A}_1$) from some $\vec{A}_0$ reachable from v_0 to some $\vec{A}_1$ reachable from v_1 or vice versa (otherwise the graph cannot be weakly connected). However, then $\vec{A}_1$ (or $\vec{A}_0$) is reachable from v_0 *and* from v_1, contradicting our previous result. This implies that every weakly connected component contains at most one **Issue** node. Furthermore, every weakly connected component contains at least one node, which is reachable from some **Issue**, meaning that it also contains *at least* one **Issue** node. □

Lemma 43. If **DBsync**(s) has been queried for all spend handles s, then every **Deduct** node that is reachable from an invalid **Deduct** node is invalid.

Proof. Let i, j be **Deduct** nodes such that i is invalid and j is reachable from i with no further **Deduct** nodes on the path P between i and j. If we can show that j is invalid, transitivity implies the statement for *all* j' reachable from i.

Because P does not contain (intermediate) **Deduct** nodes, all attribute vector nodes on P have the same $\mathit{usk}, \mathit{dsid}, \mathit{dsrnd}$. dsid is input to j's oracle query. Let $\mathit{dstag}_i = (c, \gamma, \mathit{ctrace})$ be the double-spend tag output by Deduct in query i. Because of the update function used in query i, it holds that $\mathsf{Decrypt}_{\mathrm{enc}}(\mathit{pp}, \mathit{usk}, \mathit{ctrace}) = \mathit{dsid}$.

We distinguish two cases: t_i is marked invalid before t_j was added to $\mathcal{DB}$ or vice versa. Assume t_i was marked invalid *before* t_j was added to the graph. When t_i was marked invalid, the successor node dsid is added to $\mathcal{DB}$ (Lemma 44). When t_j is added to the database afterwards, its input dsid is already in the database, hence t_j is immediately marked invalid. Assume that t_i was marked invalid *after* t_j was added to $\mathcal{DB}$. When t_j is added, dsid and an edge (dsid, t_j) is added to $\mathcal{DB}$. Afterwards, at some point t_i is marked invalid. During this process, the edge (t_i, dsid) is added to $\mathcal{DB}$ (Lemma 44) and because (dsid, t_j) is in the graph, t_j is marked invalid.

Hence in both cases, t_j is marked invalid at some point. □

Lemma 44. Let t_i be some transaction node in $\mathcal{DB}$ and let i be the corresponding **Deduct** node in G with successor $\vec{A}^* = (\mathit{usk}^*, \mathit{dsid}^*, \mathit{dsrnd}^*, \cdot)$. After t_i is marked invalid, the successor of t_i in $\mathcal{DB}$ is dsid^*.

Proof. Since t_i is marked invalid, t_i's predecessor dsid in $\mathcal{DB}$ is correctly associated with some $(\mathit{upk}, \mathit{dslink})$ (Lemma 46). In particular, i's predecessor $\vec{A} = (\mathit{usk}, \mathit{dsid}, \mathit{dsrnd}, v)$ in G must have $\mathit{dslink} = \mathit{usk}$. Let $\vec{A}^* = (\mathit{usk}^*, \mathit{dsid}^*, \mathit{dsrnd}^*, v^*)$ be i's successor. Let $\mathit{dstag} = (c, \gamma, \mathit{ctrace})$ be the dstag associated with i's oracle query. Because of consistency of $\mathcal{L}$, we have $\mathsf{Decrypt}_{\mathrm{enc}}(\mathit{pp}, \mathit{usk}, \mathit{ctrace}) = \mathit{dsid}^*$. When t_i is marked invalid, DBsync computes $\mathit{dsid}^* = \mathsf{Trace}(\mathit{pp}, \mathit{dslink}, \mathit{dstag}) = \mathsf{Decrypt}_{\mathrm{enc}}(\mathit{pp}, \mathit{usk}, \mathit{ctrace}) = \mathit{dsid}^*$ and makes dsid^* the successor to t_i. □

Lemma 45. For any two attribute vectors $\vec{A}_0 = (\mathit{usk}_0, \mathit{dsid}_0, \mathit{dsrnd}_0, v_0)$ and $\vec{A}_1 = (\mathit{usk}_1, \mathit{dsid}_1, \mathit{dsrnd}_1, v_1)$ in G, it holds that if $\mathit{dsid}_0 = \mathit{dsid}_1$, then $\mathit{usk}_0 = \mathit{usk}_1$ and $\mathit{dsrnd}_0 = \mathit{dsrnd}_1$.

Proof. Because of $\neg$error, there is a unique **Issue** or **Deduct** node i whose successor $\vec{A}^* = (\mathit{usk}^*, \mathit{dsid}^*, \mathit{dsrnd}^*, v_0)$ contains $\mathit{dsid}^* = \mathit{dsid}_0 = \mathit{dsid}_1$. Because i is unique in this regard, both $\vec{A}_0$ and $\vec{A}_1$ are reachable from i on paths P_0, P_1 that contains only **Credit** and attribute vector nodes. Since

Credit does not change *usk* or *dsrnd*, we get that $usk_0 = usk_1 = usk^*$ and $dsrnd_0 = dsrnd_1 = dsrnd^*$. □

Lemma 46. We say that a node *dsid* in $\mathcal{DB}$ is "correctly associated" with $(upk, dslink)$ if there exists $(usk', dsid', dsrnd', \cdot)$ in G with $dsid'$ and $dslink = usk'$ and for all $(usk', dsid', dsrnd', \cdot)$ in G with $dsid'$, we have that $dslink = usk'$. All nodes *dsid* in $\mathcal{DB}$ that have some value associated with them are correctly associated.

Proof. Let *dsid* be some node in $\mathcal{DB}$ that has been associated with some value $(upk, dslink)$. We prove the claim essentially via induction: We first show that if $(upk, dslink)$ was computed when adding a second transaction to *dsid* to $\mathcal{DB}$, then it is correctly associated. We then show that if *dsid* has been correctly associated with $(upk, dslink)$, then copying the value to some $dsid^*$ in the "when t_i is marked invalid" trigger correctly associated $(upk, dslink)$ to $dsid^*$. In each case, it suffices to show that $dslink = usk$ for *some* $(usk, dsid, dsrnd, \cdot)$ in G, as Lemma 45 then implies that this holds for all attribute vector nodes with *dsid*.

To show the first statement, let t_i be a transaction node in $\mathcal{DB}$ with predecessor *dsid* and assume t_j with the same predecessor is added to $\mathcal{DB}$ by DBsync. Let $dstag_i, dstag_j$ be their *dstag*s. Let i, j be the **Deduct** nodes in G corresponding to t_i and t_j, respectively[3]. Let $\vec{A}_i = (usk_i, dsid_i, dsrnd_i, v_i)$ and $\vec{A}_j = (usk_j, dsid_j, dsrnd_j, v_j)$ be the predecessors of i and j in G, respectively. It holds that $dsid_i = dsid_j = dsid$ by consistency of $\mathcal{L}$ (since equality with *dsid* is checked by the update function). Because $dsid_i = dsid_j$, we get $usk_i = usk_j$ and $dsrnd_i = dsrnd_j$ (Lemma 45). Because of consistency of $\mathcal{L}$, necessarily $dstag_i = (c_i = usk_i \cdot \gamma_i + dsrnd_i, \gamma_i, ctrace)$ and $dstag_j = (c_j = usk_j \cdot \gamma_j + dsrnd_j, \gamma_j, ctrace)$ (as enforced by the update function). Since $usk_i = usk_j$ and $dsrnd_i = dsrnd_j$ and $\gamma_i \neq \gamma_j$ (as implied by ¬error), we get $dslink = (c_i - c_j)/(\gamma_i - \gamma_j) = usk_i$. Hence $dslink = usk_i$ for our attribute vector $(usk_i, dsid_i, dsrnd_i, v_i)$ in G.

To show the second statement, let t_i be a transaction that is marked invalid. Let *dsid* be its predecessor (which is by assumption correctly associated with $(upk, dslink)$). Let $dstag = (c, \gamma, ctrace)$ be the associated *dstag* for t_i. Let $dsid^* = \mathsf{Trace}(ispp, dslink, dstag) = \mathsf{Decrypt}_{\text{enc}}(pp, dslink, ctrace)$ be t_i's successor. We show that $dsid^*$ is correctly associated with $(upk^*, dslink^*)$. Let $(usk, dsid, dsrnd, \cdot)$ be the predecessor of i in G. By assumption it *dsid* is correctly associated, hence $usk = dslink$. Let $(usk', dsid', dsrnd', \cdot)$ be the successor of i in G. By consistency of $\mathcal{L}$, $\mathsf{Decrypt}_{\text{enc}}(pp, usk = dslink, ctrace) = dsid'$ as guaranteed by the update function ψ. Hence $dsid' = dsid^*$. Because $usk = usk' = dslink$, we have that $(usk, dsid^*, dsrnd', \cdot)$ in G contains $dsid^*$ and $dslink = usk$, implying that $dsid^*$ is correctly associated with $(upk, dslink)$. □

Lemma 47. On every path P in G starting at some **Issue** node, it holds that $v_{\text{spent}}(P) \leq v_{\text{earned}}(P)$.

Proof. Let $(usk, dsid, dsrnd, v)$ be the successor (in G) of the last **Deduct** node on P. By design of our update functions, it is easy to see that $v \leq \sum_{i \in P \text{ is } \textbf{Credit} \text{ node}} k_i - \sum_{j \in P \text{ is } \textbf{Deduct} \text{ node}} k_j = v_{\text{earned}}(P) - v_{\text{spent}}(P)$. (the inequality is usually an equality, assuming that there is no **Credit** operation where adding k to the current v exceeds $v_{max} = p - 1$. If the latter happens, the integers will wrap around and result in the smaller $v' = v + k \mod p$) Also by design of the update functions, it holds that $v \geq 0$. Hence $v_{\text{earned}}(P) - v_{\text{spent}}(P) \geq 0$. □

F.4 Incentive System Framing Resistance

Theorem 16. Let $\mathcal{A}$ be a ppt adversary against framing resistance of our incentive system. Without loss of generality, we assume that $\mathcal{A}$ always outputs some actual user's handle u in the challenge

[3]This is a slight abuse of notation as the index i of t_i does not necessarily correspond to node i in G, which is associated with the ith oracle query.

phase. Let k be a (polynomial in λ) upper bound for the number of **Keygen** calls that $\mathcal{A}$ may make. We construct $\mathcal{B}$ against CPA-security of Π_{enc}.

- $\mathcal{B}$ gets pp, pk^* from its challenger. It finishes the incentive system setup by simulating the UACS setup $cpp \leftarrow \mathfrak{S}_{\mathsf{Setup}}(pp)$ and computing pk_{cmt} as usual. It hands $ispp = (pp, cpp, pk_{\text{cmt}})$ to $\mathcal{A}$.
- $\mathcal{B}$ randomly chooses an index $j \leq k$. For the jth **Keygen** query, $\mathcal{B}$ responds with $upk = pk^*$ and some handle u^*.
- Any queries involving u^* are run honestly by $\mathcal{B}$ except that it uses the UACS simulators to simulate the $\mathsf{Receive}$ and Update protocols without usk.
- Eventually, $\mathcal{A}$ enters the challenge phase and outputs some $dslink$ and a user handle u.
- If $u \neq u^*$, $\mathcal{B}$ aborts.
- If $upk^* \neq \mathsf{ComputePK}_{\text{enc}}(pp, dslink)$, $\mathcal{B}$ aborts.
- Otherwise, $\mathcal{B}$ uses $dslink$ as the secret key to pk^* to break its CPA challenge with probability 1.

The view of $\mathcal{A}$ in the framing resistance game is simulated perfectly and independently of j. We have that $\Pr[B \text{ wins the CPA game}] = \Pr[\text{Exp}^{\text{fram-res}}(\Pi, \mathcal{A}, \lambda) = 1] \cdot \Pr[u = u^*]$. By assumption, Π_{enc} is CPA-secure. It holds that $\Pr[u = u^*]$ is non-negligible, hence $\Pr[\text{Exp}^{\text{fram-res}}(\Pi, \mathcal{A}, \lambda) = 1]$ must be negligible. $\square$