Practical, Anonymous, and Publicly Linkable Universally-Composable Reputation Systems

Johannes Blömer, Fabian Eidens, and Jakob Juhnke(✉)

Department of Computer Science, Paderborn University, Paderborn, Germany
{bloemer,feidens,juhnke}@mail.uni-paderborn.de

Abstract. We consider reputation systems in the Universal Composability Framework where users can anonymously rate each others products that they purchased previously. To obtain trustworthy, reliable, and honest ratings, users are allowed to rate products only once. Everybody is able to detect users that rate products multiple times. In this paper we present an ideal functionality for such reputation systems and give an efficient realization that is usable in practical applications.

Keywords: Reputation · Trust · Anonymity
Universal Composability

1 Introduction

Reputation systems provide valuable information about previous transactions and are popular tools to measure trustworthiness of interacting parties. This measurement relies on the existence of a large number of ratings for one specific subject. But in most practical applications the process of rating reveals, besides the actual rating, many information about the rater. Providers of reputation systems use this information in many different ways, e.g. for profiling users, which are not necessarily desired by the users. Moreover, users can feel compelled to rate "dishonestly/benevolent" when they fear negative consequences from negative ratings. Therefore, it is important that the process of rating does not reveal more information than the actual rating. Besides that, reputation systems need to be protected against various attacks to provide trustworthy, reliable and honest ratings. These attacks include self-rating attacks (also known as self-promoting attacks), Sybil attacks, whitewashing attacks, bad mouthing attacks, ballot stuffing attacks, and value imbalance attacks. Both the privacy concerns and the prevention of attacks are discussed frequently in the literature, e.g. [1,8,13,17,20,21,23,24,26,27], albeit they are not considered simultaneously.

J. Blömer, F. Eidens, and J. Juhnke—This author was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre On-The-Fly Computing (SFB 901).

J. Blömer and J. Juhnke—This author was partially supported by the Ministry of Education and Research, grant 16SV7055, project "KogniHome".

N. P. Smart (Ed.): CT-RSA 2018, LNCS 10808, pp. 470–490, 2018.
https://doi.org/10.1007/978-3-319-76953-0_25

Further important security properties for reputation systems are *anonymity*, *(public) linkability*, *traceability*, and *non-frameability*, as discussed in [1,6,13,27]. Anonymity means that ratings of honest users are indistinguishable, whereas public linkability requires that anyone can decide whether or not two ratings for the same product were created by the same user. Also, ratings need to be traceable: the identity of any rater can be determined by a designated System Manager. In the course of this non-frameability guarantees that honest parties are not blamed having rated some product, when they did not. The combination of traceability and non-frameability enables penalizing dishonest behavior.

All previously mentioned works consider reputation systems in isolation, although reputation systems are always used in combination with other applications. In such situations stand-alone security definitions, as in [6], do not guarantee security. With the Universal Composability Framework (UC) [9] there exists a methodology that guarantees security even in composed applications. Informally, in UC the execution of a real-life protocol is compared to the execution of an ideal protocol. If the real-life and ideal protocol executions are indistinguishable, then the real-life protocol is *UC-secure*. Based on this security definition Canetti [9] formulates a composition theorem which states that any UC-secure protocol is also secure when it is composed with other protocols.

Our Contribution. We present an ideal functionality for reputation systems $\mathcal{F}_{\mathsf{RS}}$ in the Universal Composability Framework [9]. Our ideal functionality prevents all previously mentioned attacks and provides anonymity, public linkability, traceability, and non-frameability. In contrast to [6], users can rate each others products; there is no separation of customers and providers.

Besides defining an ideal functionality we present an efficient protocol for reputation systems that realizes $\mathcal{F}_{\mathsf{RS}}$. This protocol is influenced by different techniques known from Σ-protocols [16] and (dynamic) group signatures [2–4,7], similarly to the scheme in [6]. But our protocol is more efficient and more flexible than the scheme in [6] and it is secure even under concurrent composition (UC-secure).

2 The Ideal Functionality for Reputation Systems

In the first part of this section, we give some intuition to our ideal functionality of a reputation system $\mathcal{F}_{\mathsf{RS}}$. The second part concerns the formal definition of $\mathcal{F}_{\mathsf{RS}}$ in the Universal Composability Framework [9]. We discuss the functionality and its security properties in the third part of the section.

Intuition to Our Reputation System. A meaningful reputation system must provide trustworthy, reliable, and honest ratings. Furthermore, it should be flexible in the sense that it can be combined with many different applications. Therefore, we focus on the process of secure rating and provide a scheme that can be combined with any high-level application. For this reason, the aggregation of ratings and the evaluation of a specific reputation function is excluded from our model. Specifically, we handle the actual rating-message as a placeholder for the higher level application.

We consider reputation systems where users within the system can rate each others products. The term *product* refers to anything that can be used as a basis for ratings. Each user in our system has to *register* once at a *System Manager*, before a product can be rated. This prevents Sybil attacks, whitewashing attacks, bad mouthing attacks, and ballot stuffing attacks, and gives the System Manager the ability to punish misbehaving users. For this to work the system must prevent users to register with different identities. When users do not want to rate other products, a registration is not necessary - publishing products and verifying ratings is independent of the registration, which increases trust in the system. Analogously to registering, a product must be *purchased* prior to rating. This requirement assures that ratings are only given by raters using the product. Also, this is a protection mechanism against value imbalance attacks.

To further increase trust in the reputation system, raters must be able to rate purchased products anonymously. Without anonymity raters may tend to rate dishonestly when they fear negative consequences from the product owner. At the same time a product owner must be protected against unjustified negative ratings. This is achieved by giving the System Manager the ability to revoke the anonymity of a rater. Of course, the System Manager must not be able to accuse an honest user having misbehaved.

The negative side-effects of anonymity are that self-ratings, i.e. ratings for a product from the product owner, are hard to prevent and that a single rater who purchased a product could rate this product multiple times. Therefore we require a reputation system to explicitly forbid self-ratings and to provide *linkable ratings*: everybody - even outsiders of the system - must be able to detect multiple ratings from the same user for the same product.

As pointed out above, the security requirements a reputation system has to fulfill include - but are not limited to - *anonymity* for raters, *unforgeability* and *public linkability* of ratings, and the ability to determine the raters' identity. These properties have already been studied in the simpler context of group signatures [2–4,7,18]. However, reputation systems have more security requirements than group signatures, as they do not consist of a single group of users. Instead, reputation systems can be seen as a collection of multiple group signature schemes - one for each product. Moreover, a single user may offer several products. Hence, in the definition of security properties the different group signature schemes must be considered in conjunction. Therefore, we adapt and extend these notions and give our formal definition of a secure reputation system in the Universal Composability Framework [9]. This framework guarantees security even for concurrently composed protocols. Stand-alone security definitions do not provide this strong guarantees, which are very important for our reputation system, as we intend it to be combined with other applications.

Additionally to the experiment-based security definitions for reputation systems [6] and group signatures [3,4], our ideal functionality $\mathcal{F}_{\mathsf{RS}}$ is influenced by the ideal functionalities for digital signatures $\mathcal{F}_{\mathsf{SIG}}$ [10], public-key encryption $\mathcal{F}_{\mathsf{PKE}}$ [9] and group signatures [2].

The Universal Composability Framework. In contrast to stand-alone security definitions (both experiment-based and simulation-based), the Universal Composability Framework, introduced by Canetti [9], provides security under concurrent composition of different applications. To achieve this strong security notion, the execution of a real-life protocol is compared to the execution of an ideal protocol. Both protocol executions are controlled by an environment $\mathcal{Z}$ that tries to distinguish whether it interacts with the real-life protocol or the ideal protocol.

The ideal protocol is described by an ideal functionality $\mathcal{F}$ that handles every (cryptographic) task as a trusted party and interacts with an ideal adversary $\mathcal{S}$ (also called a simulator) and all parties involved in the protocol. Every party hands its inputs from the environment securely to $\mathcal{F}$. Then $\mathcal{F}$ computes the parties' output and sends it back to the party. Whenever a party receives a message from $\mathcal{F}$, the party outputs this message directly to the environment. The ideal adversary $\mathcal{S}$ may corrupt some parties and can block the delivery of messages from $\mathcal{F}$ to a party. The inputs a party hands to $\mathcal{F}$ cannot be seen by $\mathcal{S}$. In the real-life execution all parties compute their outputs by running the defined protocol. Analogously to $\mathcal{S}$, a real-life adversary $\mathcal{A}$ may corrupt parties within the real-life protocol execution.

We say that the real-life protocol UC-realizes the ideal protocol, if no environment can distinguish an interaction with the real-life protocol and $\mathcal{A}$ from an interaction with the ideal protocol and $\mathcal{S}$. Based on this security definition Canetti [9] formulates a composition theorem which states that any UC-secure protocol is also secure when it is executed concurrently with other protocols.

For our proof of security we will consider black-box simulators $\mathcal{S}$, denoted by $\mathcal{S}^{\mathcal{A}}$, that have block-box access to real-life adversaries $\mathcal{A}$. Also we consider a model with ideally authenticated channels, meaning that an adversary is able to read the messages sent, but is unable to modify them. We refer to this communication model as the *authenticated channels assumption*.

2.1 The Formal Definition of $\mathcal{F}_{\mathsf{RS}}$

Our ideal functionality interacts with the parties $P_{\mathrm{IDM}}, P_1, P_2, \ldots, P_n$ and an ideal adversary $\mathcal{S}$, which is also called a *simulator*. The party P_{IDM} acts as the System Manager, whereas the parties P_i correspond to the users within the reputation system. Furthermore, $\mathcal{F}_{\mathsf{RS}}$ manages the lists $\mathfrak{Params}$, $\mathfrak{Reg}$, $\mathfrak{Prods}$, $\mathfrak{Purch}$, $\mathfrak{Ratings}$, and $\mathfrak{Open}$ to store important information. Before giving the formal definition of $\mathcal{F}_{\mathsf{RS}}$, we explain how these lists are used. We also introduce the notation needed in the definition of $\mathcal{F}_{\mathsf{RS}}$.

$\mathfrak{Params}$: This list stores all pairs of the form (P_{IDM}, pp) containing *public parameters* the simulator $\mathcal{S}$ gives to $\mathcal{F}_{\mathsf{RS}}$ during KeyGen-requests. The first component of a pair is fixed to P_{IDM}, whereas the second component represents the actual parameters given by $\mathcal{S}$.

$\mathfrak{Reg}$: The list $\mathfrak{Reg}$ stores pairs of the form (pp, P_i) containing *registration information*. The first component stores the public parameters the registrated party used in the Register-protocol, whereas the second component is the registrated party.

$\mathfrak{Prods}$: All *products* that are used within the reputation system are stored as 4-tuples $(P_i, prod, ppk, b)$ in the list $\mathfrak{Prods}$. The first component of a tuple declares the product owner, the second is a product identifier (a bitstring chosen by the environment), the third specifies the corresponding *product-public key* and the fourth component is a validity bit. There can exist different products with the same product identifier, but for different product owners. The validity bit indicates whether the product-public key matches the given product owner and the product identifier.

$\mathfrak{Purch}$: When some party successfully purchased a product, this information is stored as 4-tuple $(P_i, P_j, prod, ppk)$ in the list $\mathfrak{Purch}$. For every tuple in the list the first component represents the purchaser, whereas the other components determine the product that was purchased (the product owner, the product identifier and the product-public key).

$\mathfrak{Ratings}$: The list $\mathfrak{Ratings}$ stores the most complex information as 10-tuples of the form $(pp, P_i, P_j, prod, ppk, m, \sigma, b, lid, oid)$. The components of each tuple represent the following information:

1. pp - the public parameters a rating is generated for,
2. P_i - the identity of the rater ((pp, P_i) should match an entry in $\mathfrak{Reg}$),
3. P_j - the product owner of the product the rating is generated for,
4. $prod$ - the product identifier of the product the rating is generated for,
5. ppk - the product-public key of the product the rating is generated for (the tuple $(P_i, P_j, prod, ppk)$ should match an entry in $\mathfrak{Purch}$),
6. m - rating message (a placeholder for high-level applications),
7. σ - the rating,
8. b - the validity bit (indicating whether the rating is *valid*),
9. lid - the *linking-class identifier*, which is managed by the algorithm RebLDB, and
10. oid - the *opening-proof identifier*.

The linking-class identifier is needed to model the linkability property: two ratings with the same linking-class identifier have the same author. The opening-class identifier binds a list of opening-proofs to a specific rating. Whenever a new rating is added to the list $\mathfrak{Ratings}$, $\mathcal{F}_{\mathsf{RS}}$ uses the current value of a global counter $lidc$ as the linking-class identifier and increments the counter. The subsequent execution of RebLDB ensures that the rating is put into the correct linking-class, according to the linkability-relation. A more detailed explanation of this behavior and the oid-mechanism is given in the discussion of the security properties of $\mathcal{F}_{\mathsf{RS}}$.

$\mathfrak{Open}$: This list stores all *opening-proofs* as 4-tuples of the form (oid, τ, b, P). The first component is an opening-proof identifier that binds a tuple to a specific rating with the same identifier. The second component is the actual opening-proof. The third component is a validity bit indicating whether the proof is *valid* and the fourth component is the claimed party that shall be the author of the associated rating. The value $oid = \perp$ within a rating expresses that the rating was not opened yet and hence no opening-proof exists. To uniquely bind opening-proofs to ratings a global counter $oidc$ is used and incremented whenever a new opening-proof is bound to an unopened rating.

To manipulate the described lists, we introduce two operations:

- adding a tuple v to a list L is expressed by $L.\mathsf{Add}(v)$, and
- substituting a tuple v_{old} with a tuple v_{new} is expressed by $L.\mathsf{Sub}(v_{\text{old}}, v_{\text{new}})$.

Substituting a tuple v_{old} means that this tuple is removed from the list, while the tuple v_{new} is added to the list.

The classical notation to address components of tuples is using indices, i.e. $v = (v_1, v_2, \ldots, v_n)$, where v_i is the i'th component of tuple v. We deviate from this notation to prevent confusion with different variables and address the i'th component of a tuple v by $v[i]$.

Remark 1 (Technical Details of $\mathcal{F}_{\mathsf{RS}}$). Whenever $\mathcal{F}_{\mathsf{RS}}$ misses some information, the symbol $\perp$ is used to highlight this fact. Also the Simulator $\mathcal{S}$ can output this symbol at some points to indicate that it is not able to respond to a request. Depending on the situation, this is not necessarily a failure.

To reduce repeating code we introduce the *internal* activations VfyProd, VfyRtg, LinkRtgs, and RebLDB. These activations are only used by $\mathcal{F}_{\mathsf{RS}}$ as an internal subroutine and are not callable by parties or adversaries.

The activations for user registration (Register) and purchasing a product (Purchase) generate outputs to multiple parties. Albeit this mechanism is rarely used in the UC framework another example for this technique can be found in the definition of homomorphic UC commitments $\mathcal{F}_{\text{HCOM}}$ by Damgård et al. [15].

With these prerequisites we now give the formal definition of $\mathcal{F}_{\mathsf{RS}}$.

$\mathcal{F}_{\mathsf{RS}}$

$\mathcal{F}_{\mathsf{RS}}$ interacts with parties $P_{\text{IDM}}, P_1, \ldots, P_n$, and the ideal adversary (simulator) $\mathcal{S}$. Further it manages the lists $\mathfrak{Params}, \mathfrak{Reg}, \mathfrak{Prods}, \mathfrak{Purch}, \mathfrak{Ratings}$, and $\mathfrak{Open}$ which are initially empty, and the counters $lidc, oidc$, which are initialized with 0. All outputs from $\mathcal{F}_{\mathsf{RS}}$ to some party P are public delayed outputs.

Registry Key Generation: On input (KeyGen, sid) from P_{IDM}

1: Check that $sid = (P_{\text{IDM}}, sid')$ for some sid'. If not, ignore the request.
2: Send (KeyGen, sid) to $\mathcal{S}$ and receive $(\mathsf{KeyGen}, sid, pp)$ from $\mathcal{S}$.
3: Set $\mathfrak{Params}.\mathsf{Add}(P_{\text{IDM}}, pp)$ and send $(\mathsf{KeyGen}, sid, pp)$ to P_{IDM}.

User Registration: On input $(\mathsf{Register}, sid, pp')$ from P_i

1: Check that $sid = (P_{\text{IDM}}, sid')$ for some sid'. If not, ignore the request.
2: Send $(\mathsf{Register}, sid, pp', P_i)$ to $\mathcal{S}$ and receive $(\mathsf{Register}, sid, pp', P_i, b)$ from $\mathcal{S}$.
3: **If** P_{IDM} and P_i are honest $\wedge\ (P_{\text{IDM}}, pp') \in \mathfrak{Params} \wedge (P_i, pp') \notin \mathfrak{Reg}$ **then** $f := 1$.
4: **Else If** P_{IDM} is honest $\wedge\ (P_{\text{IDM}}, pp') \notin \mathfrak{Params}$ **then** $f := 0$.
5: **Else** $f := b$.

6: **If** $f = 1$ **then** $\mathfrak{Reg}.\mathsf{Add}(pp', P_i)$.
7: Send $(\mathsf{Register}, sid, pp', P_i, f)$ to P_i and P_{IDM}.

Product Addition: On input $(\mathsf{NewProduct}, sid, prod)$ from P_i
1: Check that $sid = (P_{\mathrm{IDM}}, sid')$ for some sid'. If not, ignore the request.
2: Send $(\mathsf{NewProduct}, sid, P_i, prod)$ to $\mathcal{S}$ and receive $(\mathsf{NewProduct}, sid, P_i, prod, ppk)$ from $\mathcal{S}$.
3: **If** $(P', prod', ppk, 1) \in \mathfrak{Prods}$, where $(P', prod') \neq (P_i, prod)$ **then**
4: output `error` and halt.
5: **Else** $\mathfrak{Prods}.\mathsf{Add}(P_i, prod, ppk, 1)$ and send $(\mathsf{NewProduct}, sid, prod, ppk)$ to P_i.

Purchase: On input $(\mathsf{Purchase}, sid, P_j, prod, ppk)$ from P_i
1: Check that $sid = (P_{\mathrm{IDM}}, sid')$ for some sid'. If not, ignore the request.
2: **If** $P_i = P_j \vee \mathsf{VfyProd}(sid, P_j, prod, ppk) = 0$ **then** ignore the request.
3: Send $(\mathsf{Purchase}, sid, P_i, P_j, prod, ppk)$ to $\mathcal{S}$ and receive $(\mathsf{Purchase}, sid, P_i, P_j, prod, ppk, b)$ from $\mathcal{S}$.
4: **If** P_i and P_j are honest **then** $f \coloneqq 1$.
5: **Else** $f \coloneqq b$.
6: **If** $f = 1$ **then** $\mathfrak{Purch}.\mathsf{Add}(P_i, P_j, prod, ppk)$.
7: Send $(\mathsf{Purchase}, sid, P_i, P_j, prod, ppk, f)$ to P_i and P_j.

VfyProd**:** On internal input $(sid, P_j, prod, ppk)$
1: Send $(\mathsf{VfyProd}, sid, P_j, prod, ppk)$ to $\mathcal{S}$ and receive $(\mathsf{VfyProd}, sid, P_j, prod, ppk, b)$ from $\mathcal{S}$.
2: **If** $(P_j, prod, ppk, f') \in \mathfrak{Prods}$ **then** $f \coloneqq f'$.
3: **Else If** $b = 1 \wedge P_j$ is honest **then** output `error` and halt.
4: **Else If** $(P', prod', ppk, 1) \in \mathfrak{Prods}$, where $(P', prod') \neq (P_i, prod)$ **then**
5: $\mathfrak{Prods}.\mathsf{Add}(P_i, prod, ppk, 0)$ and $f \coloneqq 0$.
6: **Else** Set $\mathfrak{Prods}.\mathsf{Add}(P_i, prod, ppk, b)$ and $f \coloneqq b$.
7: Return f.

Rate a Product: On input $(\mathsf{Rate}, sid, pp, P_j, prod, ppk, m)$ from P_i
1: Check that $sid = (P_{\mathrm{IDM}}, sid')$ for some sid'. If not, ignore the request.
2: **If** $(pp, P_i) \notin \mathfrak{Reg} \vee (P_i, P_j, prod, ppk) \notin \mathfrak{Purch} \vee (pp, P_i, P_j, prod, ppk, m', \sigma', 1, lid, oid) \in \mathfrak{Ratings}$ for some m', σ', lid **then** ignore the request.
3: **If** P_{IDM} is honest **then** Send $(\mathsf{Rate}, sid, pp, P_j, prod, ppk, m)$ to $\mathcal{S}$ and receive $(\mathsf{Rate}, sid, pp, P_j, prod, ppk, m, \sigma)$ from $\mathcal{S}$.
4: **Else** Send $(\mathsf{Rate}, sid, pp, P_i, P_j, prod, ppk, m)$ to $\mathcal{S}$ and receive $(\mathsf{Rate}, sid, pp, P_i, P_j, prod, ppk, m, \sigma)$ from $\mathcal{S}$.
5: **If** $(pp, P', P_j, prod, ppk, m, \sigma, 0, lid, oid) \in \mathfrak{Ratings}$ for some P', lid, oid **then**
6: Output `error` and halt.
7: Set $r \coloneqq (pp, P_i, P_j, prod, ppk, m, \sigma, 1, lidc, \perp)$ and $lidc \coloneqq lidc + 1$.
8: Set $\mathfrak{Ratings}.\mathsf{Add}(r)$ and run $\mathsf{RebLDB}(sid, r, \perp)$.
9: Send $(\mathsf{Rate}, sid, pp, P_j, prod, ppk, m, \sigma)$ to P_i.

Verifying a Rating: On input $(\mathsf{Verify}, sid, pp, P_j, prod, ppk, m, \sigma)$ from P_i
1: Check that $sid = (P_{\mathrm{IDM}}, sid')$ for some sid'. If not, ignore the request.

2: Set $(X, f, oid) \coloneqq \mathsf{VfyRtg}(sid, pp, P_j, prod, ppk, m, \sigma)$.
3: Send $(\mathsf{Verify}, sid, pp, P_j, prod, ppk, m, \sigma, f)$ to P_i.

VfyRtg: On internal input $(sid, pp, P_j, prod, ppk, m, \sigma)$
1: **If** $\mathsf{VfyProd}(sid, P_j, prod, ppk) = 0$ **then** ignore the request.
2: Send $(\mathsf{Verify}, sid, pp, P_j, prod, ppk, m, \sigma)$ to $\mathcal{S}$ and receive $(\mathsf{Verify}, sid, pp, P_j, prod, ppk, m, \sigma, b, P)$ from $\mathcal{S}$.
3: **If** $(pp, X', P_j, prod, ppk, m, \sigma, f', lid', oid') \in \mathfrak{Ratings}$ for some X', f', lid' and oid' **then** $X \coloneqq X', f \coloneqq f', oid \coloneqq oid'$.
4: **Else If** $b = 0 \vee P = P_j$ **then**
5: Set $\mathfrak{Ratings}.\mathsf{Add}(pp, \bot, P_j, prod, ppk, m, \sigma, 0, \bot, \bot)$, $X \coloneqq \bot$, $f \coloneqq 0$, and $oid \coloneqq \bot$.
6: **Else If** P_j is honest, $P \neq \bot$ and $(P, P_j, prod, ppk) \notin \mathfrak{Purch}$ **then**
7: Output `error` and halt.
8: **Else If** $P \neq \bot$ and P is honest **then** output `error` and halt.
9: **Else If** $P = \bot$ and P_{IDM} is honest **then** output `error` and halt.
10: **Else** Set $r \coloneqq (pp, P, P_j, prod, ppk, m, \sigma, 1, lidc, \bot)$, $X \coloneqq P$, $f \coloneqq 1$, and $oid \coloneqq \bot$.
11: Set $lidc \coloneqq lidc + 1$, $\mathfrak{Ratings}.\mathsf{Add}(r)$, and run $\mathsf{RebLDB}(sid, r, \bot)$.
12: Return (X, f, oid).

Linking Ratings: On input $(\mathsf{Link}, sid, pp, P_j, prod, ppk, m_0, \sigma_0, m_1, \sigma_1)$ from P_i
1: Check that $sid = (P_{\mathrm{IDM}}, sid')$ for some sid'. If not, ignore the request.
2: Set $b \coloneqq \mathsf{LinkRtgs}(sid, pp, P_j, prod, ppk, m_0, \sigma_0, m_1, \sigma_1)$.
3: Send $(\mathsf{Link}, sid, pp, P_j, prod, ppk, m_0, \sigma_0, m_1, \sigma_1, b)$ to P_i.

LinkRtgs: On internal input $(sid, pp, P_j, prod, ppk, m_0, \sigma_0, m_1, \sigma_1)$
1: Set $(X_k, f_k, oid_k) \coloneqq \mathsf{VfyRtg}(sid, pp, P_j, prod, ppk, m_k, \sigma_k)$ for $k \in \{0, 1\}$.
2: Send $(\mathsf{Link}, sid, pp, P_j, prod, ppk, m_0, \sigma_0, m_1, \sigma_1)$ to $\mathcal{S}$, receive $(\mathsf{Link}, sid, pp, P_j, prod, ppk, m_0, \sigma_0, m_1, \sigma_1, b)$ from $\mathcal{S}$, and set $f \coloneqq 0$.
3: **If** $f_0 = f_1 = 1$ **then**
4: Let $r_k \in \mathfrak{Ratings}$ be the unique tuples $r_k \coloneqq (pp, X_k, P_j, prod, ppk, m_k, \sigma_k, 1, lid_k, oid_k)$, for $k \in \{0, 1\}$.
5: **If** $lid_0 = lid_1$ **then** $f \coloneqq 1$.
6: **Else If** $X_0 = X_1 \wedge X_0 = \bot \wedge X_1 = \bot$ **then** $f \coloneqq b$.
7: **Else If** $X_0 \neq X_1 \wedge X_0 \neq \bot \wedge X_1 \neq \bot$ **then** $f \coloneqq 0$.
8: **Else If** $(X_k = \bot \wedge X_{1-k} \neq \bot \wedge X_{1-k}$ is honest) for $k = 0 \vee k = 1$ **then** $f \coloneqq 0$.
9: **Else If** $(X_k = \bot \wedge X_{1-k} \neq \bot \wedge X_{1-k}$ is corrupted) for $k = 0 \vee k = 1$ **then** $f \coloneqq b$.
10: **If** $f = 1$ **then** run $\mathsf{RebLDB}(sid, r_0, r_1)$.
11: Return f.

RebLDB: On internal input (sid, r, s)
1: Parse r as $(pp, X_0, P_j, prod, ppk, m_0, \sigma_0, 1, lid_0, oid_0)$.
2: **If** $s = \bot \wedge X_1 \neq \bot$ **then**
3: Set $\mathcal{L} \coloneqq \{\ell | \ell \in \mathfrak{Ratings} \wedge \ell[1] = pp \wedge \ell[2] = X_0 \wedge \ell[3] = P_j \wedge \ell[4] = prod \wedge \ell[5] = ppk \wedge \ell[8] = 1\}$ and $lid \coloneqq \min\{\ell[9] | \ell \in \mathcal{L}\}$.

4: **For each** $\ell \in \mathcal{L}$ **do** Set $\ell' := \ell$, $\ell'[9] := lid$, and $\mathfrak{Ratings}.\mathsf{Sub}(\ell, \ell')$.
5: **If** $s \neq \perp$ **then**
6: Parse s as $(pp, X_1, P_j, prod, ppk, m_1, \sigma_1, 1, lid_1, oid_1)$
7: **If** $X_0 = \perp \wedge X_1 \neq \perp$ **then** Set $X := X_1$.
8: **Else** Set $X := X_0$.
9: Set $\mathcal{L} := \{\ell | \ell \in \mathfrak{Ratings} \wedge \ell[1] = pp \wedge \ell[3] = P_j \wedge \ell[4] = prod \wedge \ell[5] = ppk \wedge \ell[8] = 1 \ \wedge (\ell[9] = lid_0 \vee \ell[9] = lid_1)\}$ and $lid := \min\{lid_0, lid_1\}$.
10: **For each** $\ell \in \mathcal{L}$ **do** Set $\ell' := \ell$, $\ell'[2] := X$, $\ell'[9] := lid$, and $\mathfrak{Ratings}.\mathsf{Sub}(\ell, \ell')$.
11: Set $\mathcal{P} := \{p | p \in \mathfrak{Purch} \wedge p[2] = P_j \wedge p[3] = prod \wedge p[4] = ppk\}$.
12: Set $\mathcal{L} := \{\ell | \ell \in \mathfrak{Ratings} \wedge \ell[1] = pp \wedge \ell[3] = P_j \wedge \ell[4] = prod \wedge \ell[5] = ppk \wedge \ell[8] = 1\}$.
13: **If** P_{IDM} is corrupted, P_j is honest and $|\mathcal{P}| < |\{\ell[9] | \ell \in \mathcal{L}\}|$ **then**
14: **For each** $(\ell, \ell') \in \mathcal{L}^2$ **do**
15: Run $\mathsf{LinkRtgs}(sid, \ell[1], \ell[3], \ell[4], \ell[5], \ell[6], \ell[7], \ell'[6], \ell'[7])$ ignoring the output of $\mathsf{LinkRtgs}$.
16: Set $\mathcal{P} := \{p | p \in \mathfrak{Purch} \wedge p[2] = P_j \wedge p[3] = prod \wedge p[4] = ppk\}$.
17: Set $\mathcal{L} := \{\ell | \ell \in \mathfrak{Ratings} \wedge \ell[1] = pp \wedge \ell[3] = P_j \wedge \ell[4] = prod \wedge \ell[5] = ppk \wedge \ell[8] = 1\}$.
18: **If** P_j is honest and $|\mathcal{P}| < |\{\ell[9] | \ell \in \mathcal{L}\}|$ **then** Output `error` and halt.

Determine Raters Identity: On input $(\mathsf{Open}, sid, pp, P_j, prod, ppk, m, \sigma)$ from P_{IDM}
1: Check that $sid = (P_{\text{IDM}}, sid')$ for some sid'. If not, ignore the request.
2: **If** $(P_{\text{IDM}}, pp) \notin \mathfrak{Params}$ **then** ignore the request.
3: Set $(X, f, oid) := \mathsf{VfyRtg}(sid, pp, P_j, prod, ppk, m, \sigma)$.
4: **If** $f = 1$ **then**
5: Let $r \in \mathfrak{Ratings}$ be the unique tuple $r := (pp, X, P_j, prod, ppk, m, \sigma, 1, lid', oid)$ for some lid'.
6: **If** $oid = \perp$ **then**
7: Set $r' := r$, $r'[10] := oidc$, $\mathfrak{Ratings}.\mathsf{Sub}(r, r')$ and $oidc := oidc + 1$.
8: Send $(\mathsf{Open}, sid, pp, P_j, prod, ppk, m, \sigma, X)$ to P_{IDM}.
9: **Else** Send $(\mathsf{Open}, sid, pp, P_j, prod, ppk, m, \sigma, \perp)$ to P_{IDM}.

Generate Opening Proofs: On input $(\mathsf{OProof}, sid, pp, P_j, prod, ppk, m, \sigma, P)$ from P_{IDM}
1: Check that $sid = (P_{\text{IDM}}, sid')$ for some sid'. If not, ignore the request.
2: **If** $(P_{\text{IDM}}, pp) \notin \mathfrak{Params}$ **then** ignore the request.
3: Set $(X, f, oid) := \mathsf{VfyRtg}(sid, pp, P_j, prod, ppk, m, \sigma)$.
4: Send $(\mathsf{OProof}, sid, pp, P_j, prod, ppk, m, \sigma, P)$ to $\mathcal{S}$ and receive $(\mathsf{OProof}, sid, pp, P_j, prod, ppk, m, \sigma, P, \tau)$ from $\mathcal{S}$.
5: **If** $f \neq 1 \vee X \neq P \vee oid = \perp$ **then**
6: Send $(\mathsf{OProof}, sid, pp, P_j, prod, ppk, m, \sigma, P, \perp)$ to P_{IDM}.
7: **Else**
8: **If** $\tau = \perp \vee (oid, \tau, 0, P) \in \mathfrak{Open}$ **then** output `error` and halt.
9: $\mathfrak{Open}.\mathsf{Add}(oid, \tau, 1, P)$ and send $(\mathsf{OProof}, sid, pp, P_j, prod, ppk, m, \sigma, P, \tau)$ to P_{IDM}.

Verifying Opening-Proofs: On input $(\mathsf{Judge}, sid, pp, P_j, prod, ppk, m, \sigma, P, \tau)$ from P_i

1: Check that $sid = (P_{\text{IDM}}, sid')$ for some sid'. If not, ignore the request.
2: Set $(X, f, oid) \coloneqq \mathsf{VfyRtg}(sid, pp, P_j, prod, ppk, m, \sigma)$.
3: Send $(\mathsf{Judge}, sid, pp, P_j, prod, ppk, m, \sigma, P, \tau)$ to $\mathcal{S}$, receive $(\mathsf{Judge}, sid, pp, P_j, prod, ppk, m, \sigma, P, \tau, b)$ from $\mathcal{S}$, and set $v \coloneqq b$.
4: **If** $f = 0 \vee P = \perp \vee \tau = \perp$ **then**
5: Send $(\mathsf{Judge}, sid, pp, P_j, prod, ppk, m, \sigma, P, \tau, 0)$ to P_i.
6: **Else If** $X \neq \perp$ **then**
7: Let $r \in \mathfrak{Ratings}$ be the unique tuple $r \coloneqq (pp, X, P_j, prod, ppk, m, \sigma, 1, lid', oid)$ for some lid', and set $r' \coloneqq r$.
8: **If** $X = P \wedge (oid, \tau, 1, P) \in \mathfrak{Open}$ **then** $v \coloneqq 1$.
9: **Else If** $X \neq P \vee (oid, \tau, 0, P) \in \mathfrak{Open}$ **then** $v \coloneqq 0$.
10: **Else If** P_{IDM} and P are honest and $b = 1$ **then** output `error` and halt.
11: **Else**
12: Let $r \in \mathfrak{Ratings}$ be the unique tuple $r \coloneqq (pp, \perp, P_j, prod, ppk, m, \sigma, 1, lid', oid)$ for some lid', and set $r' \coloneqq r$.
13: **If** $(oid, \tau, 0, P) \in \mathfrak{Open}$ **then** $v \coloneqq 0$.
14: **Else If** $b = 1 \wedge P$ is honest **then** output `error` and halt.
15: **Else If** $b = 1$ **then**
16: Set $v \coloneqq 1$, $r'[2] \coloneqq P$, $\mathfrak{Ratings}.\mathsf{Sub}(r, r')$, $r \coloneqq r'$
17: Run $\mathsf{RebLDB}(sid, r', \perp)$.
18: **If** $oid = \perp$ **then**
19: $r'[10] \coloneqq oidc$, $\mathfrak{Ratings}.\mathsf{Sub}(r, r')$, $\mathfrak{Open}.\mathsf{Add}(oidc, \tau, v, P)$, $oidc \coloneqq oidc + 1$.
20: **Else** $\mathfrak{Open}.\mathsf{Add}(oid, \tau, v, P)$.
21: Send $(\mathsf{Judge}, sid, pp, P_j, prod, ppk, m, \sigma, P, \tau, v)$ to P_i.

Functionality 1: Reputation System

Security Properties of $\mathcal{F}_{\mathsf{RS}}$. As many other ideal functionalities in the UC framework, we define $\mathcal{F}_{\mathsf{RS}}$ to work as a "registy service" to store parameters, ratings, and opening-proofs. Using the right parameters, every party is able to check whether ratings and opening-proofs are stored by $\mathcal{F}_{\mathsf{RS}}$. In all activations, $\mathcal{F}_{\mathsf{RS}}$ lets the simulator $\mathcal{S}$ choose the values needed to respond to the activation. The requirements on these values are defined as restrictions for each activation. In the following, we discuss these restrictions and the implied security properties.

Registry Key Generation: Similar to the Signature Functionality $\mathcal{F}_{\mathsf{SIG}}$ [10] and the Public-Key Encryption Functionality $\mathcal{F}_{\mathsf{PKE}}$ [9], we do not make any security relevant requirements on the *public parameters pp*.

User Registration: Being registered is a prerequisite to rate a product and covers the first step to prevent Sybil attacks, whitewashing attacks, bad mouthing attacks, and ballot stuffing attacks. The user registration models an interactive protocol between P_{IDM} and some party P_i. In general, $\mathcal{F}_{\mathsf{RS}}$ lets the simulator $\mathcal{S}$ decide whether party P_i successfully registered, with the following two restrictions: non-registered honest parties communicating with an honest P_{IDM} using the right public parameters will always be registered after the protocol execution ($b = 1$) and an honest P_{IDM} will reject a party from registering, when wrong parameters are used ($b = 0$).

Product Addition and VfyProd: The NewProduct-activation is used by party P_i to publish a new *product-public key ppk* for a given product $prod \in \{0,1\}^*$. The value *ppk* is bound to the bitstring *prod* and to the party requesting it, such that every party can validate the ownership of a product. Formally this means, that a product-public key is only *valid* for one specific pair $(P, prod)$. This is a very important requirement, because it models *unforgeability* of product-public keys. Without this property any corrupted party P_j could "copy" some *ppk* (that was generated by an honest party P_i) and declare foreign ratings as own ratings: all valid ratings for $(P_i, prod, ppk)$ would also be valid for $(P_j, prod', ppk')$. Since we want to have a reliable, trustworthy and fair system such attacks must be prevented. We emphasize that VfyProd is modeled as an internal subroutine within $\mathcal{F}_{\mathsf{RS}}$ and is implicitly used in other activations.

Purchase: Another prerequisite to rate a product is to purchase it. This is necessary to prevent value imbalance attacks. The purchasing protocol is an interactive protocol between two parties: the seller P_j and the purchaser P_i. Naturally, before purchasing a product its corresponding product-public key is verified. Only if this is valid, the protocol will be executed. For two honest parties the purchasing process will successfully finish, whereas the simulator $\mathcal{S}$ determines the outcome of the protocol execution in any other case.

Rating a Product: When party P_i wants to rate the product *prod* with public key *ppk* owned by party P_j, P_i must be registered, must have purchased the specified product, and must not have rated the product before. Being registered is necessary to *open* ratings, whereas having purchased the product enables rating verifiers to detect self-ratings, bad mouthing attacks and ballot stuffing attacks. In the case that P_{IDM} is honest, $\mathcal{F}_{\mathsf{RS}}$ guarantees *anonymity of raters*: the simulator $\mathcal{S}$ is asked to output a *rating* σ, that is valid for the specified product, without knowing the rating party. Hence, the output rating cannot depend on the raters' identity. In the case that P_{IDM} is corrupted, the simulator $\mathcal{S}$ obtains the identity of the rater, because in this case anonymity cannot be achieved.

Rating Verification and Determining the Raters' Identity: Given the right parameters, every rating can be verified. Note that ratings are only verified, if the specified product is valid. A *valid rating* guarantees the following properties, even for maliciously generated ratings:

- Non-Self-Rating: the rater is not the owner of the product.
- Linkability: the rater purchased the product (will be discussed later in detail).
- Traceability: the rater is registered and can be identified.

Every single property is crucial for trustworthy reputation. If self-ratings would not be prevented, ballot stuffing attacks were possible. The same holds for linkability, but this will be discussed later in detail. Being able to open ratings is also very important in practical applications, because otherwise misbehaving parties can not be identified and punished. Hence, it must be guaranteed that honest parties are not blamed having rated some product, when they did not. This property is called *non-frameability* and is discussed later in detail.

$\mathcal{F}_{\mathsf{RS}}$ not only asks the simulator $\mathcal{S}$ to validate a rating, but also to determine the raters' identity. This models the ability of P_{IDM} to open *every* rating, not only those for which an Open-request occurs. Furthermore, it simplifies the definition of $\mathcal{F}_{\mathsf{RS}}$ without weakening the security properties, because VfyRtg encapsulates all important characteristics of a valid rating in a single and reusable procedure.

Linking Ratings and RebLDB: For every party using a reputation system it is important to know whether two valid ratings for the same product are generated by the same party. If this is true, the rater behaved dishonestly. We call this property *linkability*, which prevents bad mouthing attacks and ballot stuffing attacks. Linkability represents an equivalence relation: $\mathsf{Link}(x, x) = 1$, $\mathsf{Link}(x, y) = \mathsf{Link}(y, x)$ and $\mathsf{Link}(x, y) = 1 \wedge \mathsf{Link}(y, z) = 1 \Rightarrow \mathsf{Link}(x, z) = 1$. The value *lid* stored by $\mathcal{F}_{\mathsf{RS}}$ for every rating represents the equivalence class the rating belongs to. Initially, *lid* is set to the current value of a global counter *lidc*. The linking-class identifiers are updated by the RebLDB algorithm whenever a new rating is added to the list $\mathfrak{Ratings}$ (via Rate and Verify) or new linking information is obtained (via Link and Judge). This algorithm is only for internal use and not callable by any party. The RebLDB-algorithm merges two equivalence classes in the following cases:

- Step 2 covers calls to the algorithm from Rate, Verify, and Judge ($s = \perp$), where P_{IDM} is not corrupted and/or X_1 is an uncorrupted rater ($X_1 \neq \perp$). In these cases RebLDB selects all valid ratings for the specified product from the same rater X_1 (the set $\mathcal{L}$) and sets the value *lid* ($\ell[9]$ for $\ell \in \mathcal{L}$) for all ratings in $\mathcal{L}$ to the minimal value within the selected ratings.
- Step 5 handles requests from Link where either the identity of the rater is not known but the simulator $\mathcal{S}$ tells $\mathcal{F}_{\mathsf{RS}}$ that these ratings are linkable (Step 6 of Link), or the identity of some corrupted party can be updated for some rating, because it is linkable to another rating $\mathcal{F}_{\mathsf{RS}}$ already knows the identity of (Step 9 in Link). According to the transitivity of the linkability relation, RebLDB merges the two equivalence classes into one class by selecting all ratings within the two classes (Step 9) and setting *lid* to be the smaller of both values. Additionally, if a party identity is given in X_1 or X_2 this value will be set for all ratings within the equivalence class (Step 10).
- In Steps 11–18 RebLDB verifies that there do not exist more equivalence classes for an honestly generated product than the party owning the product sold. This ensures that it is only possible to rate a product once (without being linkable) after purchasing.
  When P_{IDM} is corrupted, it is possible that no linking information is available to $\mathcal{F}_{\mathsf{RS}}$. In this case $\mathcal{F}_{\mathsf{RS}}$ asks the simulator $\mathcal{S}$ to link all ratings for the product in question. Without this step a simple attack is possible:
  - $\mathcal{Z}$ lets the real-world adversary $\mathcal{A}$ corrupt P_{IDM} and some party P_i, lets P_i purchase some product from an honest party P_j, generates multiple valid ratings for this product and verifies them.

- In this scenario $\mathcal{F}_{\mathsf{RS}}$ adds the ratings to $\mathfrak{Ratings}$ during the Verify-protocol, which in turn calls RebLDB. Since no linking information is available to $\mathcal{F}_{\mathsf{RS}}$, without Step 13 $\mathcal{F}_{\mathsf{RS}}$ outputs `error`, even when all ratings are linkable. Hence, no protocol can realize $\mathcal{F}_{\mathsf{RS}}$.

If after Step 13 there are still more equivalence classes than purchases, this violates the security requirements of $\mathcal{F}_{\mathsf{RS}}$.

Summarizing, the handling of equivalence classes is modeled by the RebLDB-algorithm which uses linking information obtained from the algorithms Rate, Verify, Link, and Judge.

Generating and Verifying Opening-Proofs: Opening-proofs are values that enable every party to verify that a blamed party is really the author of a given rating. This covers the property of *non-frameability*: no honest party can be accused being the author of a given rating, when it is not. $\mathcal{F}_{\mathsf{RS}}$ asks the simulator $\mathcal{S}$ to output *valid* opening-proofs and ignores the output of $\mathcal{S}$, if the given rating is invalid, a wrong identity is given or the rating has not been opened yet. Since there can be more than one valid opening-proof, the value *oid* is used to connect a rating with its list of opening-proofs. This mechanism ensures that an opening-proof cannot be used to determine a raters identity for other ratings.

3 Realizing $\mathcal{F}_{\mathsf{RS}}$

Before introducing the protocol that realizes $\mathcal{F}_{\mathsf{RS}}$, we give the required preliminaries and building blocks in this section.

Preliminaries. Our realization relies on bilinear groups, the Symmetric External Diffie-Hellman-Assumption, and the Pointcheval-Sanders-Assumption. For completeness, we give the respective definitions in this section.

Definition 1 (Bilinear Groups). *A bilinear group $\mathbb{GD}$ is a set of three cyclic groups $\mathbb{G}_1, \mathbb{G}_2$ and $\mathbb{G}_T$, each group of prime order p, along with a bilinear map $\mathrm{e}\colon \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ with the following properties:*

1. *Bilinearity: for all $u \in \mathbb{G}_1$, $v \in \mathbb{G}_2$ and $a, b \in \mathbb{Z}_p$: $\mathrm{e}(u^a, v^b) = \mathrm{e}(u, v)^{ab}$.*
2. *Non-degeneracy: for $u \neq 1_{\mathbb{G}_1}$ and $v \neq 1_{\mathbb{G}_2}$: $\mathrm{e}(u, v) \neq 1_{\mathbb{G}_T}$.*
3. *The map* e *is efficiently computable.*

We will use pairings of Type-3 for our construction, because they allow efficient implementations and the Pointcheval-Sanders-Assumption does not hold in Type-1 and Type-2 pairing groups. Furthermore, for Type-3 pairing groups it is believed that the Decisional-Diffie-Hellman-Problem is hard in both $\mathbb{G}_1$ and $\mathbb{G}_2$. This assumption is often referred to as the Symmetric External Diffie-Hellman-Assumption (SXDH) [19].

Definition 2 (Bilinear Group Generator). *A bilinear group generator, denoted by* BiGrGen*, is a probabilistic polynomial time algorithm that, on input 1^λ, outputs a description of a bilinear group $\mathbb{GD}$. We denote the output of* BiGrGen *by $\mathbb{GD} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, \mathrm{e}, g_1, g_2)$.*

Definition 3 (Pointcheval-Sanders-Problem – PS1). *Let* $\mathbb{GD} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, \mathrm{e}, g_1, g_2)$ *be a bilinear group setting of Type-3, with generators* $g_1 \in \mathbb{G}_1$ *and* $g_2 \in \mathbb{G}_2$. *Further, let* $g \leftarrow_u \mathbb{G}_1$, $\tilde{g} \leftarrow_u \mathbb{G}_2$, $X := g^x$, $Y := g^y \in \mathbb{G}_1$ *and* $\tilde{X} := \tilde{g}^x$, $\tilde{Y} := \tilde{g}^y \in \mathbb{G}_2$, *for* $x, y \leftarrow_u \mathbb{Z}_p$. *We define the oracle* $\mathcal{O}(m)$ *as follows: on input* $m \in \mathbb{Z}_p$, *choose* $h \leftarrow_u \mathbb{G}_1$ *and output* $(h, h^{x+m \cdot y})$. *Given* $(g, Y, \tilde{g}, \tilde{X}, \tilde{Y})$ *and unlimited access to oracle* $\mathcal{O}$, *the Pointcheval-Sanders-Problem is to output a tuple* $(m^*, s, s^{x+m^* \cdot y})$, *where* $s \neq 1_{\mathbb{G}_1}$ *and* m^* *was not asked to* $\mathcal{O}$.

We say the Pointcheval-Sanders-Assumption holds for bilinear group generator $\mathsf{BiGrGen}$ *if for all probabilistic polynomial time adversaries* $\mathcal{A}$ *there exists a negligible function* negl *such that*

$$\Pr\left[\mathcal{A}^{\mathcal{O}(\cdot)}\left(\mathbb{GD}, g, Y, \tilde{g}, \tilde{X}, \tilde{Y}\right) = \left(m^*, s, s^{x+m^* \cdot y}\right)\right] \leq \mathrm{negl}(\lambda),$$

where the probability is taken over the random bits used by $\mathsf{BiGrGen}, \mathcal{A}$, *and the random choices of* $x, y \leftarrow_u \mathbb{Z}_p$.

Building Blocks and Intuition for Our Realization. In this section we briefly introduce the building blocks of our realization and explain how they are combined to realize $\mathcal{F}_{\mathsf{RS}}$. Due to lack of space, all formal definitions are given in the full version of this paper [5].

We use *Pointcheval-Sanders Signatures* ($\mathsf{PS} = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$) [25] as certificates for registration and for purchased products. We call the certificate for registration a *registration token*, the certificate for purchased products a *rating token*. To obtain such tokens every user has to prove knowledge of a self-chosen *user-secret-key usk*. We use the concurrent zero-knowledge variant of Σ-protocols, which uses *Trapdoor Pedersen Commitments* ($\mathsf{PD} = (\mathsf{KeyGen}, \mathsf{Commit}, \mathsf{Reveal}, \mathsf{Equiv})$) for this purpose.

To rate a product a user has to non-interactively prove knowledge of the registration token, the rating token, and its personal user-secret, for which the tokens were generated. As non-interactive proof system we use Signatures of Knowledge [12]. Also, *opening-proofs*, generated by P_{IDM}, are non-interactive proofs of knowledge of *opening tokens*. These tokens are given by a user P_i to the System Manager P_{IDM} during the registration protocol. In our construction it is important not to publish these tokens, because they allow to open any rating. Hence, we encrypt opening tokens with the CCA2-secure *Cramer-Shoup encryption* ($\mathsf{CS} = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$) [14].

The Signatures of Knowledge we use need a Random Oracle, which can be modeled as the ideal functionality $\mathcal{F}_{\mathsf{RO}}$ [22] in the UC framework. We further need the ideal functionalities for *Common Reference Strings* $\mathcal{F}_{\mathsf{CRS}}$ [11] and *Certification* $\mathcal{F}_{\mathsf{CA}}$ [10]. $\mathcal{F}_{\mathsf{CRS}}$ is needed for secure commitment schemes like the above mentioned Trapdoor Pedersen Commitments and $\mathcal{F}_{\mathsf{CA}}$ ensures that users cannot register with different identities.

The output of $\mathcal{F}_{\mathsf{CRS}}$ is $(\mathbb{GD}, \mathsf{PD}.pk, \mathcal{H}, \mathcal{H}_1, \mathcal{H}_2)$, where $\mathbb{GD}$ is the output of the bilinear group generator $\mathsf{BiGrGen}(1^\lambda)$, $\mathsf{PD}.pk = (u, v) \in \mathbb{G}_1^2$ is the public key of the Trapdoor Pedersen Commitment scheme, and $\mathcal{H}\colon \{0,1\}^* \to \mathbb{Z}_p$,

$\mathcal{H}_1 \colon \{0,1\}^* \to \mathbb{G}_1$, and $\mathcal{H}_2 \colon \{0,1\}^* \to \mathbb{G}_2$ are collision-resistant hash functions. We assume that every party obtains the common-reference string prior to its first activation. We write $y \coloneqq \mathcal{F}_{\mathsf{RO}}(x)$ to indicate a call to $\mathcal{F}_{\mathsf{RO}}$ on input (sid, x) and outputting y to the calling party.

A Protocol for Realizing $\mathcal{F}_{\mathsf{RS}}$. We assume to communicate via authenticated channels between two parties. This implies that the identities of communicating parties are known to each other and that the adversary cannot modify the message's payload.

Π_{RS}

All parties except P_{IDM}: On the first activation of P_i

1: Choose $usk_i \xleftarrow{\$} \mathbb{Z}_p$ and compute $M_i \coloneqq g_1^{usk_i}$, where $g_1 \in \mathbb{G}_1$ is given by $\mathcal{F}_{\mathsf{CRS}}$.
2: Send $(\mathsf{Register}, P_i, M_i)$ to $\mathcal{F}_{\mathsf{CA}}$, and store the user-secret-key usk_i.

Registry Key Generation: When P_{IDM} receives (KeyGen, sid) from $\mathcal{Z}$

1: Run $\mathsf{PS.KeyGen}(\mathbb{GD})$ to obtain $\mathsf{PS}.pk \coloneqq (\tilde{g}, \tilde{X}, \tilde{Y})$ and $\mathsf{PS}.sk \coloneqq (\xi_1, \xi_2)$.
2: Run $\mathsf{CS.Setup}(\mathbb{GD})$ to obtain $\mathsf{CS}.pk \coloneqq (g_2, \tilde{h}, \tilde{b}, \tilde{d}, \tilde{f}, \mathcal{H})$ and $\mathsf{CS}.sk \coloneqq (\zeta_1, \zeta_2, \zeta_3, \zeta_4, \zeta_5)$.
3: Set $pp \coloneqq (\mathsf{PS}.pk, \mathsf{CS}.pk)$ and $idmsk \coloneqq (\mathsf{PS}.sk, \mathsf{CS}.sk)$.
4: Set $\mathfrak{Params}.\mathsf{Add}(pp)$ and $\mathfrak{Params}_s.\mathsf{Add}(pp, idmsk)$.
5: Output $(\mathsf{KeyGen}, sid, pp)$.

User Registration: When P_i receives $(\mathsf{Register}, sid, pp')$ from $\mathcal{Z}$

P_i: 1: Choose $\alpha, r \xleftarrow{\$} \mathbb{Z}_p$, compute $T \coloneqq g_1^{\alpha}$, $R \coloneqq u^{\mathcal{H}(T)} \cdot v^r$, and send (pp', R) to P_{IDM}.

P_{IDM}: 2: Obtain M_i from $\mathcal{F}_{\mathsf{CA}}(\mathsf{Retrieve}, P_i)$.
3: **If** $\mathcal{F}_{\mathsf{CA}}$ returned $(\mathsf{Retrieve}, P_i, \perp)$, $pp' \notin \mathfrak{Params}$ or $(P_i, pp', M', Y', \sigma') \in \mathfrak{Reg}$ for some M', Y', σ' **then**
4: Send `abort` to P_i and output $(\mathsf{Register}, sid, pp', P_i, 0)$.
5: **Else** Choose $ch \xleftarrow{\$} \mathbb{Z}_p$ and send ch to P_i.

P_i: 6: **If** P_{IDM} sent `abort` **then** output $(\mathsf{Register}, sid, pp', P_i, 0)$.
7: **Else** Compute $s_\alpha \coloneqq \alpha + ch \cdot usk_i$, $ct \leftarrow \mathsf{CS.Enc}(\mathsf{CS}.pk, \tilde{Y}^{usk_i})$ and send (s_α, T, r, ct) to P_{IDM}.

P_{IDM}: 8: Compute $\tilde{Y}_i \coloneqq \mathsf{CS.Dec}(\mathsf{CS}.sk, ct)$.
9: **If** decrypting ct failed, or $M_i^{ch} \cdot T \neq g_1^{s_\alpha}$, or $R \neq u^{\mathcal{H}(T)} \cdot v^r$, or $\mathrm{e}(M_i, \tilde{Y}) \neq \mathrm{e}(g_1, \tilde{Y}_i)$ **then** send `abort` to P_i and output $(\mathsf{Register}, sid, pp', P_i, 0)$.
10: **Else** Compute $\sigma_i \leftarrow \mathsf{PS.Sign}(\mathsf{PS}.sk, M_i)$, and set $\mathfrak{Reg}.\mathsf{Add}(P_i, pp', M_i, \tilde{Y}_i, \sigma_i)$
11: Send σ_i to P_i, and output $(\mathsf{Register}, sid, pp', P_i, 1)$.

P_i: 12: **If** P_{IDM} sent `abort` **then** output $(\mathsf{Register}, sid, pp', P_i, 0)$.
13: **Else If** $\mathsf{PS.Verify}(pp', usk_i, \sigma_i) = 1$ **then**
14: store (usk_i, σ_i), and output $(\mathsf{Register}, sid, pp', P_i, 1)$.
15: **Else** output $(\mathsf{Register}, sid, pp', P_i, 0)$.

Product Addition: When P_i receives $(\mathsf{NewProduct}, sid, prod)$ from $\mathcal{Z}$

1: Compute $\tilde{g}_{i,prod} := \mathcal{H}_2(i, prod)$ and run $\mathsf{PS.KeyGen}(\mathbb{GD})$ to obtain $\mathsf{PS}.pk_{i,prod} := (\tilde{g}_{i,prod}, \tilde{X}_{i,prod}, \tilde{Y}_{i,prod})$ and $\mathsf{PS}.sk_{i,prod} := (\xi 1_{i,prod}, \xi 2_{i,prod})$.
2: Compute $M_{i,prod} := \mathcal{H}_1(i, prod)^{usk_i}$.
3: Choose $r \xleftarrow{u} \mathbb{Z}_p$ and compute $R_1 := \mathcal{H}_1(i, prod)^r$ and $R_2 := g_1^r$.
4: Set $ch_{i,prod} := \mathcal{F}_{\mathsf{RO}}(\mathsf{PS}.pk_{i,prod}, M_i, M_{i,prod}, R_1, R_2)$ and $s_{i,prod} := r + ch_{i,prod} \cdot usk_i$.
5: Set $ppk_{i,prod} := (M_i, M_{i,prod}, ch_{i,prod}, s_{i,prod}, \mathsf{PS}.pk_{i,prod})$.
6: Set $\mathfrak{Prods}_i.\mathsf{Add}(prod, ppk_{i,prod})$.
7: Output $(\mathsf{NewProduct}, sid, prod, ppk_{i,prod})$.

Purchase: When P_i receives $(\mathsf{Purchase}, sid, P_j, prod, ppk)$ from $\mathcal{Z}$

P_i: 1: **If** $\mathsf{VfyProd}(P_j, prod, ppk) = 0 \vee P_i = P_j$ **then** ignore the request.
2: **Else** choose $\alpha, r \xleftarrow{u} \mathbb{Z}_p$, compute $T := g_1^{\alpha}$, $R := u^{\mathcal{H}(T)} \cdot v^r$
3: Send $(prod, ppk, R)$ to P_j.
P_j: 4: Obtain M_i from $\mathcal{F}_{\mathsf{CA}}(\mathsf{Retrieve}, P_i)$.
5: **If** $\mathcal{F}_{\mathsf{CA}}$ returned $(\mathsf{Retrieve}, P_i, \bot)$ or $(prod, ppk) \notin \mathfrak{Prods}_j$ **then**
6: send `abort` to P_i and output $(\mathsf{Purchase}, sid, P_i, P_j, prod, ppk, 0)$.
7: **Else** choose $ch \xleftarrow{u} \mathbb{Z}_p$ and send ch to P_i.
P_i: 8: **If** P_j sent `abort` **then** output $(\mathsf{Purchase}, sid, P_i, P_j, prod, ppk, 0)$.
9: **Else** compute $s_\alpha := \alpha + ch \cdot usk_i$ and send (s_α, T, r) to P_j.
P_j:10: **If** $M_i^{ch} \cdot T \neq g_1^{s_\alpha}$ or $R \neq u^{\mathcal{H}(T)} \cdot v^r$ **then**
11: send `abort` to P_i and output $(\mathsf{Purchase}, sid, P_i, P_j, prod, ppk, 0)$.
12: **Else** compute $\sigma_{i,j,prod} \leftarrow \mathsf{PS.Sign}(\mathsf{PS}.sk_{i,prod}, M_i)$
13: Set $\mathfrak{Purch}_j.\mathsf{Add}(P_i, prod, \sigma_{i,j,prod})$ and send $\sigma_{i,j,prod}$ to P_i.
P_i:14: **If** P_j sent `abort` **then** output $(\mathsf{Purchase}, sid, P_i, P_j, prod, ppk, 0)$.
15: **Else If** $\mathsf{PS.Verify}(\mathsf{PS}.pk_{i,prod}, usk_i, \sigma_{i,j,prod}) = 1$ **then**
16: store $\sigma_{i,j,prod}$, and output $(\mathsf{Purchase}, sid, P_i, P_j, prod, ppk, 1)$.
17: **Else** output $(\mathsf{Purchase}, sid, P_i, P_j, prod, ppk, 0)$.

VfyProd: On local input $(P_j, prod, ppk)$

1: Obtain M_j from $\mathcal{F}_{\mathsf{CA}}(\mathsf{Retrieve}, P_j)$
2: **If** $\mathcal{F}_{\mathsf{CA}}$ returned $(\mathsf{Retrieve}, P_j, \bot)$ **then** return 0.
3: **Else** parse ppk as $(M'_j, M_{j,prod}, ch_{j,prod}, s_{j,prod}, \mathsf{PS}.pk_{j,prod})$.
4: Set $R_1 := \mathcal{H}_1(j, prod)^{s_{j,prod}} \cdot M_{j,prod}^{-ch_{j,prod}}$ and $R_2 := g_1^{s_{j,prod}} \cdot M_j^{-ch_{j,prod}}$.
5: **If** $M_j \neq M'_j$ or $ch_{j,prod} \neq \mathcal{F}_{\mathsf{RO}}(\mathsf{PS}.pk_{j,prod}, M_j, M_{j,prod}, R_1, R_2)$ **then** return 0.
6: **Else** return 1.

Rate a Product: When P_i receives $(\mathsf{Rate}, sid, pp, P_j, prod, ppk, m)$ from $\mathcal{Z}$

1: **If** no tuple (usk_i, σ_i) is stored such that $\mathsf{PS.Verify}(pp, usk_i, \sigma_i) = 1$, or no $\sigma_{i,j,prod}$ is stored such that $\mathsf{PS.Verify}(\mathsf{PS}.pk_{i,prod}, usk_i, \sigma_{i,j,prod}) = 1$, or a tuple $(m', \sigma) = (m', T_1, T_2, T_3, T_4, T_5, ch, s)$ is stored such that $(\mathsf{Verify}, sid, pp, P_j, prod, ppk, m', \sigma) = 1$ **then** ignore the request.
2: Choose $t_1, t_2, k \xleftarrow{u} \mathbb{Z}_p$ and compute $T_1 := \sigma_{i,1}^{t_1}$, $T_2 := \sigma_{i,2}^{t_1}$, $T_3 := \sigma_{i,j,prod,1}^{t_2}$,
3: $T_4 := \sigma_{i,j,prod,2}^{t_2}$, $T_5 := \mathcal{H}_1(j, prod)^{usk_i}$.

4: Compute $R_1 := \mathrm{e}(T_1, \tilde{Y})^k$, $R_2 := \mathrm{e}(T_3, \tilde{Y}_{j,prod})^k$, $R_3 := \mathcal{H}_1(j, prod)^k$.
5: Set $ch := \mathcal{F}_{\mathsf{RO}}(T_1, T_2, T_3, T_4, T_5, R_1, R_2, R_3, prod, ppk, m)$, and $s := k + ch \cdot usk_i$.
6: Set $\sigma := (T_1, T_2, T_3, T_4, T_5, ch, s)$ and store σ.
7: Output $(\mathsf{Rate}, sid, pp, P_j, prod, ppk, m, \sigma)$.

Verifying a Rating: When P_i receives $(\mathsf{Verify}, sid, pp, P_j, prod, ppk, m, \sigma)$ from $\mathcal{Z}$
1: **If** $\mathsf{VfyProd}(P_j, prod, ppk) = 0$ **then** ignore the request.
2: Parse σ as $(T_1, T_2, T_3, T_4, T_5, ch, s)$.
3: Set $R_1' := \mathrm{e}(T_1, \tilde{X})^{ch} \cdot \mathrm{e}(T_2, \tilde{g})^{-ch} \cdot \mathrm{e}(T_1, \tilde{Y})^s$, $R_3' := T_5^{-ch} \cdot \mathcal{H}_1(j, prod)^s$,
$R_2' := \mathrm{e}(T_3, \tilde{X}_{j,prod})^{ch} \cdot \mathrm{e}(T_4, \tilde{g}_{j,prod})^{-ch} \cdot \mathrm{e}(T_3, \tilde{Y}_{j,prod})^s$.
4: Set $f := \left[T_5 \neq M_{j,prod} \wedge ch = \mathcal{F}_{\mathsf{RO}}(T_1, T_2, T_3, T_4, T_5, R_1', R_2', R_3', prod, ppk, m)\right]$
5: Output $(\mathsf{Verify}, sid, pp, P_j, prod, ppk, m, \sigma, f)$.

Link Ratings: When P_i receives $(\mathsf{Link}, sid, pp, P_j, prod, ppk, m_0, \sigma_0, m_1, \sigma_1)$ from $\mathcal{Z}$
1: **If** $(\mathsf{Verify}, sid, pp, P_j, prod, ppk, m_k, \sigma_k) = 1$, for $k \in \{0, 1\}$ **then**
2: Parse σ_0 as $(T_1, T_2, T_3, T_4, T_5, ch, s)$, and σ_1 as $(T_1', T_2', T_3', T_4', T_5', ch', s')$.
3: Output $(\mathsf{Link}, sid, pp, P_j, prod, ppk, m_0, \sigma_0, m_1, \sigma_1, (T_5 = T_5'))$.
4: **Else** Output $(\mathsf{Link}, sid, pp, P_j, prod, ppk, m_0, \sigma_0, m_1, \sigma_1, 0)$.

Determine Raters Identity: When P_{IDM} receives $(\mathsf{Open}, sid, pp, P_j, prod, ppk, m, \sigma)$ from $\mathcal{Z}$
1: **If** $pp \notin \mathfrak{Params}$ **then** ignore the request.
2: Set $f := (\mathsf{Verify}, sid, pp, P_j, prod, ppk, m, \sigma)$.
3: **If** $f = 1$ **then** parse σ as $(T_1, T_2, T_3, T_4, T_5, ch, s)$ and iterate through $\mathfrak{Reg}$ to find a tuple $(P_i, pp, M_i, \tilde{Y}_i, \sigma_i)$ such that $\mathrm{e}(T_5, \tilde{Y}) = \mathrm{e}(\mathcal{H}_1(j, prod), \tilde{Y}_i)$.
4: **If** $f = 0$ or no such tuple could be found **then**
5: Output $(\mathsf{Open}, sid, pp, P_j, prod, ppk, m, \sigma, \perp)$.
6: **Else** Set $\mathfrak{Open}.\mathsf{Add}(pp, P_j, prod, ppk, m, \sigma, P_i)$.
7: Output $(\mathsf{Open}, sid, pp, P_j, prod, ppk, m, \sigma, P_i)$.

Generate Opening Proof: When P_{IDM} receives $(\mathsf{OProof}, sid, pp, P_j, prod, ppk, m, \sigma, P)$ from $\mathcal{Z}$
1: **If** $pp \notin \mathfrak{Params}$ **then** ignore the request.
2: Set $f := (\mathsf{Verify}, sid, pp, P_j, prod, ppk, m, \sigma)$.
3: **If** $f = 0 \vee (pp, P_j, prod, ppk, m, \sigma, P) \notin \mathfrak{Open}$ **then**
4: Output $(\mathsf{OProof}, sid, pp, P_j, prod, ppk, m, \sigma, P, \perp)$.
5: **Else** Parse σ as $(T_1, T_2, T_3, T_4, T_5, ch, s)$.
6: Select the tuple $(P, pp, M_i, \tilde{Y}_i, \sigma_i)$ such that $\mathrm{e}(T_5, \tilde{Y}) = \mathrm{e}(\mathcal{H}_1(j, prod), \tilde{Y}_i)$.
7: Choose $\beta \leftarrow\!\!\$\, \mathbb{Z}_p$ and compute $ct = (ct_1, ct_2, ct_3, ct_4) \leftarrow \mathsf{CS.Enc}(\mathsf{CS}.pk, \tilde{Y}_i; \beta)$.
8: Choose $r \leftarrow\!\!\$\, \mathbb{Z}_p$ and compute $R_1 := g_2^r$, $R_2 := \tilde{h}^r$, $R_3 := \mathrm{e}(\mathcal{H}_1(j, prod), \tilde{f})^r$.
9: Compute $\omega := \mathcal{H}(ct_1, ct_2, ct_3)$, $R_4 := (\tilde{b} \cdot \tilde{d}^{\omega})^r$, $R_5 := \mathrm{e}(g_1, \tilde{f})^r$.
10: Set $\hat{ch} := \mathcal{F}_{\mathsf{RO}}(ct, R_1, R_2, R_3, R_4, R_5, \sigma, i, M_i)$.
11: Set $\hat{s} := r + \hat{ch} \cdot \beta$ and $\tau := (P_i, ct, \hat{ch}, \hat{s})$.
12: Set $\mathfrak{Open}.\mathsf{Add}(pp, P_j, prod, ppk, m, \sigma, P, \tau)$.
13: Output $(\mathsf{OProof}, sid, pp, P_j, prod, ppk, m, \sigma, P, \tau)$.

Verifying Opening-Proofs: When P_i receives $(\mathsf{Judge}, sid, pp, P_j, prod, ppk, m, \sigma, P, \tau)$ from $\mathcal{Z}$

1: Set $f \coloneqq (\mathsf{Verify}, sid, pp, P_j, prod, ppk, m, \sigma)$.
2: Obtain M from $\mathcal{F}_{\mathsf{CA}}(\mathsf{Retrieve}, P)$.
3: **If** $\mathcal{F}_{\mathsf{CA}}$ returned $(\mathsf{Retrieve}, P, \bot) \vee f = 0 \vee P = \bot \vee \tau = \bot$ **then**
4: output $(\mathsf{Judge}, sid, pp, P_j, prod, ppk, m, \sigma, P, \tau, 0)$.
5: **Else** Parse τ as $(P_i, (ct_1, ct_2, ct_3, ct_4), \hat{ch}, \hat{s})$.
6: Compute $R_1 \coloneqq ct_1^{-\hat{ch}} \cdot g_2^{\hat{s}}$, $R_2 \coloneqq ct_2^{-\hat{ch}} \cdot \tilde{h}^{\hat{s}}$.
7: Compute $R_3 \coloneqq \mathrm{e}(\mathcal{H}_1(j, prod), ct_3)^{-\hat{ch}} \cdot \mathrm{e}(T_5, \tilde{Y})^{\hat{ch}} \cdot \mathrm{e}(\mathcal{H}_1(j, prod), \tilde{f})^{\hat{s}}$
8: Compute $\omega \coloneqq \mathcal{H}(ct_1, ct_2, ct_3)$.
9: Compute $R_4 \coloneqq ct_4^{-\hat{ch}} \cdot (\tilde{b} \cdot \tilde{d}^{\omega})^{\hat{s}}$, $R_5 \coloneqq \mathrm{e}(g_1, ct_3)^{-\hat{ch}} \cdot \mathrm{e}(M, \tilde{Y})^{\hat{ch}} \cdot \mathrm{e}(g_1, \tilde{f})^{\hat{s}}$.
10: Set $f \coloneqq (P = P_i \wedge \hat{ch} = \mathcal{F}_{\mathsf{RO}}(ct, R_1, R_2, R_3, R_4, R_5, \sigma, i, M))$.
11: Output $(\mathsf{Judge}, sid, pp, P_j, prod, ppk, m, \sigma, P, \tau, f)$

Protocol 1: Protocol for $\mathcal{F}_{\mathsf{RS}}$

Theorem 1. *Under the* Authenticated Channels Assumption*, the* SXDH-Assumption*, the* Pointcheval-Sanders-Assumption*, and the assumption that $\mathcal{H}, \mathcal{H}_1$, and $\mathcal{H}_2$ are collision-resistant hash functions, Protocol Π_{RS} UC-realizes the $\mathcal{F}_{\mathsf{RS}}$ functionality in the $(\mathcal{F}_{\mathsf{RO}}, \mathcal{F}_{\mathsf{CRS}}, \mathcal{F}_{\mathsf{CA}})$-hybrid model, in the presence of static adversaries.*

Due to lack of space, we only sketch the proof here. The full proof is given in the full version of this paper [5].

Proof (Sketch). To prove Theorem 1 we have to show that for any probabilistic polynomial-time real-world adversary $\mathcal{A}$ there exists a probabilistic polynomial-time ideal-world adversary $\mathcal{S}$ such that for any probabilistic polynomial-time environment $\mathcal{Z}$ it holds:

$$\left\{\mathrm{EXEC}_{\mathcal{F}_{\mathsf{RS}}, \mathcal{S}^{\mathcal{A}}, \mathcal{Z}}(1^{\lambda}, z)\right\}_{\lambda \in \mathbb{N}, z \in \{0,1\}^*} \overset{c}{\equiv} \left\{\mathrm{EXEC}^{\mathcal{F}_{\mathsf{RO}}, \mathcal{F}_{\mathsf{CRS}}, \mathcal{F}_{\mathsf{CA}}}_{\Pi_{\mathsf{RS}}, \mathcal{A}, \mathcal{Z}}(1^{\lambda}, z)\right\}_{\lambda \in \mathbb{N}, z \in \{0,1\}^*}.$$

We divide the proof of this statement into three parts. In the first part we define the simulator $\mathcal{S}$ that interacts with $\mathcal{F}_{\mathsf{RS}}$ and simulates the cryptographic computations. Note that during Rate-requests $\mathcal{S}$ does not obtain any identifying information of the rater. Hence, $\mathcal{S}$ uses the zero-knowledge simulator for the Signature of Knowledge that represents a rating. Analogously, opening-proofs are represented by a Signature of Knowledge. Therefore, $\mathcal{S}$ uses the corresponding zero-knowledge simulator to generate opening-proofs.

In the second part of the proof we define a hybrid game $\mathcal{G}$ and a corresponding simulator $\mathcal{S}_1$ for which we prove that no environment $\mathcal{Z}$ can distinguish whether it interacts with $(\mathcal{F}_{\mathsf{RS}}, \mathcal{S})$ or $(\mathcal{G}, \mathcal{S}_1)$. In this game $\mathcal{S}_1$ obtains all identifying information during Rate-requests and therefore can execute the computations as defined in Protocol Π_{RS}. Also opening-proofs can be generated by $\mathcal{S}_1$ as in Protocol Π_{RS}. Hence, an environment $\mathcal{Z}$ is only able to distinguish $(\mathcal{F}_{\mathsf{RS}}, \mathcal{S})$ and

$(\mathcal{G}, \mathcal{S}_1)$, if it can distinguish between simulated and real ratings and opening-proofs. Under the *SXDH-Assumption* this is not possible.

In the third part of the proof we show that $\mathcal{S}_1$ executes exactly the same computations as Protocol Π_{RS}. This implies that any environment $\mathcal{Z}$ that distinguishes between $(\mathcal{G}, \mathcal{S}_1)$ and $(\Pi_{\mathsf{RS}}, \mathcal{A})$ is able to let $\mathcal{F}_{\mathsf{RS}}$ output `error`, whereas the Protocol Π_{RS} outputs some value, or $\mathcal{F}_{\mathsf{RS}}$ outputs 0, whereas Protocol Π_{RS} outputs 1 (or vice versa). Using different reductions to the Pointcheval-Sanders-Problem and to the CCA2-security of the Cramer-Shoup encryption scheme we show that such environments cannot exist. Hence, Π_{RS} UC-realizes $\mathcal{F}_{\mathsf{RS}}$ in the $(\mathcal{F}_{\mathsf{RO}}, \mathcal{F}_{\mathsf{CRS}}, \mathcal{F}_{\mathsf{CA}})$-hybrid model. □

A Note on Revocation: Protocol Π_{RS} can be easily extended to support verifier-local revocation, which revokes a user completely: to revoke the party P_i the System Manager P_{IDM}, or even P_i himself, publishes the value $\tilde{Y}_i$ as the users' revocation token rt_i on a revocation-list $\mathcal{RL}$. Then any verifier can check whether the author of a given rating $\sigma = (T_1, T_2, T_3, T_4, T_5, ch, s)$ is revoked by testing if the equation $\mathrm{e}(T_5, \tilde{Y}) = \mathrm{e}(\mathcal{H}_1(j, prod), rt)$ holds for any entry $rt \in \mathcal{RL}$. Analogously, during Purchase-requests the product owner can test whether $\mathrm{e}(M_i, \tilde{Y}) = \mathrm{e}(g_1, rt)$ holds to detect a revoked user P_i. This revocation mechanism conflicts with our definition of anonymity and it is an open problem how to prove security when revocation is considered.

Considering Adaptive Adversaries: Theorem 1 only claims security against static adversaries, because anonymity and linkability are conflicting security properties, which impede the construction of UC-secure protocols in the presence of adaptive adversaries. We leave this as an open problem that needs further research.

References

1. Androulaki, E., Choi, S.G., Bellovin, S.M., Malkin, T.: Reputation systems for anonymous networks. In: Borisov, N., Goldberg, I. (eds.) PETS 2008. LNCS, vol. 5134, pp. 202–218. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70630-4_13
2. Ateniese, G., Camenisch, J., Hohenberger, S., de Medeiros, B.: Practical group signatures without random oracles. Cryptology ePrint Archive, Report 2005/385 (2005). http://eprint.iacr.org/2005/385
3. Bellare, M., Micciancio, D., Warinschi, B.: Foundations of group signatures: formal definitions, simplified requirements, and a construction based on general assumptions. In: Biham, E. (ed.) EUROCRYPT 2003. LNCS, vol. 2656, pp. 614–629. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-39200-9_38
4. Bellare, M., Shi, H., Zhang, C.: Foundations of group signatures: the case of dynamic groups. In: Menezes, A. (ed.) CT-RSA 2005. LNCS, vol. 3376, pp. 136–153. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-30574-3_11
5. Blömer, J., Eidens, F., Juhnke, J.: Practical, anonymous, and publicly linkable universal-composable reputation systems. Cryptology ePrint Archive, Report 2018/029 (2018). http://eprint.iacr.org/2018/029

6. Blömer, J., Juhnke, J., Kolb, C.: Anonymous and publicly linkable reputation systems. In: Böhme, R., Okamoto, T. (eds.) FC 2015. LNCS, vol. 8975, pp. 478–488. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-47854-7_29
7. Boneh, D., Boyen, X., Shacham, H.: Short group signatures. In: Franklin, M. (ed.) CRYPTO 2004. LNCS, vol. 3152, pp. 41–55. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-28628-8_3
8. Busom, N., Petrlic, R., Sebé, F., Sorge, C., Valls, M.: A privacy-preserving reputation system with user rewards. J. Netw. Comput. Appl. **80**, 58–66 (2017)
9. Canetti, R.: Universally composable security: a new paradigm for cryptographic protocols. In: 42nd FOCS, pp. 136–145. IEEE Computer Society Press, October 2001
10. Canetti, R.: Universally composable signature, certification, and authentication. In: CSFW-17, p. 219 (2004)
11. Canetti, R., Fischlin, M.: Universally composable commitments. In: Kilian, J. (ed.) CRYPTO 2001. LNCS, vol. 2139, pp. 19–40. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44647-8_2
12. Chase, M., Lysyanskaya, A.: On signatures of knowledge. In: Dwork, C. (ed.) CRYPTO 2006. LNCS, vol. 4117, pp. 78–96. Springer, Heidelberg (2006). https://doi.org/10.1007/11818175_5
13. Clauß, S., Schiffner, S., Kerschbaum, F.: k-anonymous reputation. In: Chen, K., Xie, Q., Qiu, W., Li, N., Tzeng, W.G. (eds.) ASIACCS 13, pp. 359–368. ACM Press, May 2013
14. Cramer, R., Shoup, V.: A practical public key cryptosystem provably secure against adaptive chosen ciphertext attack. In: Krawczyk, H. (ed.) CRYPTO 1998. LNCS, vol. 1462, pp. 13–25. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0055717
15. Damgård, I., David, B., Giacomelli, I., Nielsen, J.B.: Compact VSS and efficient homomorphic UC commitments. In: Sarkar, P., Iwata, T. (eds.) ASIACRYPT 2014. LNCS, vol. 8874, pp. 213–232. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-45608-8_12
16. Damgård, I.: On σ-protocols (2002). http://www.daimi.au.dk/~ivan/Sigma.ps
17. Dellarocas, C.: Immunizing online reputation reporting systems against unfair ratings and discriminatory behavior. In: EC 2000, pp. 150–157. ACM (2000)
18. Fujisaki, E., Suzuki, K.: Traceable ring signature. In: Okamoto, T., Wang, X. (eds.) PKC 2007. LNCS, vol. 4450, pp. 181–200. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71677-8_13
19. Ghadafi, E., Smart, N.P., Warinschi, B.: Groth–Sahai proofs revisited. In: Nguyen, P.Q., Pointcheval, D. (eds.) PKC 2010. LNCS, vol. 6056, pp. 177–192. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13013-7_11
20. Hasan, O., Brunie, L., Bertino, E., Shang, N.: A decentralized privacy preserving reputation protocol for the malicious adversarial model. IEEE Trans. Inf. Forensics Secur. **8**(6), 949–962 (2013)
21. Hoffman, K., Zage, D., Nita-Rotaru, C.: A survey of attack and defense techniques for reputation systems. ACM Comput. Surv. **42**, 1–31 (2009)
22. Hofheinz, D., Müller-Quade, J.: Universally composable commitments using random oracles. In: Naor, M. (ed.) TCC 2004. LNCS, vol. 2951, pp. 58–76. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24638-1_4
23. Kerschbaum, F.: A verifiable, centralized, coercion-free reputation system. In: WPES 2009, pp. 61–70. ACM (2009)
24. Petrlic, R., Lutters, S., Sorge, C.: Privacy-preserving reputation management. In: SAC 2014, pp. 1712–1718. ACM (2014)

25. Pointcheval, D., Sanders, O.: Short randomizable signatures. In: Sako, K. (ed.) CT-RSA 2016. LNCS, vol. 9610, pp. 111–126. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-29485-8_7
26. Steinbrecher, S.: Design options for privacy-respecting reputation systems within centralised internet communities. In: Fischer-Hübner, S., Rannenberg, K., Yngström, L., Lindskog, S. (eds.) SEC 2006. IIFIP, vol. 201, pp. 123–134. Springer, Boston, MA (2006). https://doi.org/10.1007/0-387-33406-8_11
27. Zhai, E., Wolinsky, D.I., Chen, R., Syta, E., Teng, C., Ford, B.: Anonrep: towards tracking-resistant anonymous reputation. In: NSDI, pp. 583–596 (2016)