UC Commitments for Modular Protocol Design and Applications to Revocation and Attribute Tokens [⋆]

Jan Camenisch[1], Maria Dubovitskaya[1], and Alfredo Rial[2⋆⋆]

[1] IBM Research – Zurich
[2] University of Luxembourg

Abstract. Complex cryptographic protocols are often designed from simple cryptographic primitives, such as signature schemes, encryption schemes, verifiable random functions, and zero-knowledge proofs, by bridging between them with commitments to some of their inputs and outputs. Unfortunately, the known universally composable (UC) functionalities for commitments and the cryptographic primitives mentioned above do not allow such constructions of higher-level protocols as hybrid protocols. Therefore, protocol designers typically resort to primitives with property-based definitions, often resulting in complex monolithic security proofs that are prone to mistakes and hard to verify.

We address this gap by presenting a UC functionality for non-interactive commitments that enables modular constructions of complex protocols within the UC framework. We also show how the new functionality can be used to construct hybrid protocols that combine different UC functionalities and use commitments to ensure that the same inputs are provided to different functionalities. We further provide UC functionalities for attribute tokens and revocation that can be used as building blocks together with our UC commitments. As an example of building a complex system from these new UC building blocks, we provide a construction (a hybrid protocol) of anonymous attribute tokens with revocation. Unlike existing accumulator-based schemes, our scheme allows one to accumulate several revocation lists into a single commitment value and to hide the revocation status of a user from other users and verifiers.

Keywords: universal composability, commitments, attribute tokens, revocation, vector commitments

1 Introduction

Complex cryptographic protocols are often designed from simple cryptographic primitives, such as signature schemes, encryption schemes, verifiable random functions, zero-knowledge proofs, and commitment schemes. Proving the security of such cryptographic protocols as well as verifying their security proofs are far from trivial and rather error-prone. Composability frameworks such as the Universal Composability (UC) framework [6] can help here. They guarantee that cryptographic primitives remain secure

[⋆] This work was supported by the European Commission through the Seventh Framework Programme, under grant agreements #321310 for the PERCY grant.

[⋆⋆] Work done while at IBM Research – Zurich.

under arbitrary composition and thus enable a modular design and security analysis of cryptographic protocols constructed from such primitives. That is, they allow one to describe higher-level protocols as hybrid protocols that use the ideal functionalities of primitives rather than their realizations. Unfortunately, the known UC functionalities for cryptographic primitives allow only for very simple hybrid protocols, and thus protocols found in the literature foremost use basic ideal functionalities, such as the common reference string functionality $\mathcal{F}_{\mathrm{CRS}}$, registration functionality $\mathcal{F}_{\mathrm{REG}}$, and secure message transmission functionality $\mathcal{F}_{\mathrm{SMT}}$, and resort to constructions with property-based primitives, which typically results in complex monolithic security proofs that are prone to mistakes and hard to verify.

Consider for instance a two-party protocol where one party needs to compute a complex function $\mathcal{F}$ (that might include commitments, signatures, encryption, and zero-knowledge proofs) on the input and send the output to the second party. The original input of the first party might be hidden from the second party. The most common approach for building such a protocol is to describe the ideal functionality for that function $\mathcal{F}$, provide a monolithic realization, and prove that the latter securely implements the former. Following this approach, however, will result into a complex security proof.

A better approach would be a modular construction that breaks down the complex function into smaller building blocks each realized by a separate functionality. This will result in much simpler and structured protocols and proofs. However, this construction approach requires a mechanism to ensure that the input values to different subfunctionalities are the same. The most natural way to implement such a mechanism is to use cryptographic commitment functionality ($\mathcal{F}_{\mathsf{COM}}$) and build a realization in the $\mathcal{F}_{\mathsf{COM}}$-hybrid model.

In a nutshell, the hybrid protocol would work as follows. The ideal functionalities of the building blocks are modified in such a way that they also accept commitments to the input values as input. When a party needs to guarantee that the inputs to two or more functionalities are equal, the party first sends that input value to the commitment functionality to obtain a commitment and the corresponding opening, and then sends the input to those functionalities along with the commitment and the opening. When the second party receives a commitment it can perform the verification without learning the original input value.

As a concrete example, consider a privacy-preserving attribute-based credential system [3] that uses a commitment to a revocation handle to bridge a proof of knowledge of a signature that signs the revocation handle with a proof that the committed revocation handle is not revoked. The commitment guarantees that the same revocation handle is used in both proofs even if they are computed separately by different building blocks. This allows the composition of a protocol for proving possession of a signature with a protocol for proving non-revocation using commitments. The construction, definitions, and security proofs of such systems are all property-based and indeed rather complex [3]. Simplifying such a construction and its security proofs by using the UC model seems very attractive. However, that requires an ideal functionality for commitments that mirrors the way property-based primitives are combined with commitments. Unfortunately, none of the existing UC functionalities for commitments [6, 9, 8, 11, 15, 17, 16, 19, 12] fit this bill because they do not output cryptographic values or implement any other mechanism to ensure that a committed message is equal to the input of other functionalities. With the existing functionalities for commitments, the committer sends the committed message to the functionality, which informs the verifying party that a commitment has been sent. When the committer wishes to open the commitment, the functionality sends the message to the verifying party. Because no cryptographic value is ever output by the known functionalities, they cannot be used in our revocation example to guarantee the equality of the revocation handle or in any other similar case where one has to ensure that the message sent as input to the functionality for commitments equals the message sent as input to other functionalities. However, as we shall see, outputting just a cryptographic value for a commitment will not be sufficient.

1.1 UC Non-interactive Commitments for Hybrid Protocols

We provide a new ideal functionality $\mathcal{F}_{\mathrm{NIC}}$ for commitments. The main differences between $\mathcal{F}_{\mathrm{NIC}}$ and the existing commitment functionalities are that ours outputs cryptographic values and is non-interactive. In this respect it is similar to the signature functionality $\mathcal{F}_{\mathrm{SIG}}$ [8].

Our functionality behaves as follows. When a party wishes to commit to a message, $\mathcal{F}_{\mathrm{NIC}}$ computes a cryptographic commitment and an opening for that commitment (using algorithms provided by the simulator/environment upon initialization) and sends them as output to the calling party. When a party wishes to verify a commitment, it sends the commitment, the message and the opening to the functionality, which verifies the commitment and sends the verification result to the party. Therefore, our functionality does not involve interaction between a committer and a verifier. Furthermore, when a party requests a commitment to a message, the identity of the verifier is not sent to the functionality. Analogously, when a party verifies a commitment, the identity of the committer is not sent to the functionality.

$\mathcal{F}_{\mathrm{NIC}}$ ensures that commitments are hiding and binding. We show that $\mathcal{F}_{\mathrm{NIC}}$ can be realized by a standard commitment scheme that is binding and has a trapdoor (which implies it is hiding), such as the Pedersen commitment scheme [22]. All extra properties, such as non-malleability, simulation-sound trapdoor [18], etc., that are required to construct the standard UC functionalities are not necessary. We prove that the construction realizes $\mathcal{F}_{\mathrm{NIC}}$ in the $\mathcal{F}_{\mathrm{CRS}}$-hybrid model, which is also required for UC commitments in general [8].

There are protocols, however, that require extractable commitments. This is similar to requiring extractability in zero-knowledge proofs (ZKP). For some protocols, extractability is needed and thus a functionality for ZKP of knowledge must be used, whereas for other protocols sound ZKPs are sufficient and it is possible (and more efficient) to use a functionality for zero-knowledge that does not require extractability.

Therefore, we also propose an ideal functionality $\mathcal{F}_{\mathrm{ENIC}}$ for extractable commitments and give a construction that realizes $\mathcal{F}_{\mathrm{ENIC}}$. We compare both functionalities in Section 3.1 and explain why $\mathcal{F}_{\mathrm{NIC}}$ suffices for some cases.

1.2 Modular Protocol Design in $\mathcal{F}_{\mathrm{NIC}}$-Hybrid Model

Our ideal functionality for commitments can be used to construct higher-level protocols in a hybrid model because it allows one to bridge different ideal functionalities. To this end, the ideal functionalities of the building blocks can be modified so that their input values are accompanied by commitments and corresponding openings. These commitment and opening values are generated by the party providing the input using $\mathcal{F}_{\mathrm{NIC}}$. Then, to convince a second party that the same inputs were provided to different functionalities, the first party sends the commitments to the second party, who will then also input the commitments to the different functionalities. For this to work, the building-block functionalities need to validate the commitments received and check whether the openings provided are correct. As functionalities cannot interact with each other, a verification algorithm $\mathsf{COM.Verify}$ needs to be provided as part of the commitment for local verification. The main challenge now is to ensure that a local verification implies a global binding property enforced by $\mathcal{F}_{\mathrm{NIC}}$. We show how this challenge can be overcome.

We remark that our technique for modular protocol design based on $\mathcal{F}_{\mathrm{NIC}}$ is very general. Any functionality that needs to be used in a protocol can be amended to receive committed inputs and to check those inputs by running $\mathsf{COM.Verify}$. Therefore, our technique allows one to modularly describe a wide variety of hybrid protocols in the UC model. Moreover, we believe a similar approach could also be applied to functionalities that output cryptographic values that need to be verified inside other functionalities.

1.3 Example: Flexible Revocation for Attribute-Based Credentials

As a real-life example of building a complex system from our UC building blocks, we provide a construction for anonymous attribute tokens with revocation. We first provide the respective ideal functionalities for revocation and attribute tokens (signatures with the proofs of knowledge of signature possession). Then, we construct a protocol that uses those functionalities together with $\mathcal{F}_{\mathrm{NIC}}$ to compose a protocol for proving possession of a non-revoked credential (signature). In fact, unlike existing accumulator-based schemes, our new scheme allows one to accumulate several revocation lists into a single commitment value and to hide the revocation status of a user from other users and verifiers.

In the literature, different privacy-preserving revocation mechanisms have been proposed for attribute-based credentials, such as signature lists [20], accumulators [5, 21, 1], and validity refreshing [2]. We provide a detailed overview of the related work on revocation in the full version of this paper. In some cases, credentials need to be revoked globally, e.g., when the related secret keys have been exposed, the attribute values have changed, or the user loses her right to use a credential. Often, credentials may be revoked only for specific contexts, i.e., when a user is not allowed to use her credential with a particular verifier, but can still use it elsewhere.

In such scenarios, the revocation authority needs to maintain multiple revocation lists. Because of their binary value limitation, the existing revocation systems require a separate application of a revocation mechanism for each list. This imposes an extra storage and computational overhead, not only to the users, but also to the revocation authority. Furthermore, in signature lists and accumulators, the revocation lists are disclosed to the other users and verifiers.

We propose a mechanism that allows one to commit several revocation lists into a single commitment value. Each user needs only one witness for all the revocation lists. Using this witness, a user can prove in a privacy-preserving manner the revocation status of her revocation handle in a particular revocation list.

We provide two ideal functionalities $\mathcal{F}_{\mathrm{REV}}$ for revocation and propose two different constructions built from the vector commitments [10]. The first one hides the revocation status of a user from other users and from the verifiers, whereas in the second one, as for accumulators, revocation lists are public. Additionally, our schemes are flexible in the sense that revocation lists can be added (up to a maximum number) and removed without any cost, i.e., the cost is the same as for a revocation status update that does not change the number of lists, whereas accumulators would require one to set up a new accumulator and to issue witnesses to users, or delete them.

We note that aside from extending the standard revocation scenario with a central revocation authority and multiple revocation lists, our revocation schemes can be used to build an efficient dynamic attribute-based access control system in a very elegant way. Instead of issuing a list of credentials to each user, each certifying a certain attribute or role, in our revocation scheme a user can be issued just one base credential, which can be made valid or revoked for any context. The resulting solution saves the users, verifiers and the revocation authority a lot of storage and computational effort. That is, instead of having multiple credentials and corresponding revocation witnesses, a single credential and a single witness suffice to achieve the same goal.

1.4 Paper Organization

The remainder of this paper is organized as follows. In Section 2, we introduce the notation and conventions used to describe functionalities and their realizations in the UC model. In Section 3, we provide the ideal functionalities for non-interactive commitments and extractable commitments, and show the corresponding constructions that securely realize those functionalities. We also describe the generic approach of how to build modular constructions in the $\mathcal{F}_{\mathrm{NIC}}$-hybrid model and to prove them secure. In Section 5, we describe an ideal functionality for attribute tokens with revocation, $\mathcal{F}_{\mathrm{TR}}$, and provide a hybrid construction, Π_{TR}, that uses $\mathcal{F}_{\mathrm{NIC}}$, $\mathcal{F}_{\mathrm{REV}}$ and $\mathcal{F}_{\mathrm{AT}}$ to realize $\mathcal{F}_{\mathrm{TR}}$. We prove that the construction Π_{TR} realizes $\mathcal{F}_{\mathrm{TR}}$ in that section.

2 Universally Composable Security

The universal composability framework [6] is a framework for defining and analyzing the security of cryptographic protocols so that security is retained under arbitrary composition with other protocols. The security of a protocol is defined by means of an ideal protocol that carries out the desired task. In the ideal protocol, all parties send their inputs to an ideal functionality $\mathcal{F}$ for the task. The ideal functionality locally computes the outputs of the parties and provides each party with its prescribed output.

The security of a protocol φ is analyzed by comparing the view of an environment $\mathcal{Z}$ in a real execution of φ against that of $\mathcal{Z}$ in the ideal protocol defined in $\mathcal{F}_\varphi$. The environment $\mathcal{Z}$ chooses the inputs of the parties and collects their outputs. In the real world, $\mathcal{Z}$ can communicate freely with an adversary $\mathcal{A}$ who controls both the network and any corrupt parties. In the ideal world, $\mathcal{Z}$ interacts with dummy parties, who simply relay inputs and outputs between $\mathcal{Z}$ and $\mathcal{F}_\varphi$, and a simulator $\mathcal{S}$. We say that a protocol φ securely realizes $\mathcal{F}_\varphi$ if $\mathcal{Z}$ cannot distinguish the real world from the ideal world, i.e., $\mathcal{Z}$ cannot distinguish whether it is interacting with $\mathcal{A}$ and parties running protocol φ or with $\mathcal{S}$ and dummy parties relaying to $\mathcal{F}_\varphi$.

2.1 Notation

Let $k \in \mathbb{N}$ denote the security parameter and $a \in \{0,1\}^*$ denote an input. Two binary distribution ensembles $X = \{X(k,a)\}_{k\in\mathbb{N},a\in\{0,1\}^*}$ and $Y = \{Y(k,a)\}_{k\in\mathbb{N},a\in\{0,1\}^*}$ are indistinguishable ($X \approx Y$) if for any $c, d \in \mathbb{N}$ there exists $k_0 \in \mathbb{N}$ such that for all $k > k_0$ and all $a \in \cup_{\kappa \leq k^d}\{0,1\}^\kappa$, $|\Pr[X(k,a) = 1] - \Pr[Y(k,a) = 1]| < k^{-c}$. Let $\mathrm{REAL}_{\varphi,\mathcal{A},\mathcal{Z}}(k,a)$ denote the distribution given by the output of $\mathcal{Z}$ when executed on input a with $\mathcal{A}$ and parties running φ, and let $\mathrm{IDEAL}_{\mathcal{F}_\varphi,\mathcal{S},\mathcal{Z}}(k,a)$ denote the output distribution of $\mathcal{Z}$ when executed on input a with $\mathcal{S}$ and dummy parties relaying to $\mathcal{F}_\varphi$. We say that protocol φ securely realizes $\mathcal{F}_\varphi$ if, for all polynomial-time $\mathcal{A}$, there exists a polynomial-time $\mathcal{S}$ such that, for all polynomial-time $\mathcal{Z}$, $\mathrm{REAL}_{\varphi,\mathcal{A},\mathcal{Z}} \approx \mathrm{IDEAL}_{\mathcal{F}_\varphi,\mathcal{S},\mathcal{Z}}$.

A protocol $\varphi^{\mathcal{G}}$ securely realizes $\mathcal{F}$ in the $\mathcal{G}$-hybrid model when φ is allowed to invoke the ideal functionality $\mathcal{G}$. Therefore, for any protocol ψ that securely realizes functionality $\mathcal{G}$, the composed protocol φ^ψ, which is obtained by replacing each invocation of an instance of $\mathcal{G}$ with an invocation of an instance of ψ, securely realizes $\mathcal{F}$.

2.2 Conventions

When describing ideal functionalities, we use the following conventions:

Interface Naming Convention. An ideal functionality can be invoked by using one or more interfaces. The name of a message in an interface consists of three fields separated by dots, e.g., com.setup.ini in the commitment functionality described in Section 3.1. The first field indicates the name of the functionality and is the same in all interfaces of the functionality. This field is useful for distinguishing between invocations of different functionalities in a hybrid protocol that uses two or more different functionalities. The second field indicates the kind of action performed by the functionality and is the same in all messages that the functionality exchanges within the same interface. The third field distinguishes between the messages that belong to the same interface, and can take six different values. A message $*.*$.ini is the incoming message received by the functionality, i.e., the message through which the interface is invoked. A message $*.*$.end is the outgoing message sent by the functionality, i.e., the message that ends the execution of the interface. The message $*.*$.sim is used by the functionality to send a message to the simulator, and the message $*.*$.rep is used to receive a message from the simulator. The message $*.*$.req is used by the functionality to send a message to the simulator to request the description of algorithms from the simulator, and the message $*.*$.alg is used by the simulator to send the description of those algorithms to the functionality.

Subsession identifiers. Some interfaces in a functionality can be invoked more than once. When the functionality sends a message $*.*$.sim to the simulator in such an interface, a subsession identifier $ssid$ is included in the message. The subsession identifier must also be included in the response $*.*$.rep sent by the simulator. The subsession identifier is used to identify the message $*.*$.sim to which the simulator replies with a message $*.*$.rep. We note that, typically, the simulator in the security proof may not be able to provide an immediate answer to the functionality after receiving a message $*.*$.sim. The reason is that the simulator typically needs to interact with the copy of the real adversary it runs in order to produce the message $*.*$.rep, but the real adversary may not provide the desired answer or may provide a delayed answer. In such cases, when the functionality sends more than one message $*.*$.sim to the simulator, the simulator may provide delayed replies, and the order of those replies may not follow the order of the messages received.

Aborts. When we say that an ideal functionality $\mathcal{F}$ aborts after being activated with a message $(*, \ldots)$, we mean that $\mathcal{F}$ stops the execution of the instruction and sends a special abortion message $(*, \perp)$ to the party that invoked the functionality.

Network vs. local communication. The identity of an interactive Turing machine (ITM) instance (ITI) consists of a party identifier pid and a session identifier sid. A set of parties in an execution of a system of ITMs is a protocol instance if they have the same session identifier sid. ITIs can pass direct inputs to and outputs from "local" ITIs that have the same pid. An ideal functionality $\mathcal{F}$ has $pid = \perp$ and is considered local to all parties. An instance of $\mathcal{F}$ with the session identifier sid only accepts inputs from and passes outputs to machines with the same session identifier sid. Some functionalities require the session identifier to have some structure. Those functionalities check whether the session identifier possesses the required structure in the first message that invokes the functionality. For the subsequent messages, the functionality implicitly checks that the session identifier equals the session identifier used in the first message. Communication between ITIs with different party identifiers must take place over the network. The network is controlled by the adversary, meaning that he can arbitrarily delay, modify, drop, or insert messages.

Delayed outputs. We say that an ideal functionality $\mathcal{F}$ *sends a public delayed output* v to a party $\mathcal{P}$ if it engages in the following interaction. $\mathcal{F}$ sends to simulator $\mathcal{S}$ a note that it is ready to generate an output to $\mathcal{P}$. The note includes value v, identity $\mathcal{P}$, and a unique identifier for this output. When $\mathcal{S}$ replies to the note by echoing the unique identifier, $\mathcal{F}$ outputs the value v to $\mathcal{P}$. A *private delayed output* is similar, but value v is not included in the note.

3 UC Non-Interactive Commitments

In existing commitment functionalities [8], the committer sends the committed message to the functionality, which informs the verifying party that a commitment has been sent.

When the committer wishes to open the commitment, the functionality sends the message to the verifying party.

In contrast, our commitment functionalities do not involve any interaction between committer and verifier. In our commitment functionality, any party is allowed to request a commitment, and, when doing so, the identity of the verifier is not specified. Analogously, any party can verify a commitment, and the identity of the committer is not specified during verification.

In Section 3.1, we describe two ideal functionalities for non-interactive commitments $\mathcal{F}_{\mathrm{NIC}}$ and $\mathcal{F}_{\mathrm{ENIC}}$. Our commitment functionalities are similar to the functionalities of public key encryption and signatures [8, 14, 7]. For example, the signature functionality receives a message from the signer, computes a signature, and sends that signature to the signer. A verifying party sends a message and a signature to the functionality, which verifies the signature and sends the verification result. One of the reasons that the signature functionality has a "signature string" as part of its interface is to support the modularity of modeling complex protocols such as sending an encrypted signature [7].

Analogously, our ideal functionalities (unlike existing UC ideal functionalities for commitments) can be used in a hybrid protocol that also uses other functionalities that receive commitments as inputs. In a nutshell, a party would obtain a tuple $(ccom, cm, copen)$, which consists of a commitment, a message and an opening, from $\mathcal{F}_{\mathrm{NIC}}$ or $\mathcal{F}_{\mathrm{ENIC}}$, and send $(ccom, cm, copen)$ as input to the other functionalities. The use of commitments as input to those functionalities is useful when it is necessary to ensure that the inputs to those functionalities are equal.

For instance, our construction of anonymous attribute tokens with revocation in Section 5.2 uses an anonymous attribute token functionality, $\mathcal{F}_{\mathrm{AT}}$, and a revocation functionality, $\mathcal{F}_{\mathrm{REV}}$, that receive commitments output by $\mathcal{F}_{\mathrm{NIC}}$ as input. The commitments allow us to prove that the revocation handle used as input to $\mathcal{F}_{\mathrm{REV}}$ equals the one used as input to $\mathcal{F}_{\mathrm{AT}}$.

$\mathcal{F}_{\mathrm{ENIC}}$ requires commitments to be extractable, whereas $\mathcal{F}_{\mathrm{NIC}}$ does not. $\mathcal{F}_{\mathrm{NIC}}$ suffices for our construction of anonymous attribute tokens with revocation described in Section 5.2. The reason is that, in that construction, commitments are always sent along with their openings or along with proofs of knowledge of their openings, which provides the extraction property. In Section 3.4, we show that $\mathcal{F}_{\mathrm{NIC}}$ can be realized by any trapdoor and binding commitment scheme. We describe a construction for $\mathcal{F}_{\mathrm{ENIC}}$ and prove its security in the full version of this paper.

3.1 Ideal Functionalities $\mathcal{F}_{\mathrm{NIC}}$ and $\mathcal{F}_{\mathrm{ENIC}}$ for Non-Interactive Commitments

$\mathcal{F}_{\mathrm{NIC}}$ and $\mathcal{F}_{\mathrm{ENIC}}$ are parameterized with the system parameters sp. This allows the parameters of the commitment scheme to depend on parameters generated externally, which could also be used in other schemes. For example, if a commitment scheme is used together with a non-interactive zero-knowledge proof of knowledge scheme, sp could include parameters shared by both the parameters of the commitment scheme and the parameters of the proof of knowledge scheme.

$\mathcal{F}_{\mathrm{NIC}}$ and $\mathcal{F}_{\mathrm{ENIC}}$ interact with parties $\mathcal{P}_i$ that create the parameters of the commitment scheme and compute and verify commitments. The interaction between $\mathcal{F}_{\mathrm{NIC}}$

(or $\mathcal{F}_{\mathrm{ENIC}}$) and $\mathcal{P}_i$ takes place through the interfaces com.setup.$*$, com.validate.$*$, com.commit.$*$, and com.verify.$*$.

1. Any party $\mathcal{P}_i$ can call the interface com.setup.$*$ to initialize the functionality. Only the first call will affect the functionality.
2. Any party $\mathcal{P}_i$ uses the interface com.validate.$*$ to verify that $ccom$ contains the correct commitment parameters and verification algorithm.
3. Any party $\mathcal{P}_i$ uses the interface com.commit.$*$ to send a message cm and then obtain a commitment $ccom$ and an opening $copen$.
4. Any party $\mathcal{P}_i$ uses the interface com.verify.$*$ to verify that $ccom$ is a commitment to the message cm with the opening $copen$.

$\mathcal{F}_{\mathrm{NIC}}$ and $\mathcal{F}_{\mathrm{ENIC}}$ use a table Tbl_{com}. Tbl_{com} consists of entries of the form $[ccom, cm, copen, u]$, where $ccom$ is a commitment, cm is a message, $copen$ is an opening, and u is a bit whose value is 1 if the tuple $(ccom, cm, copen)$ is valid and 0 otherwise.

In the figure below, we depict $\mathcal{F}_{\mathrm{NIC}}$ and $\mathcal{F}_{\mathrm{ENIC}}$ and use a box to indicate those computations that take place only in $\mathcal{F}_{\mathrm{ENIC}}$.

Functionality $\mathcal{F}_{\mathrm{NIC}}$ and $\mathcal{F}_{\mathrm{ENIC}}$

$\mathcal{F}_{\mathrm{NIC}}$ and $\mathcal{F}_{\mathrm{ENIC}}$ are parameterized by system parameters sp. The following COM.TrapCom, COM.TrapOpen, $\boxed{\text{COM.Extract,}}$ and COM.Verify are ppt algorithms.

1. On input (com.setup.ini, sid) from a party $\mathcal{P}_i$:
   (a) If $(sid, cparcom,$ COM.TrapCom, COM.TrapOpen, $\boxed{\text{COM.Extract,}}$ COM.Verify, $ctdcom)$ is already stored, include $\mathcal{P}_i$ in the set $\mathbb{P}$, and send a delayed output (com.setup.end, sid, OK) to $\mathcal{P}_i$.
   (b) Otherwise proceed to generate a random $ssid$, store $(ssid, \mathcal{P}_i)$ and send (com.setup.req, sid, $ssid$) to $\mathcal{S}$.

S. On input (com.setup.alg, $sid, ssid, m$) from $\mathcal{S}$:
   (a) Abort if no pair $(ssid, \mathcal{P}_i)$ for some $\mathcal{P}_i$ is stored.
   (b) Delete record $(ssid, \mathcal{P}_i)$.
   (c) If $(sid, cparcom,$ COM.TrapCom, COM.TrapOpen, $\boxed{\text{COM.Extract,}}$ COM.Verify, $ctdcom)$ is already stored, include $\mathcal{P}_i$ in the set $\mathbb{P}$ and send (com.setup.end, sid, OK) to $\mathcal{P}_i$.
   (d) Otherwise proceed as follows.
      i. Parse m as $(cparcom,$ COM.TrapCom, COM.TrapOpen, $\boxed{\text{COM.Extract,}}$ COM.Verify, $ctdcom)$.
      ii. Store $(sid, cparcom,$ COM.TrapCom, COM.TrapOpen, $\boxed{\text{COM.Extract,}}$ COM.Verify, $ctdcom)$ and initialize both an empty table Tbl_{com} and an empty set $\mathbb{P}$.
      iii. Include $\mathcal{P}_i$ in the set $\mathbb{P}$ and send (com.setup.end, sid, OK) to $\mathcal{P}_i$.

2. On input (com.validate.ini, sid, $ccom$) from a party $\mathcal{P}_i$:
   (a) Abort if $\mathcal{P}_i \notin \mathbb{P}$.
   (b) Parse $ccom$ as $(ccom', cparcom', \text{COM.Verify}')$.
   (c) Set $v \leftarrow 1$ if $cparcom' = cparcom$ and $\text{COM.Verify}' = \text{COM.Verify}$. Otherwise, set $v \leftarrow 0$.
   (d) Send (com.validate.end, sid, v) to $\mathcal{P}_i$.

3. On input (com.commit.ini, sid, cm) from any honest party $\mathcal{P}_i$:
   (a) Abort if $\mathcal{P}_i \notin \mathbb{P}$ or if $cm \notin \mathcal{M}$, where $\mathcal{M}$ is defined in $cparcom$.
   (b) Compute $(ccom, cinfo) \leftarrow \mathsf{COM.TrapCom}(sid, cparcom, ctdcom)$.
   (c) Abort if there is an entry $[ccom, cm', copen', 1]$ in Tbl_{com} such that $cm \neq cm'$ in Tbl_{com}.
   (d) Run $copen \leftarrow \mathsf{COM.TrapOpen}(sid, cm, cinfo)$.
   (e) Abort if $1 \neq \mathsf{COM.Verify}(sid, cparcom, ccom, cm, copen)$.
   (f) Append $[ccom, cm, copen, 1]$ to Tbl_{com}.
   (g) Set $ccom \leftarrow (ccom, cparcom, \mathsf{COM.Verify})$.
   (h) Send (com.commit.end, sid, $ccom$, $copen$) to $\mathcal{P}_i$.
4. On input (com.verify.ini, sid, $ccom$, cm, $copen$) from any honest party $\mathcal{P}_i$:
   (a) Abort if $\mathcal{P}_i \notin \mathbb{P}$ or if $cm \notin \mathcal{M}$ or if $copen \notin \mathcal{R}$, where $\mathcal{M}$ and $\mathcal{R}$ are defined in $cparcom$.
   (b) Parse $ccom$ as $(ccom', cparcom', \mathsf{COM.Verify}')$. Abort if $cparcom' \neq cparcom$ or $\mathsf{COM.Verify}' \neq \mathsf{COM.Verify}$.
   (c) If there is an entry $[ccom', cm, copen, u]$ in Tbl_{com}, set $v \leftarrow u$.
   (d) Else, proceed as follows:
      i. If there is an entry $[ccom', cm', copen', 1]$ in Tbl_{com} such that $cm \neq cm'$, set $v \leftarrow 0$.
      ii. Else, proceed as follows:
         A. $\boxed{\text{If } cm \neq \mathsf{COM.Extract}(sid, ctdcom, ccom'), \text{ set } v \leftarrow 0. \text{ Else:}}$
         B. Set $v \leftarrow \mathsf{COM.Verify}(sid, cparcom, ccom', cm, copen)$.
         C. Append $[ccom', cm, copen, v]$ to Tbl_{com}.
   (e) Send (com.verify.end, sid, v) to $\mathcal{P}_i$.

We now discuss the four interfaces of the ideal functionalities $\mathcal{F}_{\mathrm{NIC}}$ and $\mathcal{F}_{\mathrm{ENIC}}$. We mention $\mathcal{F}_{\mathrm{ENIC}}$ only in those computations that are exclusive to $\mathcal{F}_{\mathrm{ENIC}}$.

1. The com.setup.ini message is sent by any party $\mathcal{P}_i$. If the functionality has not yet been initialized, it will trigger a com.setup.req message to ask the simulator $\mathcal{S}$ to send algorithms COM.TrapCom, COM.TrapOpen, $\boxed{\text{COM.Extract,}}$ and COM.Verify, the commitment parameters and the trapdoor. Once the simulator has provided the algorithms for the first time, $\mathcal{F}_{\mathrm{NIC}}$ stores the algorithms, the commitment parameters $cparcom$ and the trapdoor $ctdcom$, and then notifies $\mathcal{P}_i$ that initialization was successful. If the functionality has already been set up, $\mathcal{P}_i$ is just told that initialization was successful.
2. The com.validate.ini message is sent by an honest party $\mathcal{P}_i$. $\mathcal{F}_{\mathrm{NIC}}$ checks if $\mathcal{P}_i$ has already run the setup. This is needed because otherwise in the real-world protocol the party would have to retrieve the parameters to validate the commitment, and this retrieval cannot be simulated because $\mathcal{F}_{\mathrm{NIC}}$ enforces that the validation of a commitment must be local. The computation and verification of commitments are also local. $\mathcal{F}_{\mathrm{NIC}}$ parses the commitment, and checks if the parameters and the verification algorithm from the commitment match with those stored by the functionality.
3. The com.commit.ini message is sent by any honest party $\mathcal{P}_i$ on input a message cm. $\mathcal{F}_{\mathrm{NIC}}$ aborts if $\mathcal{P}_i$ did not run the setup. $\mathcal{F}_{\mathrm{NIC}}$ runs the algorithm COM.TrapCom on input $cparcom$ and $ctdcom$ to get a simulated commitment $ccom$ and state information

$cinfo$. $\mathsf{COM.TrapCom}$ does not receive the message cm to compute $ccom$, and therefore a commitment scheme that realizes this functionality must fulfill the hiding property. $\mathcal{F}_{\mathrm{NIC}}$ also aborts if the table Tbl_{com} already stores an entry $[ccom, cm', copen', 1]$ such that $cm \neq cm'$ because this would violate the binding property. $\mathcal{F}_{\mathrm{NIC}}$ runs the algorithm $\mathsf{COM.TrapOpen}$ on input cm and $cinfo$ to get an opening $copen$ and checks the validity of $(ccom, cm, copen)$ by running $\mathsf{COM.Verify}$. If $\mathsf{COM.Verify}$ outputs 1, $\mathcal{F}_{\mathrm{NIC}}$ stores $[ccom, cm, copen, 1]$ in Tbl_{com}, appends $(cparcom, \mathsf{COM.Verify})$ to $ccom$, and sends $(ccom, copen)$ to $\mathcal{P}_i$.

4. The $\mathsf{com.verify.ini}$ message is sent by any honest party $\mathcal{P}_i$ on input a commitment $ccom$, a message cm and an opening $copen$. $\mathcal{F}_{\mathrm{NIC}}$ aborts if $\mathcal{P}_i$ did not run the setup. If there is an entry $[ccom, cm, copen, u]$ already stored in Tbl_{com}, then the functionality returns the bit u. Therefore, a commitment scheme that realizes this functionality must be consistent. If there is an entry $[ccom, cm', copen', 1]$ such that $cm \neq cm'$, the functionality returns 0. Therefore, a scheme that realizes the functionality must fulfill the binding property. Else, in $\mathcal{F}_{\mathrm{ENIC}}$, the functionality checks whether the output of $\mathsf{COM.Extract}$ equals the message sent for verification and rejects the commitment if that is not the case. Then, the functionality runs the algorithm $\mathsf{COM.Verify}$ to verify $(ccom, cm, copen)$. The functionality records the result in Tbl_{com} and returns that result.

The functionality $\mathcal{F}_{\mathrm{NIC}}$ does not allow the computation and verification of commitments using any parameters $cparcom$ that were not generated by the functionality. As can be seen, the interfaces $\mathsf{com.commit.}*$ and $\mathsf{com.verify.}*$ use the commitment parameters that are stored by the functionality to compute and verify commitments. Therefore, a construction that realizes this functionality must ensure that the honest parties use the same commitment parameters. In general, such a "CRS-based" setup is required to realize UC commitments [8].

We note that we introduce the $\mathsf{com.validate.}*$ interface so that the parties can ensure that the commitment contains the right parameters and verification algorithm. This is needed especially for the parties that only receive a commitment value, without the opening. Otherwise, the $\mathsf{com.verify.}*$ interface can be called directly. Another way of doing this is to introduce an interface in the commitment functionality that returns the parameters and verification algorithm and require parties to call it first and compare the received parameters with the ones from the commitment.

3.2 Binding and hiding properties of $\mathcal{F}_{\mathrm{NIC}}$ and $\mathcal{F}_{\mathrm{ENIC}}$

Let us analyse the security properties of our two commitment functionalities. While inspection readily shows that both functionalities satisfy the standard binding and hiding properties, this merits some discussion.

We first note that both functionalities are perfectly hiding (because the commitment is computed independently of the message to be committed) and perfectly binding (the functionalities will accept only one value per commitment as committed value). Both properties being perfect seems like a contradiction, but it is not because the functionalities will only be *computationally* indistinguishable from their realizations. This implies of course that only computationally binding and hiding are enforced onto realizations.

Having said this, the binding property of $\mathcal{F}_{\mathrm{NIC}}$ merits further discussion, because, although it is guaranteed that adversarially computed commitments (outside $\mathcal{F}_{\mathrm{NIC}}$) can only be opened in one way, it is conceivable that an adversary could produce a commitment that it could open in two ways, and then, depending on its choice, provide one or the other opening, which would be allowed by $\mathcal{F}_{\mathrm{NIC}}$. This seems like a weaker property than what a traditional commitment scheme offers. There, after computing a commitment on input a message, that commitment can only be opened to that message. In this respect, we first remark that for traditional, perfectly hiding commitments, this might also be possible (unless one can extract more than one opening from an adversary, for instance, via rewinding). Second, we can show the following proposition, stating that for all realizations of $\mathcal{F}_{\mathrm{NIC}}$, no adversary is actually able to provide two different openings for adversarially generated commitments (the proof is provided in the full version of this paper).

Proposition 1. *For any construction Π_{NIC} that realizes $\mathcal{F}_{\mathrm{NIC}}$, there is no algorithm* $\mathsf{COM.Verify}$ *input by the simulator $\mathcal{S}_{\mathrm{NIC}}$ to $\mathcal{F}_{\mathrm{NIC}}$ such that, for any tuples $(ccom, cm, copen)$ and $(ccom, cm', copen')$ such that $cm \neq cm'$, $1 = \mathsf{COM.Verify}(sid, cparcom, ccom, cm, copen)$ and $1 = \mathsf{COM.Verify}(sid, cparcom, ccom, cm', copen')$.*

Let us finally note that the behaviour of $\mathcal{F}_{\mathrm{ENIC}}$ is different here, i.e., if the extraction algorithm is deterministic, it is guaranteed that there exists only one value to which a commitment can be opened.

3.3 Using $\mathcal{F}_{\mathrm{NIC}}$ in Conjunction with Other Functionalities

We turn to our main goal, namely how $\mathcal{F}_{\mathrm{NIC}}$ can be used to ensure that the same value is used as input to different functionalities or that an output from one functionality is used as an input to another functionality. We show the first case in detail with a toy example and then discuss the second case.

Ensuring Consistent Inputs. Let us consider the case where a construction requires that one party provides the same value to two (or more) different functionalities. To achieve this, the two functionalities need to get as input that value and also a commitment to that value and the corresponding opening value. It is further necessary that 1) also the other parties input the same commitment to the functionalities (or, alternatively, get the commitment from the functionalities and then check that they get the same commitment from them); 2) it is verified that the commitment is valid w.r.t. $\mathcal{F}_{\mathrm{NIC}}$, and that 3) the functionalities are able to somehow verify whether the value provided is indeed contained in the commitment. For the last item, it would seem natural that $\mathcal{F}_{\mathrm{NIC}}$ would be queried, but the UC framework does not allow that, and therefore we need to use a different mechanism: the commitments themselves contain a verification algorithm such that if the algorithm accepts an opening, then it is implied that $\mathcal{F}_{\mathrm{NIC}}$ would also accept the value and the opening for that commitment.

To enable this, let us start with two observations. In Proposition 1, we showed that $\mathsf{COM.Verify}$ will only accept one opening per *adversarially* computed commitment. However, this is not sufficient, because $\mathsf{COM.Verify}$ could accept different openings for commitments computed by $\mathcal{F}_{\mathrm{NIC}}$ because in that case $\mathcal{F}_{\mathrm{NIC}}$ does not invoke $\mathsf{COM.Verify}$ when processing requests to $\mathsf{com.verify.ini}$ and it is indeed conceivable that $\mathsf{COM.Verify}$ could behave differently.

However, for any secure realization Π_{NIC}, calls to the algorithm $\mathsf{com.verify.ini}$ of Π_{NIC} are indistinguishable from calls on the $\mathsf{com.verify.ini}$ interface to $\mathcal{F}_{\mathrm{NIC}} \| \mathcal{S}_{\Pi_{\mathrm{NIC}}}$, and $\mathsf{com.verify.ini}$ must be a non-interactive algorithm. Therefore, if $\mathcal{S}_{\Pi_{\mathrm{NIC}}}$ (i.e., the simulator such that $\mathcal{F}_{\mathrm{NIC}} \| \mathcal{S}_{\Pi_{\mathrm{NIC}}}$ is indistinguishable from Π_{NIC}) provides the real-world algorithm $\mathsf{com.verify.ini}$ of Π_{NIC} as $\mathsf{COM.Verify}()$ algorithm to $\mathcal{F}_{\mathrm{NIC}}$, then calling $\mathsf{COM.Verify}()$ in another functionality to verify an opening and committed message w.r.t. a commitment will necessarily produce the same result as a call to the $\mathsf{com.verify.ini}$ interface to $\mathcal{F}_{\mathrm{NIC}} \| \mathcal{S}_{\Pi_{\mathrm{NIC}}}$. We will use the latter in an essential way when composing different functionalities, as we will illustrate with an example in the following.

We note that the assumption that $\mathcal{S}_{\Pi_{\mathrm{NIC}}}$ provides the algorithms to $\mathcal{F}_{\mathrm{NIC}} \| \mathcal{S}_{\Pi_{\mathrm{NIC}}}$ that are used in the real world is natural and not a serious restriction. After all, the purpose of defining a functionality using cryptographic algorithms is that the functionality specifies the behavior of the real algorithms, especially those that are used to verify cryptographic values. Assuming that the calls $\mathsf{com.commit.ini}$ and $\mathsf{com.verify.ini}$ to Π_{NIC} are local to the calling parties is also natural as this is how traditional commitment schemes are realized.

Furthermore, we note that $\mathcal{F}_{\mathrm{NIC}}$ restricts $\mathcal{S}_{\Pi_{\mathrm{NIC}}}$ to send the real-world verification algorithm as $\mathsf{COM.Verify}$. The reason is that $\mathcal{F}_{\mathrm{NIC}}$ outputs $\mathsf{COM.Verify}$ inside $ccom$ through the $(\mathsf{com.commit.end}, sid, ccom, copen)$ message. In the real world, any construction for $\mathcal{F}_{\mathrm{NIC}}$ outputs the real-world verification algorithm through the $(\mathsf{com.commit.end}, sid, ccom, copen)$ message. Therefore, because the outputs in the real-word and in the ideal-world must be indistinguishable, any simulator must input the real-world verification algorithm as $\mathsf{COM.Verify}$ to $\mathcal{F}_{\mathrm{NIC}}$. Otherwise the message $\mathsf{com.commit.end}$ in the ideal world can be distinguished from that in the real world by the environment.

We are now ready to show how our goal can be achieved using a toy example. To this end, let us define three two party functionalities $\mathcal{F}_1$, $\mathcal{F}_2$, and $\mathcal{F}_{(1,2)}$. The first two $\mathcal{F}_1$ and $\mathcal{F}_2$ compute the function $f_1(\cdot)$ and $f_2(\cdot)$, respectively, on P_1's input and send the result to P_2. Analogously, $\mathcal{F}_{(1,2)}$ computes $(f_1(\cdot), f_2(\cdot))$ on P_1's input and sends the result to P_2. Our goal is now to realize $\mathcal{F}_{(1,2)}$ by a hybrid protocol $\Pi_{(1,2)}$ using $\mathcal{F}_1$ and $\mathcal{F}_2$ to compute $f_1(\cdot)$ and $f_2(\cdot)$, respectively, and $\mathcal{F}_{\mathrm{NIC}}$ to ensure that the inputs to both $\mathcal{F}_1$ and $\mathcal{F}_2$ are the same. To achieve this, $\mathcal{F}_1$ and $\mathcal{F}_2$ will take as inputs also commitments and do some basic checks on them. These functionalities and construction $\Pi_{(1,2)}$ are as follows.

Functionality $\mathcal{F}_i$

1. On input $(\mathsf{f_i.in.ini}, sid, a, ccom_1, copen)$ from a party P_1, check if $sid = (P_1, P_2, sid')$ for some P_2 and sid', and no record is stored. If so, record $(a, ccom_1, copen)$ and send $(\mathsf{f_i.in.end}, sid)$ to P_1, otherwise $(\mathsf{f_i.in.end}, sid, \perp)$ to P_1.
2. On input $(\mathsf{f_i.eval.ini}, sid, ccom_2)$ from P_2, check if $sid = (P_1, P_2, sid')$ for some P_1 and sid', if a record $(a, ccom_1, copen)$ is stored, and if $ccom_1 = ccom_2$ and

$\mathsf{COM.Verify}(sid, cparcom, ccom_1, a, copen) = 1$ holds. If so, send delayed $(\mathsf{f_i.eval.end}, sid, f_i(a))$ to P_2. Otherwise send delayed $(\mathsf{f_i.eval.end}, sid, \perp)$ to P_2.

Functionality $\mathcal{F}_{(1,2)}$

1. On input $(\mathsf{f_{12}.eval.ini}, sid, a)$ from a party P_1, check if $sid = (P_1, P_2, sid')$. If so, send delayed $(\mathsf{f_{12}.eval.end}, sid, (f_1(a), f_2(a)))$ to P_2 and otherwise send delayed $(\mathsf{f_{12}.eval.end}, sid, \perp)$ to P_1.

Construction $\Pi_{(1,2)}$

1. On input $(\mathsf{f_{12}.eval.ini}, sid, a)$, P_1 proceeds as follows.
   (a) i. P_1 checks if $sid = (P_1, P_2, sid')$.
   ii. P_1 calls $\mathcal{F}_{\mathrm{NIC}}$ with $(\mathsf{com.setup.ini}, sid)$ and receives $(\mathsf{com.setup.end}, sid, OK)$.
   iii. P_1 calls $\mathcal{F}_{\mathrm{NIC}}$ with $(\mathsf{com.commit.ini}, sid, a)$ to receive $(\mathsf{com.commit.end}, sid, ccom, copen)$.
   iv. P_1 calls $\mathcal{F}_1$ with $(\mathsf{f_1.in.ini}, sid, a, ccom, copen)$ and receives $(\mathsf{f_1.in.end}, sid)$.
   v. P_1 calls $\mathcal{F}_2$ with $(\mathsf{f_2.in.ini}, sid, a, ccom, copen)$ and receives $(\mathsf{f_2.in.end}, sid)$.
   vi. P_1 sends $(\mathsf{smt.send.ini}, sid, ccom)$ to P_2 using $\mathcal{F}_{\mathrm{SMT}}$.
   (b) Upon receiving $(\mathsf{smt.send.end}, sid, ccom)$ from P_1 via $\mathcal{F}_{\mathrm{SMT}}$, P_2 proceeds as follows.
   i. P_2 checks if $sid = (P_1, P_2, sid')$.
   ii. P_2 calls $\mathcal{F}_{\mathrm{NIC}}$ with $(\mathsf{com.setup.ini}, sid)$ and receives $(\mathsf{com.setup.end}, sid, OK)$.
   iii. P_2 calls $\mathcal{F}_{\mathrm{NIC}}$ with $(\mathsf{com.validate.ini}, sid, ccom)$.
   iv. P_2 calls $\mathcal{F}_1$ with $(\mathsf{f_1.eval.ini}, sid, ccom)$ and receives $(\mathsf{f_1.eval.end}, sid, f_1(a))$.
   v. P_2 calls $\mathcal{F}_2$ with $(\mathsf{f_2.eval.ini}, sid, ccom)$ and receives $(\mathsf{f_2.eval.end}, sid, f_2(a))$.
   vi. P_2 outputs $(\mathsf{f_{12}.eval.end}, sid, (f_1(a), f_2(a)))$.

If at any step a party receives a wrong message from a functionality or some check fails, it outputs $(\mathsf{f_{12}.eval.end}, sid, \perp)$.

We next show that $\Pi_{(1,2)}$ realizes $\mathcal{F}_{(1,2)}$ and thereby give an example of a security proof that uses $\mathcal{F}_{\mathrm{NIC}}$ and does not need to reduce to property-based security definitions of a commitment scheme. Note that although formally we consider a $\mathcal{F}_{\mathrm{NIC}} \| \mathcal{S}_{\Pi_{\mathrm{NIC}}}$-hybrid protocol, our example protocol $\Pi_{(1,2)}$ uses $\mathcal{F}_{\mathrm{NIC}}$ in the same way as any other functionality, i.e., without having to consider the simulator $\mathcal{S}_{\Pi_{\mathrm{NIC}}}$ for some realization Π_{NIC} of $\mathcal{F}_{\mathrm{NIC}}$.

Theorem 1. *Assume that $\mathcal{F}_{\mathrm{NIC}} \| \mathcal{S}_{\Pi_{\mathrm{NIC}}}$ is indistinguishable from Π_{NIC} and that $\mathcal{S}_{\Pi_{\mathrm{NIC}}}$ provides Π_{NIC}'s verification algorithm as* $\mathsf{COM.Verify}()$ *to $\mathcal{F}_{\mathrm{NIC}}$. Then $\Pi_{(1,2)}$ realizes $\mathcal{F}_{(1,2)}$ in the $(\mathcal{F}_{\mathrm{SMT}}, \mathcal{F}_1, \mathcal{F}_2, \mathcal{F}_{\mathrm{NIC}} \| \mathcal{S}_{\Pi_{\mathrm{NIC}}})$-hybrid model. $\mathcal{F}_{\mathrm{SMT}}$ [6] is described in the full version.*

Proof. We provide a simulator $\mathcal{S}_{\Pi_{(1,2)}}$ and prove that $\mathcal{F}_{(1,2)} \| \mathcal{S}_{\Pi_{(1,2)}}$ is indistinguishable from $\Pi_{(1,2)}$ if there exists a Π_{NIC} that realizes $\mathcal{F}_{\mathrm{NIC}}$.

We consider four cases, depending on which party is corrupt. In case both P_1 and P_2 are corrupt, there is nothing to simulate. In case both parties are honest, the simulator will be asked by $\mathcal{F}_{(1,2)}$ to send $(\mathsf{f}_{12}.\mathsf{eval.end}, sid, (f_1(a), f_2(a)))$ to P_2 and then proceed as follows. First it initializes $\mathcal{F}_{\mathrm{NIC}}$. It then picks a random value a' and executes $\Pi_{(1,2)}$ as P_1 and P_2 using a' as the input of P_1 and running $\mathcal{F}_{\mathrm{SMT}}$, $\mathcal{F}_1$, $\mathcal{F}_2$, and $\mathcal{F}_{\mathrm{NIC}} \| \mathcal{S}_{\Pi_{\mathrm{NIC}}}$ as they are specified, with exception that when $\mathcal{F}_1$ and $\mathcal{F}_2$ would output $f_1(a')$ and $f_2(a')$, respectively, to P_2, the simulator instead make these two functionalities output $f_1(a)$ and $f_2(a)$, respectively (which are the values $\mathcal{S}_{\Pi_{(1,2)}}$ had obtained earlier from $\mathcal{F}_{(1,2)}$). If this protocol execution is successful, $\mathcal{S}_{\Pi_{(1,2)}}$ will let the delayed output $(\mathsf{f}_{12}.\mathsf{eval.end}, sid, (f_1(a), f_2(a)))$ to P_2 pass. Otherwise it will drop it, as it will have already sent $(\mathsf{f}_{12}.\mathsf{eval.end}, sid, \perp)$ to P_2 or P_1 according to the protocol specification. It is not hard to see that this simulation will cause the same distribution on the values sent to the adversary as the real protocol. The only difference is that the simulator uses a different input value for P_1 and the only other value that depends in a' is $copen$ (by the specification of $\mathcal{F}_{\mathrm{NIC}}$). As the environment/adversary never sees any of these two values or any value that depends on it (which is seen by inspection of all simulated functionalities and because $f_1(a')$ and $f_2(a')$ are replaced by $f_1(a)$ and $f_2(a)$ in the outputs of $\mathcal{F}_1$ and $\mathcal{F}_2$), the argument follows.

As next case, assume that P_1 is honest and P_2 is corrupt. This case is similar to the one where both are honest. The simulator proceeds the same way only that it will not execute the steps of P_2 and it will allow the delivery of the message $(\mathsf{f}_{12}.\mathsf{eval.end}, sid, (f_1(a), f_2(a)))$ to P_2. The argument that the simulation is successful remains essentially the same. Here, the environment will additionally see $ccom$ which, as said before, does not depend on a'.

As last case, assume that P_2 is honest and P_1 is corrupt. Thus, $\mathcal{S}_{\Pi_{(1,2)}}$ interacts with the adversarial P_1 and the environment/adversary, simulating $\Pi_{1,2}$ towards P_1 and the functionalities $\mathcal{F}_{\mathrm{SMT}}$, $\mathcal{F}_1$, $\mathcal{F}_2$, and $\mathcal{F}_{\mathrm{NIC}} \| \mathcal{S}_{\Pi_{\mathrm{NIC}}}$ towards both the environment and P_1, and finally P_1 towards $\mathcal{F}_{1,2}$. Simulation is straightforward: the simulator just runs everything as specified, learning P_1's input a from P_1's input to $\mathcal{F}_{\mathrm{NIC}} \| \mathcal{S}_{\Pi_{\mathrm{NIC}}}$, $\mathcal{F}_1$, and $\mathcal{F}_2$. If this simulation reaches Step 1(b)vi, $\mathcal{S}_{\Pi_{(1,2)}}$ will input that a to $\mathcal{F}_{(1,2)}$ as P_1, causing it to send a delayed output $(\mathsf{f}_{12}.\mathsf{eval.end}, sid, (f_1(a), f_2(a)))$ to P_2 for $\mathcal{S}_{\Pi_{(1,2)}}$ to deliver, which it will do. This simulation will be correct, as long as P_1 cannot cause $\mathcal{F}_1$ and $\mathcal{F}_2$ to send a result for a different input value. However, this cannot happen because if both functionalities accept P_1's input, the committed value must be identical thanks to the properties of $\mathsf{COM.Verify}$ (cf. discussion above). $\square$

Comparison with a Construction that Used a Standard Commitment Scheme. One could of course also realize $\mathcal{F}_{(1,2)}$ with a construction that uses a standard commitment scheme, i.e., one defined by property-based security definitions, instead of $\mathcal{F}_{\mathrm{NIC}}$. The resulting construction and the security proof would be less modular, comparable to a construction that uses a standard signature scheme instead of $\mathcal{F}_{SIG}$. For the security proof, the overall strategy would be rather similar, the main difference being that one would have to do reductions to the properties of the commitment scheme, i.e., additional game hops. That is, one would have to show that the binding property does not hold if an adversarial P_1

manages to send different inputs to $\mathcal{F}_1$ and $\mathcal{F}_2$. Also, one would have to show that the hiding property does not hold if an adversarial P_2 is able to distinguish between the real protocols and the simulator that interacts with the functionality $\mathcal{F}_{(1,2)}$ and thus has to send P_2 a commitment to a different value.

Ensuring an Output is used as an Input. Let us consider a two-party construction that requires that an output from one functionality be used as an input to another functionality. This can be achieved in different ways, the simplest way seems to be that the first party, upon obtaining its output from the first functionality, calls $\mathcal{F}_{\mathrm{NIC}}$ to obtain a commitment on that value and an opening and sends the commitment and the opening to the first functionality. The first functionality will then check whether the commitment indeed contains the output and, if so, will send the commitment to the second party who can then use that commitment as input to the second functionality. We leave the details of this to the reader.

3.4 Construction of UC Non-Interactive Commitments

We now provide our construction for UC non-interactive commitments. It uses a commitment scheme (CSetup, Com, VfCom) that fulfils the binding and trapdoor properties [13].

Our construction works in the $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CSetup}}$-hybrid model, where parties use the ideal functionality $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CSetup}}$ that is parameterized by the algorithm CSetup, which takes as input the system parameters sp.

Construction Π_{NIC}^{sp}

Construction Π_{NIC} is parameterized by system parameters sp, and uses the ideal functionality $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CSetup},sp}$ and a commitment scheme (CSetup, Com, VfCom).

```
1. On input $(\mathsf{com.setup.ini}, sid)$, a party $\mathcal{P}$ executes the following program:
   (a) Send $(\mathsf{crs.setup.ini}, sid)$ to $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CSetup},sp}$ to receive $(\mathsf{crs.setup.end}, sid, par_c)$.
   (b) Store $(par_c, \mathsf{VfCom})$ and output $(\mathsf{com.setup.end}, sid, OK)$.
2. On input $(\mathsf{com.validate.ini}, sid, ccom)$, a party $\mathcal{P}$ executes the following program:
   (a) If $(par_c, \mathsf{VfCom})$ is not stored, abort.
   (b) Parse $ccom$ as $(ccom', par_c', \mathsf{VfCom}')$.
   (c) Set $v \leftarrow 1$ if $par_c' = par_c$ and $\mathsf{VfCom}' = \mathsf{VfCom}$. Otherwise, set $v \leftarrow 0$.
   (d) Output $(\mathsf{com.validate.end}, sid, v)$.
3. On input $(\mathsf{com.commit.ini}, sid, cm)$, a party $\mathcal{P}$ executes the following program:
   (a) If $(par_c, \mathsf{VfCom})$ is not stored, abort.
   (b) Abort if $cm \notin \mathcal{M}$, where $\mathcal{M}$ is defined in $par_c$.
   (c) Run $(com, open) \leftarrow \mathsf{Com}(par_c, cm)$.
   (d) Output $(\mathsf{com.commit.end}, sid, ccom \leftarrow (com, par_c, \mathsf{VfCom}), open)$.
4. On input $(\mathsf{com.verify.ini}, sid, ccom, cm, copen)$, $\mathcal{P}$ executes the following program:
   (a) If $(par_c, \mathsf{VfCom})$ is not stored, abort.
   (b) Abort if $cm \notin \mathcal{M}$ or if $copen \notin \mathcal{R}$, where $\mathcal{M}$ and $\mathcal{R}$ are defined in $par_c$.
   (c) Parse $ccom$ as $(ccom', par_c', \mathsf{VfCom}')$.
   (d) If $par_c' = par_c$ and $\mathsf{VfCom}' = \mathsf{VfCom}$ then run $v \leftarrow \mathsf{VfCom}(par_c, ccom', cm,$
       $copen)$. Otherwise, set $v \leftarrow 0$.
   (e) Output $(\mathsf{com.verify.end}, sid, v)$.
```

Theorem 2. *The construction Π_{NIC} realizes $\mathcal{F}_{\mathrm{NIC}}$ in the $\mathcal{F}_{\mathrm{CRS}}^{\mathsf{CSetup}}$-hybrid model if the underlying commitment scheme (*CSetup, Com, VfCom*) is binding and trapdoor.*

We provide the proof in the full version of this paper.

4 The Ideal Functionalities $\mathcal{F}_{\mathbf{REV}}$ and $\mathcal{F}_{\mathbf{AT}}$

We describe our ideal functionality for non-hiding and hiding revocation, $\mathcal{F}_{\mathrm{REV}}$, in Section 4.1. Our constructions for non-hiding and hiding revocation and their security analysis can be found in the full version of this paper. The construction for non-hiding revocation uses a non-hiding vector commitment scheme, whereas the hiding construction employs a trapdoor vector commitment scheme. In the full version of this paper we also define the trapdoor property for vector commitments and propose a construction for non-hiding and trapdoor vector commitments.

We describe our ideal functionality for attribute tokens, $\mathcal{F}_{\mathrm{AT}}$, in Section 4.2. We provide the construction and prove it secure in the full version of this paper.

4.1 Ideal Functionality for Revocation $\mathcal{F}_{\mathbf{REV}}$

Here we describe our ideal functionality $\mathcal{F}_{\mathrm{REV}}$ for revocation. $\mathcal{F}_{\mathrm{REV}}$ interacts with a revocation authority $\mathcal{RA}$, users $\mathcal{U}$ and any verifying parties $\mathcal{P}$. The revocation authority $\mathcal{RA}$ associates a revocation status $\mathbf{x}[rh]$ with every revocation handle rh. A revocation status consists of m bits, such that each bit $\mathbf{x}[rh, j]$ denotes the revocation status of the revocation handle rh with respect to the revocation list $j \in [1, m]$. The time is divided into epochs ep, and the revocation authority $\mathcal{RA}$ can change the revocation status of every revocation handle at the beginning of each epoch.

A user $\mathcal{U}$ can obtain a proof pr that the revocation status of the revocation handle rh committed in a commitment $ccom$ is $\mathbf{x}[rh, j]$ for the list j at the epoch ep. Given pr, $ccom$, $\mathbf{x}[rh, j]$, j, and ep, any party $\mathcal{P}$ can verify the proof pr.

$\mathcal{F}_{\mathrm{REV}}$ describes two ideal functionalities: a *hiding* revocation functionality where, if the revocation authority is honest, the revocation status of a revocation handle is only revealed to the user associated with that revocation handle, and a *non-hiding* revocation functionality where the revocation statuses of all revocation handles are public. We provide a unified description of both ideal functionalities. The box $\boxed{\text{H: } \ldots}$ is used to describe something that occurs only in the hiding revocation functionality, whereas the box $\boxed{\text{NH: } \ldots}$ is used in the same way for the non-hiding revocation functionality.

$\mathcal{F}_{\mathrm{REV}}$ interacts with the revocation authority $\mathcal{RA}$, the users $\mathcal{U}$ and any verifying parties $\mathcal{P}$ through the interfaces $\mathsf{rev.setup.}*$, $\mathsf{rev.get.}*$, $\mathsf{rev.epoch.}*$, $\mathsf{rev.getepoch.}*$, $\mathsf{rev.getstatus.}*$, $\mathsf{rev.prove.}*$, and $\mathsf{rev.verify.}*$.

1. The revocation authority $\mathcal{RA}$ uses the $\mathsf{rev.setup.}*$ interface to receive the revocation parameters par_r.
2. Any party $\mathcal{P}$ invokes the $\mathsf{rev.get.}*$ interface to receive par_r.
3. The revocation authority $\mathcal{RA}$ uses the $\mathsf{rev.epoch.}*$ interface to send a list $\langle \boxed{\text{H: } \mathcal{U}_i,}\ rh_i, \mathbf{x}[rh_i] \rangle_{i=1}^{n'}$ of revocation handles and revocation statuses for the epoch ep and receive the epoch information $info$ for the epoch ep.

4. Any party $\mathcal{P}$ uses the rev.getepoch.$*$ interface to get the epoch information $info$ for the epoch ep. In the non-hiding functionality, $\mathcal{P}$ also obtains the full list of revocation handles and revocation statuses $\langle rh_i, \mathbf{x}[rh_i]\rangle_{i=1}^{n'}$ for the epoch ep.
5. In the hiding revocation functionality, a user $\mathcal{U}$ with a revocation handle rh uses the rev.getstatus.$*$ interface to receive the revocation status $\mathbf{x}[rh]$ at a given epoch ep.
6. An honest user $\mathcal{U}$ uses the rev.prove.$*$ interface to obtain a proof pr that the revocation status of the revocation handle rh committed in a commitment $ccom$ is $\mathbf{x}[rh, j]$ for the list j at the epoch ep.
7. Any honest party $\mathcal{P}$ uses the rev.verify.$*$ interface to verify a proof pr on input $ccom$, $\mathbf{x}[rh, j]$, j and ep.

Functionality $\mathcal{F}_{\mathrm{REV}}$

REV.SimProve and REV.Extract are ppt algorithms. $\mathcal{F}_{\mathrm{REV}}$ is parameterized by system parameters sp, by a maximum number of revocation lists m, and a maximum number of revocation handles n.

1. On input (rev.setup.ini, sid) from $\mathcal{RA}$:
   (a) Abort if $sid \neq (\mathcal{RA}, sid')$ or if (sid) is already stored.
   (b) Store (sid).
   (c) Send (rev.setup.req, sid) to $\mathcal{S}$.

S. On input (rev.setup.alg, sid, par_r, td_r, REV.SimProve, REV.Extract) from $\mathcal{S}$:
   (a) Abort if (sid) is not stored or if (sid, par_r, td_r, REV.SimProve, REV.Extract) is already stored.
   (b) Store (sid, par_r, td_r, REV.SimProve, REV.Extract).
   (c) Initialize an empty table Tbl_{pr} and an empty set $\mathbb{P}$.
   (d) Send (rev.setup.end, sid, par_r) to $\mathcal{RA}$.

2. On input (rev.get.ini, sid) from a party $\mathcal{P}$:
   (a) If (sid, par_r, td_r, REV.SimProve, REV.Extract) is stored, set $par_r' \leftarrow par_r$; else $par_r' \leftarrow \perp$.
   (b) Create a fresh $ssid$ and store $(ssid, \mathcal{P}, par_r')$.
   (c) Send (rev.get.sim, $sid, ssid, par_r'$) to $\mathcal{S}$.

S. On input (rev.get.rep, $sid, ssid$) from $\mathcal{S}$:
   (a) Abort if $(ssid, \mathcal{P}, par_r')$ is not stored.
   (b) If $par_r' \neq \perp$, include $\mathcal{P}$ in the set $\mathbb{P}$.
   (c) Replace $(ssid, \mathcal{P}, par_r')$ with$(\mathcal{P}, par_r')$.
   (d) Send (rev.get.end, sid, par_r') to $\mathcal{P}$.

3. On input (rev.epoch.ini, $sid, ep, \langle \boxed{\text{H: } \mathcal{U}_i,} rh_i, \mathbf{x}[rh_i]\rangle_{i=1}^{n'}$) from $\mathcal{RA}$:
   (a) Abort if record (sid, par_r, td_r, REV.SimProve, REV.Extract) is not stored, or if $(sid, ep, \langle \boxed{\text{H: } \mathcal{U}_i',} rh_i', \mathbf{x}[rh_i]'\rangle_{i=1}^{n'})$ is already stored, or if $n' > n$, or if, for $i = 1$ to n', $rh_i \notin [1, n]$ or $\mathbf{x}[rh_i] \notin [0, 2^m)$ $\boxed{\text{H: or } \mathcal{U}_i \text{ is not a valid user identifier}}$.
   (b) Store $(sid, ep, \langle \boxed{\text{H: } \mathcal{U}_i,} rh_i, \mathbf{x}[rh_i]\rangle_{i=1}^{n'})$.
   (c) Send (rev.epoch.sim, $sid, ep \boxed{\text{NH:}, \langle rh_i, \mathbf{x}[rh_i]\rangle_{i=1}^{n'}}$) to $\mathcal{S}$.

S. On input (rev.epoch.rep, $sid, ep, info$) from $\mathcal{S}$:
   (a) Abort if record $(sid, ep, \langle \boxed{\text{H: } \mathcal{U}_i,} rh_i, \mathbf{x}[rh_i]\rangle_{i=1}^{n'})$ is not stored or if $(sid, ep, info'$, $\mathsf{Tbl}_{ep})$ is already stored.

(b) Set $\mathsf{Tbl}_{ep} \leftarrow \langle \boxed{\text{H: } \mathcal{U}_i,}\, rh_i, \mathbf{x}[rh_i] \rangle_{i=1}^{n'}$ and initialize a set $\mathbb{E}_{ep}$.

(c) $\boxed{\text{H: Initialize a set } \mathbb{S}_{ep}\text{. If } \mathcal{RA} \text{ is corrupt, initialize an empty table } \mathsf{Tbl}_{ep}.}$

(d) Store $(sid, ep, info, \mathsf{Tbl}_{ep})$.

(e) Send $(\mathsf{rev.epoch.end}, sid, ep, info)$ to $\mathcal{RA}$.

4. On input $(\mathsf{rev.getepoch.ini}, sid, ep)$ from any party $\mathcal{P}$:

(a) Abort if $\mathcal{P} \nsubseteq \mathbb{P}$.

(b) If $(sid, ep, info, \mathsf{Tbl}_{ep})$ is stored, set $info' \leftarrow info$ $\boxed{\text{NH: and } \mathsf{Tbl}'_{ep} \leftarrow \mathsf{Tbl}_{ep}}$; else set $info' \leftarrow \bot$ $\boxed{\text{NH: and } \mathsf{Tbl}'_{ep} \leftarrow \bot}$.

(c) Create a fresh $ssid$ and store $(ssid, \mathcal{P}, ep, info' \boxed{\text{NH:, } \mathsf{Tbl}'_{ep}})$.

(d) Send $(\mathsf{rev.getepoch.sim}, sid, ssid, ep, info' \boxed{\text{NH:, } \mathsf{Tbl}'_{ep}})$ to $\mathcal{S}$.

S. On input $(\mathsf{rev.getepoch.rep}, sid, ssid)$ from $\mathcal{S}$:

(a) Abort if record $(ssid, \mathcal{P}, ep, info' \boxed{\text{NH:, } \mathsf{Tbl}'_{ep}})$ is not stored.

(b) If $info' \neq \bot$ $\boxed{\text{NH: and } \mathsf{Tbl}'_{ep} \neq \bot}$, include $\mathcal{P}$ in $\mathbb{E}_{ep}$.

(c) Delete record $(ssid, \mathcal{P}, ep, info' \boxed{\text{NH:, } \mathsf{Tbl}'_{ep}})$.

(d) Send $(\mathsf{rev.getepoch.end}, sid, info' \boxed{\text{NH:, } \mathsf{Tbl}'_{ep}})$ to $\mathcal{P}$.

5. H: On input $(\mathsf{rev.getstatus.ini}, sid, rh_i, ep)$ from a user $\mathcal{U}_i$:

(a) Abort if $\mathcal{U}_i \nsubseteq \mathbb{E}_{ep}$.

(b) If $\mathcal{RA}$ is honest, abort if there is no entry $[\mathcal{U}'_i, rh'_i, \mathbf{x}'[rh_i]]$ in Tbl_{ep} such that $rh'_i = rh_i$ and $\mathcal{U}'_i = \mathcal{U}_i$.

(c) Create a fresh $ssid$ and store $(ssid, \mathcal{U}_i, rh_i, ep)$.

(d) If $\mathcal{RA}$ is corrupt, send $(\mathsf{rev.getstatus.sim}, sid, ssid, \mathcal{U}_i, rh_i, ep)$ to $\mathcal{S}$; else send $(\mathsf{rev.getstatus.sim}, sid, ssid, \mathcal{U}_i, ep)$ to $\mathcal{S}$.

S. H: On input $(\mathsf{rev.getstatus.rep}, sid, ssid, \mathbf{x}[rh_i])$, if $\mathcal{RA}$ is corrupt, or $(\mathsf{rev.getstatus.rep}, sid, ssid)$, if $\mathcal{RA}$ is honest, from $\mathcal{S}$:

(a) Abort if record $(ssid, \mathcal{U}_i, rh_i, ep)$ is not stored.

(b) Delete record $(ssid, \mathcal{U}_i, rh_i, ep)$.

(c) If $\mathcal{RA}$ is corrupt, $\mathbf{x}[rh_i] \in [0, 2^m)$ and there is no entry $[\bot, rh'_i, \mathbf{x}'[rh_i]]$ in Tbl_{ep} such that $rh'_i = rh_i$, store $[\bot, rh_i, \mathbf{x}[rh_i]]$ in Tbl_{ep}.

(d) If $\mathcal{RA}$ is corrupt and $\mathbf{x}[rh_i] \neq \mathbf{x}'[rh_i]$, where $\mathbf{x}'[rh_i]$ is in the entry $[\bot, rh_i, \mathbf{x}'[rh_i]]$ in Tbl_{ep}, send $(\mathsf{rev.getstatus.end}, sid, \bot)$ to $\mathcal{U}_i$; else include $\mathcal{U}_i$ in the set $\mathbb{S}_{ep}$ and send $(\mathsf{rev.getstatus.end}, sid, \mathbf{x}'[rh_i])$ to $\mathcal{U}_i$.

6. On input $(\mathsf{rev.prove.ini}, sid, ccom, rh, copen, ep, j)$ from an honest user $\mathcal{U}_i$:

(a) Parse $ccom$ as $(ccom', cparcom, \mathsf{COM.Verify})$.

(b) Abort if $\mathcal{U}_i \nsubseteq \mathbb{E}_{ep}$, or if $1 \neq \mathsf{COM.Verify}(cparcom, ccom', rh, copen)$, or if there is no entry $[\boxed{\text{H: } \mathcal{U}'_i,}\, rh'_i, \mathbf{x}[rh_i]]$ in Tbl_{ep} such that $rh'_i = rh_i$.

(c) H: Abort if $\mathcal{U}_i \nsubseteq \mathbb{S}_{ep}$, or if $\mathcal{RA}$ is honest and there is no entry $[\mathcal{U}'_i, rh'_i, \mathbf{x}[rh_i]]$ in Tbl_{ep} such that $\mathcal{U}_i = \mathcal{U}'_i$ and $rh'_i = rh_i$.

(d) Run $pr \leftarrow \mathsf{REV.SimProve}(sid, par_r, cparcom, ccom', ep, info, j, \mathbf{x}[rh, j], td_r)$.
(e) Append $[\langle ccom, ep, j, \mathbf{x}[rh, j]\rangle, pr, 1]$ to Table Tbl_{pr}.
(f) Send $(\mathsf{rev.prove.end}, sid, pr)$ to $\mathcal{U}_i$.

7. On input $(\mathsf{rev.verify.ini}, sid, ccom, ep, j, b, pr)$ from an honest party $\mathcal{P}$:
   (a) Abort if $\mathcal{P} \not\subseteq \mathbb{E}_{ep}$.
   (b) Parse $ccom$ as $(ccom', cparcom, \mathsf{COM.Verify})$.
   (c) If there is an entry $[\langle ccom, ep, j, b\rangle, pr, u]$ in Tbl_{pr}, set $v \leftarrow u$.
   (d) Else, do the following:
      i. Extract $(rh, open, \mathbf{x}[rh]) \leftarrow \mathsf{REV.Extract}(sid, par_r, cparcom, ccom', ep, info, j, b, td_r, pr)$.
      ii. If $(rh, copen, \mathbf{x}[rh]) = \bot$ or $1 \neq \mathsf{COM.Verify}(cparcom, ccom', copen, rh)$, set $v \leftarrow 0$.
      iii. Else, do the following:

$\boxed{\text{A. H: If } \mathcal{RA} \text{ is corrupt and there is no entry } [\bot, rh', \mathbf{x}[rh]] \text{ in } \mathsf{Tbl}_{ep} \text{ such that } rh' = rh, \text{ store } [\bot, rh, \mathbf{x}[rh]].}$

         A. If $b = \mathbf{x}[rh, j]$, where $\mathbf{x}[rh, j]$ is in the entry $[\boxed{\text{H:}\ \mathcal{U}_i,}\ rh, \mathbf{x}[rh]] \in \mathsf{Tbl}_{ep}$ $\boxed{\text{H:, where, if } \mathcal{RA} \text{ is honest, } \mathcal{U}_i \text{ must be corrupt}}$, set $v \leftarrow 1$; else set $v \leftarrow 0$.
      iv. Append $[\langle ccom, ep, j, \mathbf{x}[rh, j]\rangle, pr, v]$ to Table Tbl_{pr}.
   (e) Send $(\mathsf{rev.verify.end}, sid, v)$ to $\mathcal{P}$.

$\mathcal{F}_{\mathrm{REV}}$ uses the following tables:

Tbl_{ep}. For the epoch ep, Tbl_{ep} stores entries of the form $[\boxed{\text{H:}\ \mathcal{U},}\ rh, \mathbf{x}[rh]]$ that associate the revocation handle rh with the revocation status $\mathbf{x}[rh]$. In the hiding functionality, if $\mathcal{RA}$ is honest, a user $\mathcal{U}$ is also associated with rh.

Tbl_{pr}. Tbl_{pr} stores entries of the form $[\langle ccom, ep, j, \mathbf{x}[rh, j]\rangle, pr, u]$, where $\langle ccom, ep, j, \mathbf{x}[rh, j]\rangle$ is part of the instance of a proof, pr is the proof, and u is a bit that indicates the validity of the proof.

$\mathcal{F}_{\mathrm{REV}}$ also uses a set $\mathbb{P}$. $\mathbb{P}$ contains the identifiers of the parties that retrieved the revocation parameters par_r. Additionally, $\mathcal{F}_{\mathrm{REV}}$ uses a set $\mathbb{E}_{ep}$ that, for an epoch ep, stores the identifiers of the parties that retrieved the epoch information $info$ and, in the non-hiding functionality, the revocation statuses in Tbl_{ep}. The hiding functionality $\mathcal{F}_{\mathrm{REV}}$ also uses a set $\mathbb{S}_{ep}$, which for an epoch ep stores the identifiers of the parties that retrieved the revocation statuses $\mathbf{x}[rh]$ of their revocation handles.

We now discuss the seven interfaces of the ideal functionality $\mathcal{F}_{\mathrm{REV}}$.

1. The $\mathsf{rev.setup.ini}$ message is sent by the revocation authority. $\mathcal{F}_{\mathrm{REV}}$ aborts if the $\mathsf{rev.setup.ini}$ message has already been sent. Otherwise $\mathcal{F}_{\mathrm{REV}}$ asks the simulator $\mathcal{S}$ to provide the parameters par_r, the trapdoor td_r, and the algorithms $\mathsf{REV.SimProve}$ and $\mathsf{REV.Extract}$. When $\mathcal{S}$ provides them, $\mathcal{F}_{\mathrm{REV}}$ aborts if they have already been sent. Otherwise, $\mathcal{F}_{\mathrm{REV}}$ initializes an empty set $\mathbb{P}$ and a table Tbl_{pr} to store proofs, and sends the revocation parameters to the revocation authority.
2. The $\mathsf{rev.get.ini}$ message is sent by any party $\mathcal{P}$ to get the parameters par_r.
3. The $\mathsf{rev.epoch.ini}$ message is sent by the revocation authority on input an epoch identifier ep and a list of revocation handles and revocation statuses $\langle \boxed{\text{H:}\ \mathcal{U}_i,}\ rh_i, \mathbf{x}[rh_i]\rangle_{i=1}^{n'}$.

In the hiding revocation functionality, the list also includes user identifiers $\mathcal{U}_i$. $\mathcal{F}_{\mathrm{REV}}$ asks $\mathcal{S}$ to provide the epoch information *info* for the epoch ep. In the hiding functionality, *info* is computed without knowledge of $\langle \boxed{\mathrm{H}\colon \mathcal{U}_i,} rh_i, \mathbf{x}[rh_i]\rangle_{i=1}^{n'}$, whereas in the non-hiding functionality $\mathcal{S}$ receives $\langle rh_i, \mathbf{x}[rh_i]\rangle_{i=1}^{n'}$. The epoch information *info* is later given as input to the algorithms REV.SimProve and REV.Extract. When $\mathcal{S}$ sends *info*, $\mathcal{F}_{\mathrm{REV}}$ aborts if the list $\langle \boxed{\mathrm{H}\colon \mathcal{U}_i,} rh_i, \mathbf{x}[rh_i]\rangle_{i=1}^{n'}$ for ep was not received before or if *info* for the epoch ep has already been received. Otherwise $\mathcal{F}_{\mathrm{REV}}$ creates a table Tbl_{ep} to store the list for the epoch ep and stores $(sid, ep, info, \mathsf{Tbl}_{ep})$. In the hiding functionality, if the revocation authority is corrupt, Tbl_{ep} is left empty and therefore the information $\langle \boxed{\mathrm{H}\colon \mathcal{U}_i,} rh_i, \mathbf{x}[rh_i]\rangle_{i=1}^{n'}$ is not required. The reason is that, if the hiding functionality requires this information when $\mathcal{RA}$ is corrupt, a construction that realizes this functionality would need to allow the extraction of $\langle \boxed{\mathrm{H}\colon \mathcal{U}_i,} rh_i, \mathbf{x}[rh_i]\rangle_{i=1}^{n'}$ in the security proof. In the construction, this would imply the use of extractable vector commitments, which, for the sake of efficiency, we chose to avoid. Therefore, the hiding $\mathcal{F}_{\mathrm{REV}}$, if $\mathcal{RA}$ is corrupt, learns the revocation statuses when they are disclosed through the rev.getstatus.$*$ interface or the rev.verify.$*$ interface. Finally, $\mathcal{F}_{\mathrm{REV}}$ sends the epoch ep and the epoch information *info* to the revocation authority.

4. The rev.getepoch.ini message is sent by any party $\mathcal{P}$ on input an epoch ep. After the simulator prompts the response with a message $(\mathsf{rev.getepoch.rep}, sid, ssid)$, the functionality sends the epoch information *info* to $\mathcal{P}$. In the non-hiding case, the functionality also sends the revocation statuses of all revocation handles to $\mathcal{P}$.
5. In the hiding functionality, the rev.getstatus.ini message is sent by a user $\mathcal{U}_i$ on input a revocation handle rh_i and an epoch ep. $\mathcal{F}_{\mathrm{REV}}$ works differently, depending on whether the revocation authority is corrupt or not:
   (a) If $\mathcal{RA}$ is honest, $\mathcal{F}_{\mathrm{REV}}$ aborts if there is no entry in Tbl_{ep} for $\mathcal{U}_i$ and rh_i. After the simulator prompts the response with a message $(\mathsf{rev.getstatus.rep}, sid, ssid)$, $\mathcal{F}_{\mathrm{REV}}$ sends the revocation status of rh_i to $\mathcal{U}_i$.
   (b) If $\mathcal{RA}$ is corrupt, $\mathcal{F}_{\mathrm{REV}}$ asks the simulator to provide the revocation status of rh_i. Then, if it was not stored in Tbl_{ep}, $\mathcal{F}_{\mathrm{REV}}$ stores the revocation status of rh_i in Tbl_{ep} and sends it to $\mathcal{U}_i$. If it is already stored, $\mathcal{F}_{\mathrm{REV}}$ only sends the revocation status to the user if the status sent by the functionality equals the one stored. Therefore, even if $\mathcal{RA}$ is corrupt, $\mathcal{F}_{\mathrm{REV}}$ ensures that $\mathcal{RA}$ associates a unique revocation status with a revocation handle during a given epoch.
6. The rev.prove.ini message is sent by an honest user $\mathcal{U}_i$ on input a commitment *ccom* to a revocation handle rh with the opening *copen*. $\mathcal{U}_i$ also inputs the epoch ep and the revocation list j. The commitment *ccom* consists of a commitment value *ccom*, commitment parameters *cparcom*, and a commitment verification algorithm COM.Verify. $\mathcal{F}_{\mathrm{REV}}$ aborts if rh and *copen* are not a valid opening of *ccom*. It also aborts if the revocation handle rh_i is not in Tbl_{ep}. (In the hiding functionality, it also aborts if the revocation authority is honest and rh_i is not associated with $\mathcal{U}_i$.) $\mathcal{F}_{\mathrm{REV}}$ runs $\mathsf{REV.SimProve}(sid, par_r, cparcom, ccom, ep, info_{ep}, j, \mathbf{x}[rh, j], td_r)$ to compute a proof pr that $\mathbf{x}[rh, j]$ is the revocation status of revocation handle rh with respect to the revocation list j at epoch ep. We note that pr does not reveal any information on the revocation handle rh, the opening *copen*, or the revocation status

$\mathbf{x}[rh]$ with respect to revocation lists other than j. $\mathcal{F}_{\mathrm{REV}}$ stores the proof as valid in Tbl_{pr}.

7. The $\mathsf{rev.verify.ini}$ message is sent by any honest party $\mathcal{P}$ on input a proof pr that b is the revocation status with respect to the revocation list j at epoch ep and to the revocation handle committed to in $ccom$. $ccom$ consists of a commitment value $ccom$, commitment parameters $cparcom$, and a commitment verification algorithm $\mathsf{COM.Verify}$. If the proof and the instance are stored in Tbl_{pr}, $\mathcal{F}_{\mathrm{REV}}$ outputs the verification result stored in Tbl_{pr} to ensure consistence. Otherwise $\mathcal{F}_{\mathrm{REV}}$ runs the algorithm $\mathsf{REV.Extract}$ to extract the revocation handle rh, the opening $copen$, and the revocation status $\mathbf{x}[rh]$ from the proof pr. Any construction that realizes $\mathcal{F}_{\mathrm{REV}}$ must allow extractable proofs. If extraction fails or if $ccom$ is not a commitment to rh and $copen$, $\mathcal{F}_{\mathrm{REV}}$ marks the proof as invalid. Otherwise, if the revocation authority is corrupt and the revocation status of rh is not stored in Tbl_{ep}, $\mathcal{F}_{\mathrm{REV}}$ stores it in Tbl_{ep}. After that, $\mathcal{F}_{\mathrm{REV}}$ also marks the proof as invalid if rh is not in Tbl_{ep} or if $b \neq \mathbf{x}[rh, j]$, where $\mathbf{x}[rh, j]$ is the revocation status stored in Tbl_{ep} for the revocation handle rh. In the hiding functionality, the proof is also marked as invalid when $b = \mathbf{x}[rh, j]$ but the revocation authority is honest and the user $\mathcal{U}$ associated with rh is honest. The reason is that the hiding functionality must prevent corrupt users from computing proofs about revocation handles associated with honest users because this constitutes a violation of the privacy of the revocation statuses. A construction that realizes $\mathcal{F}_{\mathrm{REV}}$ must use non-malleable proofs, i.e., it should not be possible to obtain a new proof from a valid proof without knowing the witness. We note that proofs for honest users are computed by $\mathcal{F}_{\mathrm{REV}}$ in the $\mathsf{rev.prove.*}$ interface and registered in Tbl_{pr} as valid, and are thus accepted by $\mathcal{F}_{\mathrm{REV}}$ in the verification interface without running the algorithm $\mathsf{REV.Extract}$.

We note that $\mathcal{F}_{\mathrm{REV}}$ does not allow parties to send their own revocation parameters par_r through the $\mathsf{rev.prove.*}$ and $\mathsf{rev.verify.*}$ interfaces. This means that a construction that realizes this functionality must use some form of trusted registration that allows the revocation authority to register par_r and the other parties to retrieve par_r in order to ensure that all honest parties use the same parameters.

$\mathcal{F}_{\mathrm{REV}}$ asks the simulator $\mathcal{S}$ to provide prove and extract algorithms at setup. Alternatively, it would be possible that the functionality asks the simulator to compute proofs and extract from proofs when the $\mathsf{rev.prove.*}$ and $\mathsf{rev.verify.*}$ interfaces are invoked. We chose the first alternative because it hides the computation and verification of proofs by the parties from the simulator.

4.2 Ideal Functionality for Anonymous Attribute Tokens $\mathcal{F}_{\mathrm{AT}}$

Next, we describe the ideal functionality of anonymous attribute tokens, $\mathcal{F}_{\mathrm{AT}}$. $\mathcal{F}_{\mathrm{AT}}$ interacts with an issuer $\mathcal{I}$, users $\mathcal{U}_i$ and any verifying parties $\mathcal{P}$. The issuer $\mathcal{I}$ issues some attributes $\langle a_l \rangle_{l=1}^{L}$ to a user $\mathcal{U}_i$. A user $\mathcal{U}_i$ can obtain a proof that some commitments $\langle ccom_l \rangle_{l=1}^{L}$ commit to attributes that were issued by $\mathcal{I}$. Any party $\mathcal{P}$ can verify a proof.

The interaction between the functionality $\mathcal{F}_{\mathrm{AT}}$ and the issuer $\mathcal{I}$, the users $\mathcal{U}_i$ and the verifying parties $\mathcal{P}$ takes place through the interfaces $\mathsf{at.setup.*}$, $\mathsf{at.get.*}$, $\mathsf{at.issue.*}$, $\mathsf{at.prove.*}$, and $\mathsf{at.verify.*}$.

1. The issuer $\mathcal{I}$ uses the at.setup.$*$ interface to initialize the functionality and obtain the parameters par_{at} of the anonymous attribute token scheme.
2. Any party $\mathcal{P}$ invokes the at.get.$*$ interface to obtain the parameters par_{at}.
3. The issuer $\mathcal{I}$ uses the at.issue.$*$ interface to issue some attributes $\langle a_l \rangle_{l=1}^{L}$ to a user $\mathcal{U}_i$.
4. An honest user $\mathcal{U}_i$ uses the at.prove.$*$ interface to get a proof pr that some commitments $\langle ccom_l \rangle_{l=1}^{L}$ commit to attributes that were issued by the issuer $\mathcal{I}$ to the user $\mathcal{U}_i$.
5. Any honest party $\mathcal{P}$ uses the at.verify.$*$ interface to verify a proof pr.

As we described before, the commitment parameters and the verification algorithm are attached to the commitment itself, and the functionality parses the commitment value to obtain them to run a commitment verification. $\mathcal{F}_{\mathrm{AT}}$ uses the following tables.

Tbl$_a$. Tbl$_a$ stores entries of the form $[\mathcal{U}_i, \langle a_l \rangle_{l=1}^{L}]$, which map a user $\mathcal{U}_i$ to a list of attributes $\langle a_l \rangle_{l=1}^{L}$ issued by $\mathcal{I}$.

Tbl$_{pr}$. Tbl$_{pr}$ stores entries of the form $[\langle ccom_l \rangle_{l=1}^{L}, pr, u]$, which consist of a proof instance $\langle ccom_l \rangle_{l=1}^{L}$, a proof pr, and a bit u that indicates whether the proof is valid.

$\mathcal{F}_{\mathrm{AT}}$ also uses a set $\mathbb{P}$. $\mathbb{P}$ contains the identifiers of the parties $\mathcal{P}$ that retrieved the attribute tokens parameters par_{at}.

Functionality $\mathcal{F}_{\mathrm{AT}}$

AT.SimProve and AT.Extract are ppt algorithms. $\mathcal{F}_{\mathrm{AT}}$ is parameterized by the system parameters sp, a maximum number of attributes $L_{\max}$ and a universe of attributes Ψ.

1. On input (at.setup.ini, sid) from $\mathcal{I}$:
   (a) Abort if $sid \neq (\mathcal{I}, sid')$ or if the tuple (sid) is already stored. Store (sid).
   (b) Send (at.setup.req, sid) to $\mathcal{S}$.

S. On input (at.setup.alg, sid, par_{at}, td_{at}, AT.SimProve, AT.Extract) from $\mathcal{S}$:
   (a) Abort if (sid) is not stored or if (sid, par_{at}, td_{at}, AT.SimProve, AT.Extract) is already stored. Store (sid, par_{at}, td_{at}, AT.SimProve, AT.Extract).
   (b) Initialize an empty table Tbl$_a$, an empty table Tbl$_{pr}$ and an empty set $\mathbb{P}$, and parse sid as $(\mathcal{I}, sid')$.
   (c) Send (at.setup.end, sid, par_{at}) to $\mathcal{I}$.

2. On input (at.get.ini, sid) from any party $\mathcal{P}$:
   (a) If there is a tuple (sid, par_{at}, td_{at}, AT.SimProve, AT.Extract) stored, set $par_{at}' \leftarrow par_{at}$; else set $par_{at}' \leftarrow \perp$.
   (b) Create a fresh $ssid$ and store $(ssid, \mathcal{P}, par_{at}')$.
   (c) Send (at.get.sim, $sid, ssid, par_{at}'$) to $\mathcal{S}$.

S. On input (at.get.rep, $sid, ssid$) from $\mathcal{S}$:
   (a) Abort if $(ssid, \mathcal{P}, par_{at}')$ is not stored.
   (b) If $par_{at}' \neq \perp$, include $\mathcal{P}$ in the set $\mathbb{P}$.
   (c) Delete record $(ssid, \mathcal{P}, par_{at}')$.
   (d) Send (at.get.end, sid, par_{at}') to $\mathcal{P}$.

3. On input (at.issue.ini, $sid, \mathcal{U}_i, \langle a_l \rangle_{l=1}^{L}$) from $\mathcal{I}$:
   (a) Abort if there is no tuple (sid, par_{at}, td_{at}, AT.SimProve, AT.Extract) stored, or if $\mathcal{U}_i$ is not a valid user identifier, or if $\langle a_l \rangle_{l=1}^{L} \not\subseteq \Psi$, or if $L > L_{\max}$.
   (b) Create a fresh $ssid$ and store $(ssid, \mathcal{U}_i, \langle a_l \rangle_{l=1}^{L})$.
   (c) Send (at.issue.sim, $sid, ssid, \mathcal{U}_i$) to $\mathcal{S}$.

S. On input $(\mathsf{at.issue.rep}, sid, ssid)$ from $\mathcal{S}$:
   (a) Abort if $(ssid, \mathcal{U}_i, \langle a_l \rangle_{l=1}^{L})$ is not stored or if $\mathcal{U}_i \not\subseteq \mathbb{P}$.
   (b) If $\mathcal{U}_i$ is honest, then append $[\mathcal{U}_i, \langle a_l \rangle_{l=1}^{L}]$ to Tbl_a; else append $[\mathcal{S}, \langle a_l \rangle_{l=1}^{L}]$ to Tbl_a.
   (c) Delete record $(ssid, \mathcal{U}_i, \langle a_l \rangle_{l=1}^{L})$.
   (d) Send $(\mathsf{at.issue.end}, sid, \langle a_l \rangle_{l=1}^{L})$ to $\mathcal{U}_i$.

4. On input $(\mathsf{at.prove.ini}, sid, \langle ccom_l, a_l, copen_l \rangle_{l=1}^{L})$ from an honest user $\mathcal{U}_i$:
   (a) Parse $ccom_l$ as $(ccom'_l, cparcom_l, \mathsf{COM.Verify}_l)$ and abort if $1 \neq \mathsf{COM.Verify}_l(cparcom_l, ccom'_l, a_l, copen_l)$ for any $l \in [1, L]$, or if there is no entry $[\mathcal{U}_i, \langle a_l \rangle_{l=1}^{L}]$ in Tbl_a .
   (b) Run $pr \leftarrow \mathsf{AT.SimProve}(sid, par_{at}, \langle cparcom_l, ccom'_l \rangle_{l=1}^{L}, td_{at})$.
   (c) Append $[\langle ccom_l \rangle_{l=1}^{L}, pr, 1]$ to Table Tbl_{pr}.
   (d) Send $(\mathsf{at.prove.end}, sid, pr)$ to $\mathcal{U}_i$.

5. On input $(\mathsf{at.verify.ini}, sid, \langle ccom_l \rangle_{l=1}^{L}, pr)$ from an honest party $\mathcal{P}$:
   (a) Abort if $\mathcal{P} \not\subseteq \mathbb{P}$.
   (b) Parse $ccom_l$ as $(ccom'_l, cparcom_l, \mathsf{COM.Verify}_l)$ for any $l \in [1, L]$.
   (c) If there is an entry $[\langle ccom_l \rangle_{l=1}^{L}, pr, u]$ in Tbl_{pr}, set $v \leftarrow u$.
   (d) Else, do the following:
      i. Run $\langle a_l, copen_l \rangle_{l=1}^{L} \leftarrow \mathsf{AT.Extract}(sid, par_{at}, \langle cparcom_l, ccom'_l \rangle_{l=1}^{L}, td_{at}, pr)$.
      ii. If $\mathcal{I}$ is corrupt, proceed as follows. If $\langle a_l, copen_l \rangle_{l=1}^{L} = 1$, set $v \leftarrow 1$; else set $v \leftarrow 0$.
      iii. Else, if $\langle a_l, copen_l \rangle_{l=1}^{L} = \perp$ or $1 \neq \mathsf{COM.Verify}_l(cparcom_l, ccom'_l, a_l, copen_l)$ for any $l \in [1, L]$, set $v \leftarrow 0$.
      iv. Else, if there exists an entry $[\mathcal{S}, \langle a_l \rangle_{l=1}^{L}] \in \mathsf{Tbl}_a$, set $v \leftarrow 1$; else set $v \leftarrow 0$.
      v. Append $[\langle ccom_l \rangle_{l=1}^{L}, pr, v]$ to Table Tbl_{pr}.
   (e) Send $(\mathsf{at.verify.end}, sid, v)$ to $\mathcal{P}$.

We now discuss the five interfaces of the ideal functionality $\mathcal{F}_{\text{AT}}$.

1. The at.setup.ini message is sent by the issuer $\mathcal{I}$. The restriction that the issuer's identity must be included in the session identifier $sid = (\mathcal{I}, sid')$ guarantees that each issuer can initialize its own instance of the functionality. $\mathcal{F}_{\text{AT}}$ requests the simulator $\mathcal{S}$ for the parameters and algorithms. $\mathcal{S}$ sends the parameters par_{at}, the trapdoor td_{at}, and the algorithms (AT.SimProve, AT.Extract) for proof computation and proof extraction. Finally, $\mathcal{F}_{\text{AT}}$ initializes two empty tables, Tbl_a and Tbl_{pr}, and sends the received parameters par_{at} to $\mathcal{I}$.
2. The at.get.ini message allows any party to request par_{at}.
3. The at.issue.ini message is sent by the issuer on input a user identity and a list of attributes. $\mathcal{F}_{\text{AT}}$ creates a subsession identifier $ssid$ and sends the user identity to the simulator. The simulator indicates when the issuance is to be finalized by sending a $(\mathsf{at.issue.rep}, sid, ssid)$ message. At this point, the issuance is recorded in Tbl_a. If the user is honest, the issuance is recorded under the correct user's identity, which in the real world requires any instantiating protocol to set up an authenticated channel to the user to ensure this. If the user is corrupt, the attributes are recorded as belonging to the simulator, modeling that corrupt users may pool their attribute tokens. Note that the simulator is not given the attribute values issued, so the real-world protocol must hide these from the adversary.
4. The at.prove.ini message lets an honest user $\mathcal{U}_i$ request a proof that the attributes $\langle a_l \rangle_{l=1}^{L}$ issued to her by $\mathcal{I}$ are committed in the commitments $\langle com_l \rangle_{l=1}^{L}$. The commit-

ment parameters and the commitment verification algorithm attached to each commitment value allow $\mathcal{F}_{\text{AT}}$ to compute and verify proofs about commitments to attributes, such that the commitments are generated externally, e.g., by the functionality $\mathcal{F}_{\text{NIC}}$. $\mathcal{F}_{\text{AT}}$ computes a proof by running $\mathsf{AT.SimProve}$, which does not receive the witness as input. Therefore, any construction that realizes $\mathcal{F}_{\text{AT}}$ must use zero-knowledge proofs. $\mathcal{F}_{\text{AT}}$ stores the proof in Tbl_{pr} and sends the proof to $\mathcal{U}_i$.

5. The $\mathsf{at.verify.ini}$ message allows any honest party to request the verification of a proof pr with respect to the instance $\langle ccom_l \rangle_{l=1}^{L}$. If the instance-proof pair is stored in the table Tbl_{pr}, then the functionality replies with the stored verification result to ensure consistency. If not, the functionality runs the algorithm $\mathsf{AT.Extract}$. If the issuer is corrupt, the functionality interprets the output of $\mathsf{AT.Extract}$ as a bit b that indicates whether the proof is valid or not. If the issuer is honest, the functionality interprets the output as the witness $\langle a_l, copen_l \rangle_{l=1}^{L}$. The functionality only marks the proof as correct if extraction did not fail, if $\langle a_l, copen_l \rangle_{l=1}^{L}$ are correct openings of the commitments in the instance, and if any corrupt user was issued the attributes $\langle a_l \rangle_{l=1}^{L}$. A construction that realizes $\mathcal{F}_{\text{AT}}$ must use non-malleable proofs, i.e., $\mathcal{F}_{\text{AT}}$ enforces that it is not possible to obtain a new proof from a valid proof without knowing the witness. We note that proofs for honest users are computed by $\mathcal{F}_{\text{AT}}$ in the $\mathsf{at.prove.}*$ interface and registered in Tbl_{pr} as valid, and are thus accepted by $\mathcal{F}_{\text{AT}}$ in the verification interface without running the algorithm $\mathsf{AT.Extract}$. Finally, the functionality stores the proof-instance pair and the verification result in Tbl_{pr} and sends the verification result to the party.

We note that $\mathcal{F}_{\text{AT}}$ does not allow parties to send their own revocation parameters par_{at} through the $\mathsf{at.prove.}*$ and $\mathsf{at.verify.}*$ interfaces. This means that a construction that realizes this functionality must use some form of trusted registration that allows the issuer to register par_{at} and the other parties to retrieve par_{at} to ensure that all honest parties use the same parameters.

$\mathcal{F}_{\text{AT}}$ asks the simulator $\mathcal{S}$ to provide prove and extract algorithms at setup. Alternatively, it would be possible that the functionality asks the simulator to compute proofs and extract from proofs when the $\mathsf{at.prove.}*$ and $\mathsf{at.verify.}*$ interfaces are invoked. We chose the first alternative because it hides the computation and verification of proofs by the parties from the simulator.

We refer to the full version of the paper for a construction that realizes $\mathcal{F}_{\text{AT}}$ and its security analysis.

5 Anonymous Attribute Tokens with Revocation

In this section we a high-level description of the ideal functionality $\mathcal{F}_{\text{TR}}$ of anonymous attribute tokens with revocation. For its formal description we defer to the full version of this paper. Similarly, we here consider only the version of functionality $\mathcal{F}_{\text{TR}}$ where the revocation statuses of every revocation handle are public. We then describe a construction for this version of $\mathcal{F}_{\text{TR}}$ that uses the functionalities $\mathcal{F}_{\text{NIC}}$, $\mathcal{F}_{\text{REV}}$, and $\mathcal{F}_{\text{AT}}$ and illustrates how to modularly design hybrid protocols in the UC framework.

5.1 Ideal Functionality $\mathcal{F}_{\mathrm{TR}}$ of Anonymous Attribute Tokens with Revocation

$\mathcal{F}_{\mathrm{TR}}$ interacts with an issuer $\mathcal{I}$, a revocation authority $\mathcal{RA}$, users $\mathcal{U}_i$, and any verifying party $\mathcal{P}$. The issuer $\mathcal{I}$ issues some attributes $\langle a_l \rangle_{l=1}^{L}$ and a revocation handle rh to a user $\mathcal{U}_i$. The revocation authority $\mathcal{RA}$ associates a revocation status $\mathbf{x}[rh]$ with each revocation handle rh. $\mathbf{x}[rh]$ is a vector of m bits, such that each bit $\mathbf{x}[rh, j]$ denotes the revocation status of the revocation handle rh with respect to the revocation list $j \in [1, m]$. A user $\mathcal{U}_i$ can prove to any party that a set of attributes $\langle a_l \rangle_{l=1}^{L}$ and a revocation handle were issued by $\mathcal{I}$. $\mathcal{U}_i$ also proves that b is the revocation status that the revocation authority $\mathcal{RA}$ associated with rh for the revocation list j and the epoch ep.

The interaction between the functionality $\mathcal{F}_{\mathrm{TR}}$ and the issuer $\mathcal{I}$, the revocation authority $\mathcal{RA}$, the users $\mathcal{U}_i$, and any verifying party $\mathcal{P}$ takes place through the interfaces $\mathsf{tr.setupi.}*$, $\mathsf{tr.setupra.}*$, $\mathsf{tr.setupp.}*$, $\mathsf{tr.issue.}*$, $\mathsf{tr.epoch.}*$, $\mathsf{tr.getepoch.}*$, and $\mathsf{tr.prove.}*$.

1. The issuer $\mathcal{I}$ invokes the $\mathsf{tr.setupi.}*$ interface for initialization.
2. The revocation authority $\mathcal{RA}$ invokes the $\mathsf{tr.setupra.}*$ interface for initialization.
3. Any user or verifying party $\mathcal{P}$ invokes the $\mathsf{tr.setupp.}*$ interface for initialization.
4. The issuer $\mathcal{I}$ uses the $\mathsf{tr.issue.}*$ interface to issue the attributes $\langle a_l \rangle_{l=1}^{L}$ and the revocation handle rh to a user $\mathcal{U}_i$.
5. The revocation authority $\mathcal{RA}$ uses the $\mathsf{tr.epoch.}*$ interface to send a list of revocation handles rh along with their respective revocation statuses $\mathbf{x}[rh]$ for the epoch ep.
6. Any party $\mathcal{P}$ uses the $\mathsf{rev.getepoch.}*$ interface to get the full list of revocation handles and revocation statuses $\langle rh_i, \mathbf{x}[rh_i] \rangle_{i=1}^{n'}$ for the epoch ep.
7. A user $\mathcal{U}_i$ uses the $\mathsf{tr.prove.}*$ interface to prove that some attributes $\langle a_l \rangle_{l=1}^{L}$ and a revocation handle rh were issued by $\mathcal{I}$. $\mathcal{U}_i$ also proves that $\mathbf{x}[rh, j]$ is the revocation status associated with rh for the revocation list j and the epoch ep.

5.2 Construction of Anonymous Attribute Tokens with Revocation

We now describe the construction of anonymous attribute tokens with revocation Π_{TR}. The construction Π_{TR} works in the ($\mathcal{F}_{\mathrm{SMT}}$, $\mathcal{F}_{\mathrm{ASMT}}$, $\mathcal{F}_{\mathrm{NIC}}$, $\mathcal{F}_{\mathrm{REV}}$, $\mathcal{F}_{\mathrm{AT}}$)-hybrid model, where the parties make use of the ideal functionalities for secure message transmission $\mathcal{F}_{\mathrm{SMT}}$ and anonymous secure message transmission $\mathcal{F}_{\mathrm{ASMT}}$ in [4]. The parties also use the ideal functionality for commitments $\mathcal{F}_{\mathrm{NIC}}$ described in Section 3.1, the ideal functionality for revocation $\mathcal{F}_{\mathrm{REV}}$ described in Section 4.1, and the ideal functionality for anonymous attribute tokens $\mathcal{F}_{\mathrm{AT}}$ described in Section 4.2.

This construction illustrates our mechanism for a modular design of hybrid protocols in the UC framework. In the issuing phase, the users receive attributes and a revocation handle from the issuer through $\mathcal{F}_{\mathrm{AT}}$. To compute an attribute token and show that it has not been revoked, a user first obtains a commitment to the revocation handle and an opening from $\mathcal{F}_{\mathrm{NIC}}$. Then the revocation handle, the commitment and the opening are sent to $\mathcal{F}_{\mathrm{REV}}$ to get a non-interactive proof of non-revocation. Similarly, the revocation handle, the commitment and the opening, along with commitments and openings to some attributes issued along with the revocation handle, are sent to $\mathcal{F}_{\mathrm{AT}}$ to obtain an attribute token. Thanks to the fact that $\mathcal{F}_{\mathrm{AT}}$ and $\mathcal{F}_{\mathrm{REV}}$ run the commitment verification algorithm, it is ensured that $\mathcal{F}_{\mathrm{AT}}$ and $\mathcal{F}_{\mathrm{REV}}$ receive the same revocation handle.

The construction Π_{TR} is executed by an issuer $\mathcal{I}$, a revocation authority $\mathcal{RA}$, users $\mathcal{U}_i$, and any verifying party $\mathcal{P}$. Those parties are activated through the $\mathsf{tr.setupi}.*$, $\mathsf{tr.setupra}.*$, $\mathsf{tr.setupp}.*$, $\mathsf{tr.issue}.*$, $\mathsf{tr.epoch}.*$, $\mathsf{tr.getepoch}.*$, and $\mathsf{tr.prove}.*$ interfaces. Briefly, the construction Π_{TR} works as follows.

1. The issuer $\mathcal{I}$ receives $(\mathsf{tr.setupi.ini}, sid)$ as input. If the functionality $\mathcal{F}_{\mathrm{AT}}$ was not set up, $\mathcal{I}$ invokes the $\mathsf{at.setup}.*$ interface of $\mathcal{F}_{\mathrm{AT}}$; else $\mathcal{I}$ aborts.
2. The revocation authority receives $(\mathsf{tr.setupra.ini}, sid)$ as input. $\mathcal{RA}$ aborts if setup has already been run. Otherwise, $\mathcal{RA}$ invokes the $\mathsf{at.get}.*$ interface of $\mathcal{F}_{\mathrm{AT}}$ and aborts if $\mathcal{F}_{\mathrm{AT}}$ does not return the attribute token parameters. In addition, $\mathcal{RA}$ invokes the $\mathsf{com.setup}.*$ interface of $\mathcal{F}_{\mathrm{NIC}}$ to setup the commitment functionality. Finally, $\mathcal{RA}$ invokes the $\mathsf{rev.setup}.*$ interface of $\mathcal{F}_{\mathrm{REV}}$.
3. A user or a verifying party $\mathcal{P}$ receives $(\mathsf{tr.setupp.ini}, sid)$ as input. $\mathcal{P}$ aborts if the setup has already been run. Otherwise $\mathcal{P}$ invokes the $\mathsf{at.get}.*$ interface of $\mathcal{F}_{\mathrm{AT}}$ and aborts if $\mathcal{F}_{\mathrm{AT}}$ does not return the attribute token parameters. Then $\mathcal{P}$ invokes the $(\mathsf{rev.get.ini}, sid)$ interface of $\mathcal{F}_{\mathrm{REV}}$ and aborts if $\mathcal{F}_{\mathrm{REV}}$ does not return the revocation parameters. Finally $\mathcal{P}$ invokes the $\mathsf{com.setup}.*$ interface of $\mathcal{F}_{\mathrm{NIC}}$.
4. The issuer $\mathcal{I}$ receives $(\mathsf{tr.issue.ini}, sid, \mathcal{U}_i, \langle a_l \rangle_{l=1}^{L}, rh)$ as input. If the issuer setup has not been run, $\mathcal{I}$ aborts. Otherwise $\mathcal{I}$ invokes the $\mathsf{at.issue}.*$ interface of $\mathcal{F}_{\mathrm{AT}}$ to issue the attributes $\langle a_l \rangle_{l=1}^{L}$ and the revocation handle rh to the user $\mathcal{U}_i$. $\mathcal{U}_i$ aborts if the user setup has not been run by $\mathcal{U}_i$.
5. The revocation authority $\mathcal{RA}$ receives $(\mathsf{tr.epoch.ini}, sid, \langle rh_i, \mathbf{x}[rh_i] \rangle_{i=1}^{n'})$ as input. $\mathcal{RA}$ aborts if the $\mathcal{RA}$ setup has not been run or if the input values are invalid. Otherwise $\mathcal{RA}$ uses the $\mathsf{rev.epoch}.*$ interface of $\mathcal{F}_{\mathrm{REV}}$ to send the revocation information and obtain the epoch information $info$ for the current epoch ep.
6. A party $\mathcal{P}$ receives $(\mathsf{tr.getepoch.ini}, sid, ep)$ as input. $\mathcal{P}$ aborts if it did not run the setup. Otherwise $\mathcal{P}$ invokes the interface $\mathsf{rev.getepoch}.*$ of $\mathcal{F}_{\mathrm{REV}}$ to get the revocation information Tbl_{ep} for ep.
7. The user $\mathcal{U}_i$ receives $(\mathsf{tr.prove.ini}, sid, \mathcal{P}, \langle a_l \rangle_{l=1}^{L}, rh, ep, j, b)$ as input. $\mathcal{U}_i$ aborts if the revocation handle and the attributes were not issued to her, if $\mathcal{U}_i$ did not get the epoch ep or if the revocation status given by $\mathcal{RA}$ is not b for list j and rh at epoch ep. Otherwise $\mathcal{U}_i$ invokes the $\mathsf{com.commit}.*$ interface of functionality $\mathcal{F}_{\mathrm{NIC}}$ to obtain commitments and openings for the attributes and for the revocation handle that were issued by $\mathcal{I}$. Note that, for simplicity, we reveal all attributes of the token, except the revocation handle. Then $\mathcal{U}_i$ invokes the $\mathsf{at.prove}.*$ interface of $\mathcal{F}_{\mathrm{AT}}$ to get a proof that the committed attributes were issued by $\mathcal{I}$. $\mathcal{U}_i$ also invokes the $\mathsf{rev.prove}.*$ interface of $\mathcal{F}_{\mathrm{REV}}$ to get a proof that the revocation status of the committed revocation handle for list j at epoch ep is b. The user $\mathcal{U}_i$ sends the commitments, the proofs, and the openings of the commitments to the attributes through an instance of $\mathcal{F}_{\mathrm{ASMT}}$ to the verifying party $\mathcal{P}$. $\mathcal{P}$ aborts if it did not get the epoch ep. Otherwise $\mathcal{P}$ validates the commitment parameters for the commitment to the revocation handle by calling the $\mathsf{com.validate}.*$ interface of $\mathcal{F}_{\mathrm{NIC}}$, because it cannot verify the full commitment itself without the opening, and invokes the $\mathsf{com.verify}.*$ interface of $\mathcal{F}_{\mathrm{NIC}}$ to verify the openings of the commitments to the attributes revealed. The $\mathsf{at.verify}.*$ interface of $\mathcal{F}_{\mathrm{AT}}$ and the $\mathsf{rev.verify}.*$ interface of $\mathcal{F}_{\mathrm{REV}}$ are used to verify the respective proofs.

Construction Π_{TR}

Π_{TR} is parameterized by the system parameters sp and uses the ideal functionalities $\mathcal{F}_{\mathrm{SMT}}$, $\mathcal{F}_{\mathrm{ASMT}}$, $\mathcal{F}_{\mathrm{NIC}}$, $\mathcal{F}_{\mathrm{REV}}$, and $\mathcal{F}_{\mathrm{AT}}$. The constants used are the maximum number of attributes $L_{\max}$, the universe of attributes Ψ, the maximum number of revocation lists m, and the maximum number of revocation handles n.

1. On input $(\mathsf{tr.setupi.ini}, sid)$, $\mathcal{I}$ does the following:
   (a) $\mathcal{I}$ aborts if $sid \neq (\mathcal{I}, \mathcal{RA}, sid')$, or if $(sid, \mathsf{tr.setupi})$ is already stored.
   (b) $\mathcal{I}$ sends the message $(\mathsf{at.setup.ini}, sid)$ to $\mathcal{F}_{\mathrm{AT}}$ and receives the message $(\mathsf{at.setup.end}, sid, par_{at})$ from $\mathcal{F}_{\mathrm{AT}}$.
   (c) Store $(sid, \mathsf{tr.setupi})$.
   (d) Output $(\mathsf{tr.setupi.end}, sid)$.
2. On input $(\mathsf{tr.setupra.ini}, sid)$, $\mathcal{RA}$ does the following:
   (a) $\mathcal{RA}$ aborts if $(sid, \mathsf{tr.setupra})$ is already stored.
   (b) $\mathcal{RA}$ sends $(\mathsf{at.get.ini}, sid)$ to $\mathcal{F}_{\mathrm{AT}}$ and receives $(\mathsf{at.get.end}, sid, par_{at})$ from $\mathcal{F}_{\mathrm{AT}}$. If $par_{at} = \perp$, $\mathcal{RA}$ aborts.
   (c) $\mathcal{RA}$ sends $(\mathsf{com.setup.ini}, sid)$ to $\mathcal{F}_{\mathrm{NIC}}$ and receives $(\mathsf{com.setup.end}, sid, OK)$ from $\mathcal{F}_{\mathrm{NIC}}$.
   (d) $\mathcal{RA}$ parses sid as $(\mathcal{I}, \mathcal{RA}, sid')$, sets $sid_{\mathrm{REV}} \leftarrow (\mathcal{RA}, sid')$, sends $(\mathsf{rev.setup.ini}, sid)$ to $\mathcal{F}_{\mathrm{REV}}$ and receives $(\mathsf{rev.setup.end}, sid, par_r)$ from $\mathcal{F}_{\mathrm{REV}}$.
   (e) Store $(sid, \mathsf{tr.setupra})$.
   (f) Output $(\mathsf{tr.setupra.end}, sid)$.
3. On input $(\mathsf{tr.setupp.ini}, sid)$, a user or verifying party $\mathcal{P}$ does the following:
   (a) $\mathcal{P}$ aborts if $(sid, \mathsf{tr.setupp})$ is already stored.
   (b) $\mathcal{P}$ sends $(\mathsf{at.get.ini}, sid)$ to $\mathcal{F}_{\mathrm{AT}}$ and receives $(\mathsf{at.get.end}, sid, par_{at})$ from $\mathcal{F}_{\mathrm{AT}}$. If $par_{at} = \perp$, $\mathcal{P}$ aborts.
   (c) $\mathcal{P}$ parses sid as $(\mathcal{I}, \mathcal{RA}, sid')$, sets $sid_{\mathrm{REV}} \leftarrow (\mathcal{RA}, sid')$, sends $(\mathsf{rev.get.ini}, sid)$ to $\mathcal{F}_{\mathrm{REV}}$ and receives $(\mathsf{rev.get.end}, sid, par_r)$ from $\mathcal{F}_{\mathrm{REV}}$. If $par_r = \perp$, $\mathcal{P}$ aborts.
   (d) $\mathcal{P}$ sends $(\mathsf{com.setup.ini}, sid)$ to $\mathcal{F}_{\mathrm{NIC}}$ and receives $(\mathsf{com.setup.end}, sid, OK)$ from $\mathcal{F}_{\mathrm{NIC}}$.
   (e) Store $(sid, \mathsf{tr.setupp})$.
   (f) Output $(\mathsf{tr.setupp.end}, sid)$.
4. On input $(\mathsf{tr.issue.ini}, sid, \mathcal{U}_i, \langle a_l \rangle_{l=1}^{L}, rh)$, $\mathcal{I}$ and $\mathcal{U}_i$ do the following:
   (a) $\mathcal{I}$ aborts if $(sid, \mathsf{tr.setupi})$ is not stored, or if $\langle a_l \rangle_{l=1}^{L} \not\subseteq \Psi$, or if $rh \notin [1, n]$, or if $L > L_{\max}$.
   (b) $\mathcal{I}$ sets $a_{L+1} \leftarrow rh$ and sends $(\mathsf{at.issue.ini}, sid, \mathcal{U}_i, \langle a_l \rangle_{l=1}^{L+1})$ to $\mathcal{F}_{\mathrm{AT}}$.
   (c) $\mathcal{U}_i$ receives $(\mathsf{at.issue.end}, sid, \langle a_l \rangle_{l=1}^{L+1})$ from $\mathcal{F}_{\mathrm{AT}}$.
   (d) $\mathcal{U}_i$ aborts if $(sid, \mathsf{tr.setupp})$ is not stored.
   (e) $\mathcal{U}_i$ sets $rh \leftarrow a_{L+1}$, stores $[\langle a_l \rangle_{l=1}^{L}, rh]$, and outputs $(\mathsf{tr.issue.end}, sid, \langle a_l \rangle_{l=1}^{L}, rh)$.
5. On input $(\mathsf{tr.epoch.ini}, sid, ep, \langle rh_i, \mathbf{x}[rh_i] \rangle_{i=1}^{n'})$, $\mathcal{RA}$ does the following:
   (a) $\mathcal{RA}$ aborts if $(sid, \mathsf{tr.setupra})$ is not stored, or if $n' > n$, or if, for $i = 1$ to n', $rh_i \notin [1, n]$ or $\mathbf{x}[rh_i] \notin [0, 2^m)$, or if $(ep, info)$ is already stored.
   (b) $\mathcal{RA}$ parses sid as $(\mathcal{I}, \mathcal{RA}, sid')$, sets $sid_{\mathrm{REV}} \leftarrow (\mathcal{RA}, sid')$, sends $(\mathsf{rev.epoch.ini}, sid_{\mathrm{REV}}, ep, \langle rh_i, \mathbf{x}[rh_i] \rangle_{i=1}^{n'})$ to $\mathcal{F}_{\mathrm{REV}}$, receives $(\mathsf{rev.epoch.end}, sid_{\mathrm{REV}}, ep, info)$ from $\mathcal{F}_{\mathrm{REV}}$, and stores $(ep, info)$.
   (c) $\mathcal{RA}$ outputs $(\mathsf{tr.epoch.end}, sid, ep)$.
6. On input $(\mathsf{tr.getepoch.ini}, sid, ep)$, a party $\mathcal{P}$ does the following:

(a) $\mathcal{P}$ aborts if $(sid, \mathsf{tr.setupp})$ is not stored.

(b) $\mathcal{P}$ parses sid as $(\mathcal{I}, \mathcal{RA}, sid')$, sets $sid_{\mathrm{REV}} \leftarrow (\mathcal{RA}, sid')$, sends the message $(\mathsf{rev.getepoch.ini}, sid_{\mathrm{REV}}, ep)$ to $\mathcal{F}_{\mathrm{REV}}$, and receives the message $(\mathsf{rev.getepoch.end}, sid_{\mathrm{REV}}, info, \mathsf{Tbl}_{ep})$ from $\mathcal{F}_{\mathrm{REV}}$.

(c) If $info \neq \perp$, $\mathcal{P}$ sets $b \leftarrow 1$ and stores $[ep, info, \mathsf{Tbl}_{ep}]$; else $b \leftarrow 0$.

(d) $\mathcal{P}$ outputs $(\mathsf{tr.getepoch.end}, sid, b, \mathsf{Tbl}_{ep})$.

7. On input $(\mathsf{tr.prove.ini}, sid, \mathcal{P}, \langle a_l \rangle_{l=1}^{L}, rh, ep, j, b)$, a user $\mathcal{U}_i$ and a verifying party $\mathcal{P}$ do the following:

(a) $\mathcal{U}_i$ aborts if an entry $[\langle a'_l \rangle_{l=1}^{L}, rh']$ such that $rh = rh'$ and $\langle a_l \rangle_{l=1}^{L} = \langle a'_l \rangle_{l=1}^{L}$ is not stored.

(b) If $[ep, info, \mathsf{Tbl}_{ep}]$ is not stored, $\mathcal{U}_i$ aborts.

(c) $\mathcal{U}_i$ aborts if $b \neq \mathbf{x}[rh, j]$, where $\mathbf{x}[rh, j]$ is the stored revocation status of rh for list j at epoch ep.

(d) $\mathcal{U}_i$ sets $a_{L+1} \leftarrow rh$, and, for $l = 1$ to $L+1$, $\mathcal{U}_i$ sends $(\mathsf{com.commit.ini}, sid, a_l)$ to $\mathcal{F}_{\mathrm{NIC}}$, and receives $(\mathsf{com.commit.end}, sid, ccom_l, copen_l)$ from $\mathcal{F}_{\mathrm{NIC}}$.

(e) $\mathcal{U}_i$ parses sid as $(\mathcal{I}, \mathcal{RA}, sid')$, sets $sid_{\mathrm{REV}} \leftarrow (\mathcal{RA}, sid')$, sends $(\mathsf{rev.prove.ini}, sid_{\mathrm{REV}}, ccom_{L+1}, rh, copen_{L+1}, ep, j)$ to $\mathcal{F}_{\mathrm{REV}}$, and receives $(\mathsf{rev.prove.end}, sid_{\mathrm{REV}}, pr)$ from $\mathcal{F}_{\mathrm{REV}}$.

(f) $\mathcal{U}_i$ sends $(\mathsf{at.prove.ini}, sid, \langle ccom_l, a_l, copen_l \rangle_{l=1}^{L+1})$ to $\mathcal{F}_{\mathrm{AT}}$ and receives $(\mathsf{at.prove.end}, sid, pr')$ from $\mathcal{F}_{\mathrm{AT}}$.

(g) $\mathcal{U}_i$ picks a fresh sid_{ASMT} and sends $(\mathsf{asmt.send.ini}, sid_{\mathrm{ASMT}}, \langle sid, j, ep, \mathbf{x}[rh, j], (a_l, copen_l)_{l=1}^{L}, (ccom_l)_{l=1}^{L+1}, pr, pr' \rangle, \mathcal{P})$ to $\mathcal{F}_{\mathrm{ASMT}}$.

(h) $\mathcal{P}$ receives the message $(\mathsf{asmt.send.end}, sid_{\mathrm{ASMT}}, \langle sid, j, ep, \mathbf{x}[rh, j], (a_l, copen_l)_{l=1}^{L}, (ccom_l)_{l=1}^{L+1}, pr, pr' \rangle)$ from $\mathcal{F}_{\mathrm{ASMT}}$.

(i) $\mathcal{P}$ aborts if $[ep, info, \mathsf{Tbl}_{ep}]$ is not stored.

(j) For $l = 1$ to L, $\mathcal{P}$ sends $(\mathsf{com.verify.ini}, sid, ccom_l, a_l, copen_l)$ to $\mathcal{F}_{\mathrm{NIC}}$ and receives $(\mathsf{com.verify.end}, sid, v_l)$ from $\mathcal{F}_{\mathrm{NIC}}$.

(k) $\mathcal{P}$ sends $(\mathsf{com.validate.ini}, sid, ccom_{L+1})$ to $\mathcal{F}_{\mathrm{NIC}}$, receives $(\mathsf{com.validate.end}, sid, v'')$ from $\mathcal{F}_{\mathrm{NIC}}$.

(l) $\mathcal{P}$ parses sid as $(\mathcal{I}, \mathcal{RA}, sid')$, sets $sid_{\mathrm{REV}} \leftarrow (\mathcal{RA}, sid')$, sends $(\mathsf{rev.verify.ini}, sid_{\mathrm{REV}}, ccom_{L+1}, ep, j, \mathbf{x}[rh, j], pr)$ to $\mathcal{F}_{\mathrm{REV}}$, and receives $(\mathsf{rev.verify.end}, sid_{\mathrm{REV}}, v)$ from $\mathcal{F}_{\mathrm{REV}}$.

(m) $\mathcal{P}$ sends $(\mathsf{at.verify.ini}, sid, \langle ccom_l \rangle_{l=1}^{L+1}, pr')$ to $\mathcal{F}_{\mathrm{AT}}$ and receives $(\mathsf{at.verify.end}, sid, v')$ from $\mathcal{F}_{\mathrm{AT}}$.

(n) If $v = v' = v'' = 1$ and $v_l = 1$ for $l = 1$ to L, then $\mathcal{P}$ outputs $(\mathsf{tr.prove.end}, sid, \langle a_l \rangle_{l=1}^{L}, \mathbf{x}[rh, j], ep, j)$; else $\mathcal{P}$ aborts.

6 Conclusion and Future Work

We have proposed a method for the modular design of cryptographic protocols in the UC framework. Our method allows one to compose two or more functionalities and to ensure that some inputs to those functionalities are equal or an output of one functionality is used as input to another functionality. For this purpose, the functionalities are amended to receive commitments as inputs and to verify them. In addition, we propose new ideal functionalities for commitments that, unlike existing ones, output cryptographic commitments. To illustrate our framework, we have shown a protocol for attribute tokens with revocation that uses our commitment functionality and ideal functionalities for attribute tokens and for revocation that receive committed inputs. As future work, we consider the modular design of other cryptographic protocols with our method as well as investigating the relations between our UC-based definitions and game-based definitions for attribute-based tokens and revocation.

References

1. J. Camenisch, M. Kohlweiss, and C. Soriente. An accumulator based on bilinear maps and efficient revocation for anonymous credentials. In *PKC 2009*, pages 481–500.
2. J. Camenisch, M. Kohlweiss, and C. Soriente. Solving revocation with efficient update of anonymous credentials. In *SCN 2010*, pages 454–471.
3. J. Camenisch, S. Krenn, A. Lehmann, G. L. Mikkelsen, G. Neven, and M. Ø. Pedersen. Formal treatment of privacy-enhancing credential systems. In *SAC 2015*, pages 3–24.
4. J. Camenisch, A. Lehmann, G. Neven, and A. Rial. Privacy-preserving auditing for attribute-based credentials. In *ESORICS 2014*, pages 109–127.
5. J. Camenisch and A. Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In *CRYPTO 2002*, pages 61–76.
6. R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS 2001*, pages 136–145.
7. R. Canetti. Universally composable signature, certification, and authentication. In *CSFW 2004*, page 219.
8. R. Canetti and M. Fischlin. Universally composable commitments. In *CRYPTO 2001*, pages 19–40.
9. R. Canetti, Y. Lindell, R. Ostrovsky, and A. Sahai. Universally composable two-party and multi-party secure computation. In *STOC 2002*, pages 494–503.
10. D. Catalano and D. Fiore. Vector commitments and their applications. In *PKC 2013*, pages 55–72.
11. I. Damgård and J. B. Nielsen. Perfect hiding and perfect binding universally composable commitment schemes with constant expansion factor. In *CRYPTO 2002*, pages 581–597.
12. T. K. Frederiksen, T. P. Jakobsen, J. B. Nielsen, and R. Trifiletti. On the complexity of additively homomorphic uc commitments. ePrint, Report 2015/694.
13. Jens Groth. Homomorphic trapdoor commitments to group elements. *ePrint, 2009/007*.
14. D. Hofheinz and M. Backes. How to break and repair a Universally Composable signature functionality. In *ICS 2004*, pages 61–72.
15. D. Hofheinz and J. Müller-Quade. Universally composable commitments using random oracles. In *TCC 2004*, pages 58–76.
16. J. Katz. Universally composable multi-party computation using tamper-proof hardware. In *EUROCRYPT 2007*, pages 115–128.
17. Y. Lindell. Highly-efficient universally-composable commitments based on the DDH assumption. In *EUROCRYPT 2011*, pages 446–466.
18. P. MacKenzie and K. Yang. On simulation-sound trapdoor commitments. In *EUROCRYPT 2004*, pages 382–400.
19. T. Moran and G. Segev. David and goliath commitments: UC computation for asymmetric parties using tamper-proof hardware. In *EUROCRYPT 2008*, pages 527–544.
20. T. Nakanishi, H. Fujii, H. Yuta, and N. Funabiki. Revocable group signature schemes with constant costs for signing and verifying. *IEICE transactions on fundamentals of electronics, communications and computer sciences*, pages 50–62, 2010.
21. L. Nguyen. Accumulators from bilinear pairings and applications. In *CT-RSA 2005*, pages 275–292.
22. T. P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *CRYPTO 1992*, pages 129–140.