Credential Revocation Assisted by a Covertly Corrupted Server

Alisa Pankova[1] and Jelizaveta Vakarjuk[1,2]
{alisa.pankova, jelizaveta.vakarjuk}@cyber.ee

[1] Cybernetica AS, Mäealuse 2/1, 12618 Tallinn, Estonia
[2] Tallinn University of Technology, Akadeemia tee 15a, 12618, Tallinn, Estonia

Abstract. European Digital Identity (EUDI) Wallet aims to provide end users with a way to get attested credentials from issuers, and present them to different relying parties. An important property mentioned in the regulatory frameworks is the possibility to revoke a previously issued credential. While it is possible to issue a short-lived credential, in some cases it may be inconvenient, and a separate revocation service which allows to revoke a credential at any time may be necessary.
In this work, we propose a full end-to-end description of a generic credential revocation system, which technically relies on a single server and secure transmission channels between parties. We prove security of the proposed revocation functionality in the universal composability model, and estimate its efficiency based on a proof-of-concept implementation.

Keywords: attribute-based credentials · revocation · universal composability · accountability · covert adversary

1 Introduction

European regulation eIDAS2 (electronic IDentification, Authentication and trust Services) [1] defines the concept of European Digital Identity (EUDI) Wallet. It aims to provide end users with a way to get verifiable credentials from issuers and present them to different relying parties. The solutions compatible with EUDI should be interoperable between the EU member states. Architecture and Reference Framework (ARF)[3] defines a set of common standards, technical specifications, common guidelines and best practices for developing such wallets.

An important property of a wallet is the possibility to *revoke* a credential. According to the ARF version 2.5.0, the issuer should include verifiable revocation information in the credential if it is valid for longer than 24 hours. Once a credential is revoked, it cannot transition back to valid. While a short-lived credential is easy to implement, it has its own disadvantages, like the necessity to continuously issue an updated credential. Additionally (albeit not specified in ARF 2.5.0), if the user wants to be in full control of their own credentials, they

[3] https://github.com/eu-digital-identity-wallet/eudi-doc-architecture-and-reference-framework, last visited 7th October, 2025

should have the possibility to revoke their credential at any time, e.g. whenever key leakage occurs, even if the issuer is not available. Moreover, the the fact of leakage can be sensitive information that the user does not want to reveal to the issuer.

In this work, we propose a full end-to-end description of a generic credential revocation system, which can potentially be applied to an arbitrary credential system, as it is dealing not directly with a credential, but a special *revocation token*, which can be treated as a special attribute of a credential. It allows to give revocation rights to a chosen party (letting it know a particular secret), and no other party (except the credential user) will be able to learn whether the credential has been revoked or not, unless the fact of revocation has been explicitly disproved by the user. The system technically relies on a single (possibly corrupted) server and secure transmission channels between parties. It does not use building blocks like broadcast channels or blockchains, which could potentially make implementation complicated in practice. Avoiding these constructions inevitably puts some restrictions, as there is no way to "enforce" revocation server to respond and thus avoid intentional denial of service. Nevertheless, the server is held *accountable* for what it is doing.

We have chosen centralised server solution for the following reasons:

1. We believe that in practice it is easier to have a single centralised revocation service than letting every party with revoking rights (e.g. the issuers) constantly run their own service.
2. We want that the user would be able to revoke their credential without needing to contact a particular issuer. Note that revocation means that *no verifier will accept the revoked credential*, and for this the user cannot just mark their credential locally as revoked.
3. We want that the party with revoking rights would not be able to *unrevoke* a credential, so we do not want to let the party with revoking rights to fully control the revocation process. This is especially important in the case where the revoker is corrupted *after* it has revoked a token. A centralised server ensures that the adversary needs to corrupt *both* the revoking party and the server to nullify a revocation.
4. While a solution based on secure multiparty computation and/or blockchain would provide better guarantees in terms of integrity, a single server solution is cheaper and easier to deploy.

We prove security of the proposed functionality in the universal composability model, and estimate its efficiency based on a proof-of-concept implementation. The paper is outlined as follows. In Sec. 2, we give technical background of the UC framework, describe the threat model, and also discuss some related work on credential revocation. In Sec. 3, we provide an *accountable transmission* functionality, which we will be using as a building block. In Sec. 4, we present an ideal revocation functionality as well as the protocol that implements it. We compare our solution with the most relevant related work in Sec. 5, report efficiency in Sec. 6, and conclude in Sec. 7.

2 Background and Related Work

2.1 Universal Composability

Universal composability (UC) [13] framework allows to reason about protocols as building blocks, proving their security even if multiple instances of the same protocol are used multiple times, and in combination with any other protocols.

In this framework, the functional and security requirements of a system are defined in the form of ideal functionality $\mathcal{F}$, describing how the system *should* work. A protocol Π describing how the system *actually* works is running between parties (modelled as Interactive Turing Machine Instances) over a network controlled by an adversary $\mathcal{A}$, who can corrupt some of these parties. UC follows the real-ideal world paradigm, proving that, for any attack against Π (the real world), there exists a corresponding attack against $\mathcal{F}$ (the ideal world).

There is an environment $\mathcal{Z}$, representing the whole world outside of the observed protocol instance, including its other instances. $\mathcal{Z}$ provides inputs to all (honest) parties and receives their outputs. It also communicates with the adversary $\mathcal{A}$ attacking the protocol. In the ideal world, parties are viewed as *dummy parties* who forward their inputs directly to the ideal functionality $\mathcal{F}$ that internally executes the main task of the protocol while allowing some interaction with the adversary to represent "intentional" weaknesses of the system. The protocol Π securely realises functionality $\mathcal{F}$ if, for every real adversary $\mathcal{A}$ attacking the protocol Π, there exists an ideal adversary $\mathcal{A}_S$ interacting with $\mathcal{F}$, such that no environment $\mathcal{Z}$ can distinguish whether it is interacting with Π running in parallel with $\mathcal{A}$, or with $\mathcal{F}$ running in parallel with $\mathcal{A}_S$.

Corruptions. In the real protocol, the adversary $\mathcal{A}$ may corrupt any party P by sending to it a special message $(\mathsf{corrupt}, P)$. In the ideal world, $\mathcal{A}_S$ sends $(\mathsf{corrupt}, P)$ directly to $\mathcal{F}$. The functionality $\mathcal{F}$ maintains a set of corrupted parties $\mathcal{C}$, and its behaviour may depend on the contents of $\mathcal{C}$. The adversary $\mathcal{A}_S$ gets control over the inputs and outputs of a corrupted party in $\mathcal{F}$.

Accountable UC model In our security model, we allow the server to cheat, but cheating should be accountable, and can be proven to a third party. Accountable UC (AUC) [21] proposes a way to extend any ideal functionality to be accountable. In this work, we will not be using AUC directly, as we do not need full accountability for every party, but only for the server. However, we are following the presentation of AUC, explicitly defining the property that the adversary may break, and the accountability w.r.t. this property. The ideal revocation functionality will explicitly allow cheating at a certain places, at the same time recording such cheating and providing an interface that can convince a special *judge* party that server has cheated.

2.2 Types of Corruption and Threat Model

A corrupted party is usually treated either as *passively* corrupted (honest-but-curious), or *actively* corrupted (may deviate from the prescribed routines). Ad-

ditionally, a practical notion of *covertly* corrupted party has been proposed by Aumann and Lindell [2]. Intuitively, it means that the party will not cheat if it will be caught with a certain non-negligible probability. Such a definition is suitable for the server, which cares for its reputation, but could still apply silent undetected attacks on user privacy. In this paper, we assume a variant of covert adversary who will not cheat if it is possible to prove the fact of cheating to a third party, although we do not explicitly define the *probability* that honest parties will run a cheating detection procedure.

Another way to classify the corruptions are *static* and *dynamic* corruptions. In the case of static corruption, the party is chosen to be either honest or corrupted from the beginning. In the case of dynamic corruption, the adversary may choose to corrupt the party later.

In this work, we have the **user**, the **revoker**, the **server**, the **verifier**, and the **judge** (the third party to whom the fact of cheating can be proven). We assume that these parties may have the following corruptions:

- The user, the revoker, and the verifier, can be *dynamically actively* corrupted.
- The server can be *statically covertly* corrupted.
  This means that any cheating by the server should be accountable.
- The judge can be *statically actively* corrupted.
  That is, we ensure that accountability procedure does not leak any secrets to the judge, even if the judge misbehaves.

2.3 Credential Revocation

In [5] (2024), Carsten Baum et.al. have given a feedback to the European Digital Identity ARF regulation, pointing out important properties that a system should achieve by law, where among other properties they mention revocation. It is acknowledged to be a hard problem in practice, and it is proposed to use short-live credentials as an alternative, letting credential contain expiration date as one of the attributes. The issuer should generate credential update info for all the valid credentials before the end of the credential's lifetime is reached. As mentioned in U-Prove cryptographic specification [33], issuers can encode the validity period into U-Prove tokens, or a verifier *may* require that a fresh U-Prove token is obtained in real time from the issuer. Similar holds for the mDL [25] specification. This approach, however, requires continuous interaction with issuer, and does not allow to *immediately* revoke a credential if needed. In some works [30, 34], revocation has been left as future work.

An overview of different revocation strategies has been given in [26]. An alternative solution to a short-live credential is to maintain a special revocation status for a credential. It can be realised using signature lists of active/revoked credentials on the issuer side (or a special designated revocation party). This solution can be improved by using cryptographic accumulators, inducing overhead on the user side, and verifier-local revocation (VLR [8]), inducing overhead on the verifier side. Surveys of various types of accumulators and their use in the context of revocation of anonymous credentials has been given by Ren et.al. [35]

and Ozcelik et.al. [32]. IRMA[4] (implementation of Idemix [10] attribute-based credential scheme) is using an RSA-based accumulator of [4].

Blockchain-based solutions like [36] are assuming a consistent append-only log, which can be very useful for communicating updates, including revocations. There exists an implementation of accountable bulletin board (BB) [22] based on hyperledger Fabric [20], that is universally composable, and hence could potentially be used as a building block by a revocation protocol.

In this work, we are are assuming that there is just a single server available. Such a setting has also been considered [24] and [18], where a public revocation database is maintained by a special *Revocation Authority* (RA). An important property that these works achieve is simplicity of underlying cryptography, as the user will likely have a constrained device like a smart card. While RA does not compromise user privacy, it needs to be trusted in cooperation and not revoking arbitrary credentials on its own. Indeed, assuming a fully corrupted RA, some features of the system would be less efficient, or even be impossible, as we cannot guarantee that a single server will not "forget" a revocation.

Probably the closest to ours is the revocation scheme by Lueks et. al. [27]. In their scheme, they link a credential to a special *revocation token*, which can be in turned linked to a credential. The scheme allows unlinkability between presentations, using fresh randomness for each presentation. It assumes a semi-trusted (honest-but-curious) revocation server.

We follow the line of [24, 18, 27], aiming for a central server that assists in revocation of credentials by different types of parties. However, we assume that a server can be *actively* corrupted, and provide cheating detection and accountability procedures, which allow to prove the fact of cheating to a third party. We propose a UC functionality that captures the properties we want to obtain from revocation. We then propose a protocol that UC-realises proposed functionality in Random Oracle (RO) model and estimate its efficiency. While our revocation protocol in its initial form does not cover *unlinkability of presentations*, this property can be plugged in using ideas from [27], as discussed in Sec. 4.3.

2.4 Building Block Functionalities

In this section, we list UC functionalities used as building blocks by our revocation mechanism. We describe these functionalities on a higher level, explaining what their routines are doing, but excluding details of their interaction with the adversary. We leave their formal description to Sec. A.2.

Signing. For the purpose of accountability, our protocol will need signatures. For this, we use certification functionality $\mathcal{F}_{\mathsf{cert}}$ from Canetti [12]. In this functionality, a signature is bound directly to the *identity* of the signer. On a higher level, $\mathcal{F}_{\mathsf{cert}}$ (bound to a signer party S) lists the following routines:

Signature Generation. On input $(\mathsf{sign}, \mathsf{sid}, m)$ from the signer party S, generate a signature σ and output it to S. Record the entry (m, σ).

Signature Verification. On input $(\mathsf{verify}, \mathsf{sid}, m, \sigma)$ from any party P:

[4] https://docs.yivi.app/what-is-yivi/, last visited 7th October, 2025

– If (m, σ) is recorded, output 1.
– Otherwise, output 0.

Our revocation protocol will use $\mathcal{F}_{\mathsf{cert}}$ as an internal subroutine, meaning that we assume special signing keys that are used *only* inside the revocation protocol. We emphasise that otherwise our protocol would *not* be secure in terms of UC, as we would have no guarantees that the signing key will not be used by some external application, where the server can be fooled to sign an arbitrary claim.

Non-interactive zero-knowledge proofs. Zero-knowledge proofs allow one party (the prover) to convince some other party (the verifier) that some statement is true, without leaking the *witness* of this statement, i.e. *why* the statement is true. The statement is formally represented as a relation $\mathcal{R}(x, w) \in \{0, 1\}$ for some public x (known to the verifier) and secret w (known only to the prover). Non-interactive zero-knowledge proofs (NIZK) allow the prover to prepare the proof in advance, before contacting the verifier.

We use functionality by Lysyanskaya and Rosenbloom in [28] as it technically seems the most suitable for our revocation mechanism. Since we are using malleable NIZK proofs in the actual implementation, we append to it the property of *malleability* as it was done by Bobolz et. al. in [6]. Malleability means that a proof π can be converted to a different proof π' of the same statement. Such NIZK functionality is called *weak*, denoted $\mathcal{F}_{\mathsf{wNIZK}}$ (with w in the subscript). On a higher level, $\mathcal{F}_{\mathsf{wNIZK}}$ (bound to a prover party P) lists the following routines:

Prove. On input $(\mathsf{Prove}, \mathsf{sid}, x, w)$ from P, if $\mathcal{R}(x, w) = 1$, compute a proof π. Record the entry (x, π). Output π to P.

Verify. On input $(\mathsf{Verify}, \mathsf{sid}, x, \pi)$ from a verifier V:

– If (x, π) is recorded, output 1.
– Otherwise, if (x, π') has been recorded for another proof π' , record (x, π) and output 1. This means that the proof π is accepted even it has not been generated using $\mathcal{F}_{\mathsf{wNIZK}}$, e.g. has been mauled from a valid proof π. This point would be omitted for a "strong" NIZK functionality.
– Otherwise, output 0.

We note that proof malleability is *not* needed for our protocol, and is only introduced due to using malleable proofs in the protocol implementation, so a non-malleable NIZK functionality would be suitable as well.

3 Accountable Message Transmission

To make server accountable for messages that it has sent, we will need an accountable message transmission functionality, used as a building block. The functionality $\mathcal{F}_{\mathsf{accSMT}}$ is based on secure transmission functionality $\mathcal{F}_{\mathsf{SMT}}$ from [13]. It is defined for a single sender party, but there may be any number of receiver parties. The communication is accountable w.r.t. the sender. This means that (1) the receiver is able to prove that a message has been sent by S and (2) two receivers are able to prove who of them has received the message first.

Send and Receive interfaces are analogous of those of $\mathcal{F}_{\mathsf{SMT}}$ of [13]. The adversary may corrupt the sender, which allows it to modify the message, as well as the receiver identity. If th sender is honest, the adversary only learns the leakage $l(m)$ of the true message m, e.g. message length.

Prove transmission interface allows a receivers R to convince a third party J that R has received m from S at some point.

Prove transmission order interface allows two receivers R_1 and R_2 to convince a third party J that R_1 has received m_1 before R_2 has received m_2. It is possible that R_1 has received another copy m_1 also *later*, and that R_2 received another copy of m_2 *earlier*, but this fact is not relevant. E.g. if the sequence of received messages is $(R_2 : m_2), (R_1 : m_1), (R_2 : m_2), (R_1 : m_1)$, the output should be true, as there exists (R_1, m_1) received earlier than (R_2, m_2).

In Sec. B, we describe a protocol that UC-realises $\mathcal{F}_{\mathsf{accSMT}}$, based on signature and global clock functionalities as building blocks. Accountability is achieved by letting the sender provide a signature on every message, together with the timestamp on which the message was sent. The receiver checks whether the signed timestamp is close enough to the actual receival time. The trick is that imprecision of clocks allows the sender to cheat if only a little time has passed between the two message receivals. We will show that it can still be tolerated by the revocation protocol that will use $\mathcal{F}_{\mathsf{accSMT}}$.

4 Revocation Mechanism

We are now ready to describe the revocation mechanism. In our approach, like in [27], we assume that a credential has a designated attribute, which stores a special *revocation token*. We propose a functionality for revoking such *tokens*. A credential that is linked to a revoked token is also considered revoked.

4.1 Involved Parties and Requirements

We want that an issued token could be revoked by the user, the issuer, or a special revocation authority. While we could assume that any party should contact the issuer for revocation, we think that in practice the user and the revocation authority may want to act faster even if the issuer is (temporarily) unavailable. Considering multiple revoking parties, it is important that a party cannot later "unrevoke" a token. Moreover, a token cannot be "unrevoked" even by the party that revoked it. The latter is important for a user who is afraid that its secrets leaked to an adversary. This discussion leads to the following list of requirements:

1. A user U is able to create a revocation token x, which can later be revoked by a revoker party R (it is allowed that $U = R$).
2. The user U has the possibility to check if its token has been revoked by R.
3. Neither the server nor any other party $P \neq R$ should learn whether x has been revoked, unless U explicitly proves the fact of (non)revocation.
4. It is not possible to "unrevoke" a token that has been revoked once.

Send. On input $(\mathsf{send}, \mathsf{sid}, R, \mathsf{msg})$ from the party S,
send $(\mathsf{send\text{-}req}, \mathsf{sid}, R, \mathsf{msg})$ to $\mathcal{A}_S$.
Receive. On input $(\mathsf{receive}, \mathsf{sid})$ from any party R:

- Send $(\mathsf{receive\text{-}req}, \mathsf{sid})$ from $\mathcal{A}_S$.
- Wait for $(\mathsf{receive\text{-}ok}, \mathsf{sid}, \mathsf{msg}')$ from $\mathcal{A}_S$.
- If S is corrupted, output $(\mathsf{sent}, \mathsf{sid}, \mathsf{msg}')$ to R'.
- Otherwise, if not yet generated output for sid, and $(\mathsf{send}, \mathsf{sid}, R, \mathsf{msg})$ has been input by S, then output $(\mathsf{sent}, \mathsf{sid}, \mathsf{msg})$ to R.
- Otherwise, output $(\mathsf{sent}, \mathsf{sid}, \perp)$ to R.

Prove transmission. On input $(\mathsf{prove\text{-}sent}, \mathsf{sid}, \mathsf{sid}')$ from R, and $(\mathsf{prove\text{-}sent}, \mathsf{sid})$ from J:

1. If $R, J \notin \mathcal{C}$, take msg such that $(\mathsf{sent}, \mathsf{sid}', \mathsf{msg})$ has been sent to R.
   If there is no such msg, output $(\perp, \mathsf{sid})$ to J.
2. Otherwise:
   - Send $(\mathsf{prove\text{-}sent\text{-}req}, \mathsf{sid}, \mathsf{sid}')$ to $\mathcal{A}_S$.
   - Wait for $(\mathsf{prove\text{-}sent\text{-}ok}, \mathsf{sid}, \mathsf{msg})$ from $\mathcal{A}_S$.

   If $S \notin \mathcal{C}$, verify that $(\mathsf{send}, \mathsf{sid}', R, \mathsf{msg})$ has been sent by S.
   If this check fails, output $(\perp, \mathsf{sid})$ to J.
3. Output $((R, \mathsf{msg}), \mathsf{sid})$ to J.

Prove transmission order. On input $(\mathsf{prove\text{-}order\text{-}1}, \mathsf{sid}, \mathsf{sid}_1)$ from R_1, $(\mathsf{prove\text{-}order\text{-}2}, \mathsf{sid}, \mathsf{sid}_2)$ from R_2, and $(\mathsf{prove\text{-}order}, \mathsf{sid})$ from J:

1. If $R_1, R_2, J \notin \mathcal{C}$, take msg_1 and msg_2 such that $(\mathsf{sent}, \mathsf{sid}_1, \mathsf{msg}_1)$ has been sent to R_1 before $(\mathsf{sent}, \mathsf{sid}_2, \mathsf{msg}_2)$ has been sent to R_2.
   If there are no such messages, output $(\perp, \mathsf{sid})$ to J.
2. Otherwise:
   - Send $(\mathsf{prove\text{-}order\text{-}req}, \mathsf{sid}, \mathsf{sid}_1, \mathsf{sid}_2)$ to $\mathcal{A}_S$.
   - Wait for $(\mathsf{prove\text{-}order\text{-}ok}, \mathsf{sid}, \mathsf{msg}_1, \mathsf{msg}_2)$ from $\mathcal{A}_S$.

   If $S \notin \mathcal{C}$, verify that $(\mathsf{send}, \mathsf{sid}_1, R_1, \mathsf{msg}_1)$ has been sent to R_1 before $(\mathsf{sent}, \mathsf{sid}_2, R_2, \mathsf{msg}_2)$ has been sent to R_2.
   If this check fails, output $(\perp, \mathsf{sid})$ to J.
3. Output $((R_1, R_2, \mathsf{msg}_1, \mathsf{msg}_2), \mathsf{sid})$ to J.

Corrupt. Upon receiving $(\mathsf{corrupt}, P)$ from $\mathcal{A}_S$, add P to $\mathcal{C}$. Send to $\mathcal{A}_S$ all the values input by P and output to P so far. From this point on, forward to $\mathcal{A}_S$ all messages that P receives from $\mathcal{Z}$ and $\mathcal{F}_{\mathsf{rev}}$, and let $\mathcal{A}_S$ choose messages that P sends to $\mathcal{Z}$ and $\mathcal{F}_{\mathsf{rev}}$.

Fig. 1. Accountable secure message transmission functionality $\mathcal{F}_{\mathsf{accSMT}}$, parametrised with a sender identity S.

We want that any cheating by S would be detectable and provable to a third party J. We note that cheating will *not* be reported *immediately*, as it requires communication between the verifier and the revoker (without a bulletin board functionality, it is not possible for the verifier to check whether the server has silenced any legitimate revocation). Cheating detection is modelled as a separate call to $\mathcal{F}_{\mathsf{rev}}$.

4.2 The Ideal Revocation Functionality

Figure 2 depicts an ideal functionality $\mathcal{F}_{\mathsf{rev}}$ for issuing and revoking special *revocation tokens*. We have the party S (the revocation server) whose approval is needed for the main routines of $\mathcal{F}_{\mathsf{rev}}$ and whose corruption allows adversary to cheat, the party U (the user) who gets the token and the right to query its revocation status, the party R (the revoker) who also gets the token and may revoke it later, and the party V (the verifier) who learns whether the token provided by U is revoked or not. There can be many users, revokers, and verifiers.

Any party be corrupted. For each subroutine f, there is a *leakage function* $\mathcal{L}_f^{\mathcal{C}}$, which outputs the values leaked to the adversary, assuming the set $\mathcal{C}$ of corrupted parties. The leakage may depend on the inputs of parties and the internal state of $\mathcal{F}_{\mathsf{rev}}$. The exact definition of $\mathcal{L}_f^{\mathcal{C}}$ depends on the protocol UC-realising $\mathcal{F}_{\mathsf{rev}}$.

Initialisation. We let the adversary choose the universal set $\mathcal{X}$ of tokens. Our goal is not to conceal the revocation token x itself, but to achieve unlinkability between issued, revoked, and presented x. Issuing starts failing after having used up all tokens of $\mathcal{X}$, but the size of $\mathcal{X}$ is controlled by a parameter N of $\mathcal{F}_{\mathsf{rev}}$. The adversary also provides the class of functions $\mathcal{V}$ that can be used to make additional checks on the presented token.

Issuing. $\mathcal{F}_{\mathsf{rev}}$ samples a token x from $\mathcal{X}$ and outputs it to the user and the revoker. The token x may then be (outside of $\mathcal{F}_{\mathsf{rev}}$) included into a credential. The update $\mathcal{X} \leftarrow \mathcal{X} \setminus \{x\}$ ensures that an honest user always gets a new token, even if the set $\mathcal{X}$ is small. We allow the adversary to choose a token $\overline{x}$ for a corrupted U. However, $\mathcal{F}_{\mathsf{rev}}$ does not allow to reuse tokens of honest users.

Revocation. A party R is only allowed to revoke x if $\mathcal{F}_{\mathsf{rev}}$ has previously declared R as a revoker of x. However, a corrupted revoker R may share revocation rights with another corrupted party R'.

User notification. The user U may ask from $\mathcal{F}_{\mathsf{rev}}$ whether a particular token of U has been revoked.

Verifier notification. Differently from the user, the verifier V is not allowed to learn which tokens have been revoked, and the functionality does not return anything to V. Instead, it only fixes for V the current state X_V^- of the revocation database for future reference. While such notification could be triggered in the beginning of verification, we need it as a separate routine to be able to UC-realise protocols that allow delayed updates on the verifier side.

In both types of notification, a corrupted S may decide to silence some revocations, represented by a set of silenced revocation sessions S_{ID}. However, for any silenced revocation, the consistency property between V (U) and the revoker R should be broken ($\mathsf{brokenCons}(V, R) = \mathsf{true}$), which is accountable.

Verification. The user U provides the token x as an input. The verifier V provides a function $v \in \mathcal{V}$ that x should satisfy. $\mathcal{F}_{\mathsf{rev}}$ checks the following:

- x does not belong to the set X_V^- of revoked tokens previously notified to V;
- $v(x) = \mathsf{true}$, meaning that x satisfies additonal requirements of V.

Intuitively, v links the token to a credential. For example, presented x could be compared against credential token x', or equality between x and the token contained in a credential presentation c proven in zero-knowledge. Information about x' or c would be included into the definition of v.

Detect cheating. The verifier V and the revoker R may check whether S has silenced any revocations by R. Moreover, they can prove it to a third party J. If the adversary has cheated during the verifier notification, J will get an output $\mathsf{dis}(S)$, meaning that S is guilty. This interface assumes additional input from S. In practice, if S refuses to provide its input, it should be automatically considered guilty.

Unfairness. We allow the adversary to delay or interrupt the output to an honest party even if $\mathcal{F}_{\mathsf{rev}}$ has finished successfully and provided output to another party. This is because, in a real protocol, we cannot assume reliable channels at least between the server and the other parties, and the adversary may interrupt the connection at any moment. The internal state of $\mathcal{F}_{\mathsf{rev}}$ changes with the server S receiving the output, but every other party needs a special permission from adversary to receive the output. The impact of unfairness depends on the way in which environment will use $\mathcal{F}_{\mathsf{rev}}$.

Corruptions. During initialisation, the adversary may send a special corruption message to $\mathcal{F}_{\mathsf{rev}}$ to corrupt any party. For all parties except the server S and the judge J, corruptions are allowed also *after* initialisations, leaking to the adversary all the values input and output to that party so far. Whenever a user gets corrupted, $\mathcal{F}_{\mathsf{rev}}$ leaks also all the tokens revoked for that user so far.

4.3 The Real Revocation Protocol

Protocol description. We define a set of real protocol subroutines which we denote Π_{rev}. Protocol participants communicate via secure authenticated channels, modelled as functionality $\mathcal{F}_{\mathsf{SMT}}$ from [13]. The protocol is parametrised by a deterrence factor $0 < p < 1$ (i.e. probability of cheating detection), where $p = 1$ corresponds to an *actively* corrupted server, and $p = 0$ to a *passively* corrupted server. The notation used in the protocol description is summarised in Table 1.

Let $\mathcal{R}_S$ and $\mathcal{R}_R$ be sets of bit strings of certain length (which will depend on the security parameter). We assume a family $\mathcal{H}$ of hash functions $H : \mathcal{R}_S \times \mathcal{R}_R \to \mathcal{R}_R$ that act as a random oracle (RO). We use functionality $\mathcal{F}_{\mathsf{wNIZK}}$ from [28] for generating and verifying NIZK (Non-Interactive Zero-knowledge) proofs for relation $R((x, x', c), (w, w')) \Leftrightarrow (H(c, w) = x) \wedge (H(w', w) = x')$, for constant $c \in \mathcal{R}_S$, witness $w' \in \mathcal{R}_S$, witness $w \in \mathcal{R}_R$, and instance x, x'. We use the functionality $\mathcal{F}_{\mathsf{cert}}$ from [12] for generating and verifying signatures.

$\mathcal{F}_{\mathsf{rev}}$ is parametrised by the total number of tokens N.
If any assertion or parsing inside $\mathcal{F}_{\mathsf{rev}}$ fails, it outputs $\perp$ to all involved parties.

Initialisation. On input $(\mathsf{init}, \mathcal{X}, \mathcal{V})$ from $\mathcal{A}_S$:

- Check that $|\mathcal{X}| \geq N$. Store $\mathcal{X}$ and $\mathcal{V}$.
- Initialise an empty map of revoked tokens $X^- : \mathcal{S}_{ID} \to (\mathcal{X}, \mathcal{P})$.
- For all $x \in \mathcal{X}$, initialise empty sets $\mathcal{R}_x$ and $\mathcal{U}_x$.

Let the adversary corrupt any parties, adding them to a set $\mathcal{C}$.

Issuing. On input $(\mathsf{issue}, \mathsf{sid}, R)$ from party U and $(\mathsf{issue}, \mathsf{sid}, U)$ from party R:

1. Let $L = \mathcal{L}^{\mathcal{C}}_{issue}(U, R, x)$ for $x \leftarrow\!\!\$\ \mathcal{X}$. Send $(\mathsf{issue\text{-}req}, \mathsf{sid}, L)$ to $\mathcal{A}_S$.
2. Receive $(\mathsf{issue\text{-}req\text{-}ok}, \mathsf{sid}, \overline{x})$ from $\mathcal{A}_S$. If $U \in \mathcal{C}$, take $x \leftarrow \overline{x}$.
3. Assert that there is no $U' \notin \mathcal{C}$ such that $U' \in \mathcal{U}_x$.
   Update $\mathcal{U}_x \leftarrow \mathcal{U}_x \cup \{U\}$, $\mathcal{R}_x \leftarrow \mathcal{R}_x \cup \{R\}$, $\mathcal{X} \leftarrow \mathcal{X} \setminus \{x\}$.
4. Output $(\mathsf{issue\text{-}resp\text{-}u}, \mathsf{sid}, x)$ to U. Output $(\mathsf{issue\text{-}resp\text{-}r}, \mathsf{sid}, x)$ to R.

Revocation. On input $(\mathsf{revoke}, \mathsf{sid}, x)$ from a party R, and $(\mathsf{revoke}, \mathsf{sid})$ from S:

1. Let $L = \mathcal{L}^{\mathcal{C}}_{revoke}(R, x)$. Send $(\mathsf{revoke\text{-}req}, \mathsf{sid}, L)$ to $\mathcal{A}_S$.
2. Receive $(\mathsf{revoke\text{-}req\text{-}ok}, \mathsf{sid})$ from $\mathcal{A}_S$.
3. If $R \in \mathcal{R}_x$ or $\exists R' \in \mathcal{R}_x \cap \mathcal{C}$, assign $X^-[\mathsf{sid}] = (x, R)$.
4. Output $(\mathsf{revoke\text{-}resp\text{-}s}, \mathsf{sid}, \mathsf{ok})$ to S. Output $(\mathsf{revoke\text{-}resp\text{-}r}, \mathsf{sid}, \mathsf{ok})$ to R.

User notification. On input $(\mathsf{unotify}, \mathsf{sid}, x)$ from U and $(\mathsf{unotify}, \mathsf{sid})$ from S:

1. Let $L = \mathcal{L}^{\mathcal{C}}_{unotify}(U, x)$. Let $X \leftarrow \emptyset$. Send $(\mathsf{unotify\text{-}req}, \mathsf{sid}, L)$ to $\mathcal{A}_S$.
2. Receive $(\mathsf{unotify\text{-}req\text{-}ok}, \mathsf{sid}, S_{ID})$ from $\mathcal{A}_S$.
   If $S \in \mathcal{C}$, take $X \leftarrow \big\{x \mid (x, R) \in X^-[S_{ID}], \mathsf{brokenCons}(U, R) = \mathsf{true}\big\}$.
3. Let $b = \mathsf{true}$ iff $U \in \mathcal{U}_x$ and $x \in X^- \setminus X$.
4. Output $(\mathsf{unotify\text{-}resp\text{-}s}, \mathsf{sid}, \mathsf{ok})$ to S. Output $(\mathsf{unotify\text{-}resp\text{-}u}, \mathsf{sid}, b)$ to U.

Verifier notification. On input $(\mathsf{vnotify}, \mathsf{sid})$ from V and $(\mathsf{vnotify}, \mathsf{sid})$ from S:

1. Let $L = \mathcal{L}^{\mathcal{C}}_{vnotify}(V)$. Let $X \leftarrow \emptyset$. Send $(\mathsf{vnotify\text{-}req}, \mathsf{sid}, L)$ to $\mathcal{A}_S$.
2. Receive $(\mathsf{vnotify\text{-}req\text{-}ok}, \mathsf{sid}, S_{ID})$ from $\mathcal{A}_S$.
   If $S \in \mathcal{C}$, take $X \leftarrow \big\{x \mid (x, R) \in X^-[S_{ID}], \mathsf{brokenCons}(V, R) = \mathsf{true}\big\}$.
3. Assign $X^-_V \leftarrow X^- \setminus X$.
4. Output $(\mathsf{vnotify\text{-}resp\text{-}s}, \mathsf{sid}, \mathsf{ok})$ to S. Output $(\mathsf{vnotify\text{-}resp\text{-}v}, \mathsf{sid}, \mathsf{ok})$ to V.

Verification. On input $(\mathsf{verify}, \mathsf{sid}, x, V)$ from U, and $(\mathsf{verify}, \mathsf{sid}, v)$ from V:

1. Let $L = \mathcal{L}^{\mathcal{C}}_{verify}(U, V, x)$. Send $(\mathsf{verify\text{-}req}, \mathsf{sid}, L)$ to $\mathcal{A}_S$.
2. Receive $(\mathsf{verify\text{-}req\text{-}ok}, \mathsf{sid})$ from $\mathcal{A}_S$.
3. Assert $x \notin X^-_V$, $v \in \mathcal{V}$, and $v(x) = \mathsf{true}$.
4. Output $(\mathsf{verify\text{-}resp\text{-}v}, \mathsf{sid}, \mathsf{ok})$ to V. Output $(\mathsf{verify\text{-}resp\text{-}u}, \mathsf{sid}, \mathsf{ok})$ to U.

Fig. 2. Ideal functionality $\mathcal{F}_{\mathsf{rev}}$ (Part 1). For each subroutine f, the leakage function $\mathcal{L}^{\mathcal{C}}_f$ outputs the values leaked to the adversary assuming the set $\mathcal{C}$ of corrupted parties.

Break consistency. On input $(\mathsf{brokenCons}, (V, R))$ from $\mathcal{A}_S$,
if $S \in \mathcal{C}$, set $\mathsf{brokenCons}(V, R) = \mathsf{true}$.
Detect cheating. Upon receiving $(\mathsf{cheat\text{-}detect}, \mathsf{sid}, \mathsf{sid}_P)$ from $P \in \{U, V\}$, $(\mathsf{cheat\text{-}detect}, \mathsf{sid}, \mathsf{sid}_R)$ from R, and $(\mathsf{cheat\text{-}detect}, \mathsf{sid})$ from S and J:

1. If at least one involved party P, R, S, J is corrupted:
   - Send $(\mathsf{dectect\text{-}cheat\text{-}req}, \mathsf{sid}, \mathsf{sid}_P, \mathsf{sid}_R)$ to $\mathcal{A}_S$.
   - Wait for $(\mathsf{dectect\text{-}cheat\text{-}ok}, \mathsf{sid}, \mathcal{C}')$ from $\mathcal{A}_S$ with $\mathcal{C}' \subseteq \mathcal{C}$.

   Otherwise, set $\mathcal{C}' = \emptyset$.
2. If $\mathsf{brokenCons}(V, R) = \mathsf{true}$ and $V, R \notin \mathcal{C}$, output $(\mathsf{verdict}, \mathsf{sid}, \mathsf{dis}(\{S\} \cup \mathcal{C}'))$ to J.
   Otherwise, output $(\mathsf{verdict}, \mathsf{sid}, \mathsf{dis}(\mathcal{C}'))$ to J.

Unfairness. Whenever a message of the form $(response, \mathsf{sid}, y)$ for $response \neq \mathsf{verdict}$ is to be output to an honest party $P \neq S$, send $(response, \mathsf{sid}, y)$ to $\mathcal{A}_S$. Wait for $(response\text{-}\mathsf{ok}, \mathsf{sid})$ from $\mathcal{A}_S$ and send $(response, \mathsf{sid}, y)$ to P.

Corrupt. Upon receiving $(\mathsf{corrupt}, P)$ from $\mathcal{A}_S$, if $P \notin \{S, J\}$, add P to $\mathcal{C}$. Send to $\mathcal{A}_S$ all the values input by P and output to P so far, and also the set $\{x : P \in \mathcal{U}_x \wedge (\mathsf{revoke}, \mathsf{sid}, x)\text{was input to } \mathcal{F}_{\mathsf{rev}}\}$. From this point on, forward to $\mathcal{A}_S$ all messages that P receives from $\mathcal{Z}$ and $\mathcal{F}_{\mathsf{rev}}$, and let $\mathcal{A}_S$ choose messages that P sends to $\mathcal{Z}$ and $\mathcal{F}_{\mathsf{rev}}$.

Fig. 3. Ideal functionality $\mathcal{F}_{\mathsf{rev}}$ (Part 2, adversary interfaces and accountability).

$\mathsf{Issue}_{\mathrm{U}}(\mathsf{sid}, R)$		$\mathsf{Issue}_{\mathrm{R}}(\mathsf{sid}, U)$
1 :	$s \leftarrow\!\!\$\ \mathcal{R}_S;\ r' \leftarrow\!\!\$\ \mathcal{R}_R;$	
2 :	$\Longrightarrow R : s, r'$	$U \Longrightarrow : s, r'$
3 :	$r \leftarrow H_1(U, r');$	$r \leftarrow H_1(U, r');$
4 :	$x \leftarrow H_1(s, r);$	$x \leftarrow H_1(s, r);$
5 :	$\mathsf{DB}_{\mathrm{U}}[x] \leftarrow (s, r);$	$\mathsf{DB}_{\mathrm{R}}[x] \leftarrow (s, r);$
6 :	**return** x;	**return** x;

Fig. 4. Token issuing subroutine.

Initialisation. The parties agree on hash functions $H_2, H_1 \in \mathcal{H}$ to use in this instance of $\mathcal{F}_{\mathsf{rev}}$. Each party P initialises local storages DB_{P} and $\mathsf{DB}'_{\mathrm{P}}$ for its data. The server initialises publicly readable storage $\mathsf{DB}_{\mathrm{REV}}$.

Issuing (Fig. 4). A revocation token is of the form $x = H_1(s, r)$ for some fresh randomness $s \in \mathcal{R}_S$ and $r \in \mathcal{R}_R$. The idea is that r provides protection against unauthorised revocations, and s provides privacy against the server S.

The user U generates random values s and r', and sends them to the revoker R. The value r is computed as $r \leftarrow H_1(U, r')$, treating U as an element of $\mathcal{R}_S$, so that a corrupted user cannot make honest revoker accept r of an honest user.

Notation	Explanation
$\Longrightarrow B : m$	send the message m to party B (using $\mathcal{F}_{\mathsf{SMT}}$)
$A \Longrightarrow : m$	receive the message m from party A (using $\mathcal{F}_{\mathsf{SMT}}$)
$\overset{\mathsf{acc}}{\Longrightarrow} B : m$	send the message m to party B using $\mathcal{F}_{\mathsf{accSMT}}$
$A \overset{\mathsf{acc}}{\Longrightarrow} : m$	receive the message m from party A using $\mathcal{F}_{\mathsf{accSMT}}$
DB_{P},$\mathsf{DB}'_{\mathrm{P}}$,$\mathsf{DB}^{\mathsf{acc}}_{\mathrm{P}}$	local storage of the party P
$T[k] \leftarrow v$	update the record of the table T with the key k to the value v
$v \leftarrow T[k]$	read from the table T the value v that has the key k
values (T)	read all the values from the table T
size (T)	the number of records in the table T.
$\mathsf{DB}_{\mathrm{REV}}$	publicly readable storage of the revocation server

Table 1. Notation for protocol subroutines.

Revocation (Fig. 5). Whenever a revoker R wants to revoke a token $x = H_1(s, r)$, it uploads to S the value r. S stores $r_m \leftarrow r$, where m is the upcoming number of revoked attributes. S generates new randomness s_R and sends to R the values s_R and $h_R = H_1(s_R, r)$, where h_R is sent through an accountable channel, so that the fact of revocation can be proven by R later.

We note that there is no way to *enforce* the revocation even if S is honest, unless we assume a reliable channel that the adversary cannot interrupt. However, as far as an honest R has been convinced that x has been revoked (returned ok in the end), then the fact of revocation of x is accountable.

Additionally, for all $1 \leq i \leq m$, the server S computes $h_i \leftarrow H_2(m, r_i)$, where $r_1, \ldots, r_{m-1}$ come from the previous revocations. It rewrites the contents of the pubic revocation status table $\mathsf{DB}_{\mathrm{REV}}$ to $(h_1, \ldots, h_m)$. As the result, for m revoked attributes, $\mathsf{DB}_{\mathrm{REV}}$ stores records $H_2(m, r_1), \ldots, H_2(m, r_m)$. The idea is that $\mathsf{DB}_{\mathrm{REV}}$ represents a fresh randomised variant of the revocation database DB_{S}, and the user proofs will be made against $\mathsf{DB}_{\mathrm{REV}}$. Making the entire $\mathsf{DB}_{\mathrm{REV}}$ fresh is needed to prevent verifier from using the user's proof to check whether presented x has been revoked *earlier*, or will be revoked *later*.

User notification (Fig. 6) The user U wants to check whether the token x has been revoked. It takes the randomness r that corresponds to x, and checks if there is h_i in $\mathsf{DB}_{\mathrm{REV}}$ satisfying $h_i = H_2(m, r)$. The revocation status of x and the corresponding number of revocations m will be needed for the verification.

A corrupted S may be silent about some h_i. However, as a hash of $(h_1, \ldots, h_m)$ is transmitted using $\mathcal{F}_{\mathsf{accSMT}}$, the server is accountable for every silenced token.

Verifier notification (Fig. 7) The verifier V queries the latest state $(h_1, \ldots, h_m)$ of $\mathsf{DB}_{\mathrm{REV}}$. A corrupted S may be silent about some h_i. However, as a hash of $(h_1, \ldots, h_m)$ is transmitted using $\mathcal{F}_{\mathsf{accSMT}}$, the server is accountable for every silenced token.

Verification (Fig. 8) We assume that the verifier V knows the latest state $(h_1, \ldots, h_{m_V})$ of $\mathsf{DB}_{\mathrm{REV}}$, which has been obtained from the last notification ses-

$\mathsf{Revoke}_{\mathrm{R}}(\mathsf{sid}, x)$	$\mathsf{Revoke}_{\mathrm{S}}(\mathsf{sid})$
$(\_, r) \leftarrow \mathsf{DB}_{\mathrm{R}}[x];$	$(r_1, \ldots, r_{m-1}) \leftarrow \textbf{values}\ (\mathsf{DB}_{\mathrm{S}});$
$\Longrightarrow S : r$	$R \Longrightarrow : r$
	$s_R \xleftarrow{\$} \mathcal{R}_S;$
	$h_R \leftarrow H_1(s_R, r);$
$S \Longrightarrow : s_R$	$\Longrightarrow R : s_R$
$S \overset{\mathsf{acc}}{\Longrightarrow} : h_R$	$\overset{\mathsf{acc}}{\Longrightarrow} R : h_R$
$\textbf{assert}\ h_R = H_1(s_R, r);$	$r_m \leftarrow r;$
$\mathsf{DB}^{\mathsf{acc}}_{\mathrm{R}}[\mathsf{sid}] \leftarrow (h_R, s_R, r);$	$\mathsf{DB}_{\mathrm{S}}[m] \leftarrow r_m;$
$\mathsf{DB}_{\mathrm{R}}[x] \leftarrow \bot;$	$\textbf{for}\ i = 1..m\ \textbf{do}\ h_i \leftarrow H_2(m, r_i);$
	$\mathsf{DB}_{\mathrm{REV}} \leftarrow (h_1, \ldots, h_m);$
$\textbf{return}\ \mathsf{ok};$	$\textbf{return}\ \mathsf{ok};$

Fig. 5. Token revoking subroutine.

$\mathsf{UNotify}_{\mathrm{U}}(\mathsf{sid}, x)$	$\mathsf{UNotify}_{\mathrm{S}}(\mathsf{sid})$
	$(h_1, \ldots, h_m) \leftarrow \textbf{values}\ (\mathsf{DB}_{\mathrm{REV}});$
$S \Longrightarrow : (h_1, \ldots, h_m)$	$\Longrightarrow U : (h_1, \ldots, h_m)$
$S \overset{\mathsf{acc}}{\Longrightarrow} : (h_U, m)$	$\overset{\mathsf{acc}}{\Longrightarrow} U : (h_U = H_3(h_1, \ldots, h_m), m)$
$\textbf{assert}\ H_3(h_1, \ldots, h_m) = h_U$	$\mathsf{DB}^{\mathsf{acc}}_{\mathrm{S}}[\mathsf{sid}] \leftarrow (h_U, m);$
$\mathsf{DB}^{\mathsf{acc}}_{\mathrm{U}}[\mathsf{sid}] \leftarrow (h_U, m);$	
$(s, r) \leftarrow \mathsf{DB}_{\mathrm{P}}[x];$	
$h \leftarrow H_2(m, r);$	
$b \leftarrow h \in \{h_1, \ldots, h_m\};$	
$\mathsf{DB}'_{\mathrm{U}}[x] \leftarrow (s, r, b, m);$	
$\textbf{return}\ b;$	$\textbf{return}\ \mathsf{ok};$

Fig. 6. User notification subroutine.

sion. V reports to the user U the value m_V. The user U checks that $m_V \leq m_U$, where m_U is the number of revocations from the user's last notification session. This is needed to avoid presenting a token with outdated revocation information.

For a token with randomness r, the user U computes the hash $h = H_2(m_V, r)$. It also computes a NIZK proof of knowledge of r such that $H_1(s, r) = x$ and $H_2(m_V, r) = h$. The verifier V treats x as unrevoked if (1) the NIZK proof verifies and (2) $h \neq h_i$ for all $1 \leq i \leq m_V$. It locally applies v to x to check if it is a suitable token.

Alternatively, the verifier could send to the user v, expecting the *proof* that r included in h is the same as in x for *some* x satisfying $v(x)$ (i.e. contained in the credential implicitly described in v). This, however, would require changing the relation $\mathcal{R}$ in the NIZK proofs, which would make efficiency of verification

$\mathsf{VNotify}_{\mathrm{V}}(\mathsf{sid})$		$\mathsf{VNotify}_{\mathrm{S}}(\mathsf{sid})$
1 :		$(h_1, \ldots, h_m) \leftarrow$ **values** $(\mathsf{DB}_{\mathrm{REV}})$;
2 :	$S \Longrightarrow: (h_1, \ldots, h_m)$	$\Longrightarrow V : (h_1, \ldots, h_m)$;
3 :	$S \overset{\mathsf{acc}}{\Longrightarrow}: (h_V, m)$	$\overset{\mathsf{acc}}{\Longrightarrow} V : (h_V = H_3(h_1, \ldots, h_m), m)$;
4 :	**assert** $H_3(h_1, \ldots, h_m) = h_V$	$\mathsf{DB}_{\mathrm{S}}^{\mathsf{acc}}[\mathsf{sid}] \leftarrow (h_V, m)$;
5 :	$\mathsf{DB}_{\mathrm{V}}^{\mathsf{acc}}[\mathsf{sid}] \leftarrow (h_V, m)$;	
6 :	$\mathsf{DB}_{\mathrm{V}} \leftarrow (h_1, \ldots, h_m)$;	
7 :	**return** ok;	**return** ok;

Fig. 7. Verifier notification subroutine.

dependable on the particular credential scheme. We discuss this alternative in the section *Protocol properties*.

$\mathsf{Verify}_{\mathrm{U}}(\mathsf{sid}, x, V)$		$\mathsf{Verify}_{\mathrm{V}}(\mathsf{sid}, v)$
1 :	$(s, r, b, m_U) \leftarrow \mathsf{DB}'_{\mathrm{U}}[x]$;	$(h_1, \ldots, h_{m_V}) \leftarrow$ **values** $(\mathsf{DB}_{\mathrm{V}})$;
2 :	**assert** $b = \mathsf{false}$;	$\Longrightarrow U : m_V$;
3 :	$V \Longrightarrow: m_V$;	
4 :	**assert** $m_V \leq m_U$;	
5 :	$h \leftarrow H_2(m_V, r)$;	$U \Longrightarrow: h, x, \pi$
6 :	$\pi \leftarrow \mathcal{F}_{\mathsf{wNIZK}}(\mathsf{Prove}, \mathsf{sid}, (h, x, m_V), (s, r))$	**assert** $1 = \mathcal{F}_{\mathsf{wNIZK}}(\mathsf{Verify}, \mathsf{sid}, (h, x, m_V), \pi)$
7 :	$\Longrightarrow V : h, x, \pi$	**assert** $h \notin \{h_1, \ldots, h_{m_V}\} \wedge v(x)$;
8 :	**return** ok;	**return** ok;

Fig. 8. Token verification subroutine.

Detect Cheating (Fig. 9) A revoker R together with a verifier V (or a user U) may detect cheating of S and prove it to a third party. They use proof interfaces of $\mathcal{F}_{\mathsf{accSMT}}$, proving that the revoker R has received from S an approval of revocation of some token *before* the verifier V has received the notification. They claim that S has not included the revoked token into the notification message. For this, R proves in zero knowledge (without revealing the token) which value h should have been included into the notification message. The third party J checks whether it is indeed so. The server S is contacted to get the message that should correspond to the signed hash, and if S refuses to send such a message, it is considered guilty. Alternatively, V could themselves store the log of all notifications received by it from S, but it would require a lot of storage space.

$\mathsf{CheatDetect}_{\mathrm{P}}(\mathsf{sid}, \mathsf{sid}_P)$// $\{P, P'\} \in \{V, U\}$

1 : $(h_P, m) \leftarrow \mathsf{DB}^{\mathsf{acc}}_{\mathrm{P}}[\mathsf{sid}_P]$;
2 : // Prove that revocation was before notification.
3 : $\mathsf{ok} \leftarrow \mathcal{F}_{\mathsf{accSMT}}(\mathsf{prove\text{-}order\text{-}2}, \mathsf{sid}, \mathsf{sid}_P)$
4 :
5 :
6 : $\Longrightarrow R : m$;
7 :
8 :
9 :
10 : **return** ok;

$\mathsf{CheatDetect}_{\mathrm{R}}(\mathsf{sid}, \mathsf{sid}_R)$

$(h_R, s, r) \leftarrow \mathsf{DB}^{\mathsf{acc}}_{\mathrm{R}}[\mathsf{sid}_R]$;
$\mathsf{ok} \leftarrow \mathcal{F}_{\mathsf{accSMT}}(\mathsf{prove\text{-}order\text{-}1}, \mathsf{sid}, \mathsf{sid}_R)$
// Prove which entry should have been
// included into notification message.
$P \Longrightarrow : m$;
$h \leftarrow H_2(m, r)$;
$\pi \leftarrow \mathcal{F}_{\mathsf{wNIZK}}(\mathsf{Prove}, \mathsf{sid}, (h, h_R, m), (s, r))$
$\Longrightarrow J : \pi, h$;
return ok;

$\mathsf{CheatDetect}_{\mathrm{J}}(\mathsf{sid})$

$(R, P, h_R, (h_P, m)) \leftarrow \mathcal{F}_{\mathsf{accSMT}}(\mathsf{prove\text{-}order}, \mathsf{sid})$
$R \Longrightarrow : \pi, h$;
assert $1 = \mathcal{F}_{\mathsf{wNIZK}}(\mathsf{Verify}, \mathsf{sid}, (h, h_R, m), \pi)$;
$\Longrightarrow S : h_P, m$;
$S \Longrightarrow : (h_1, \ldots, h_m)$;
if $h_P \neq H_3(h_1, \ldots, h_m)$ **or** $\forall i : h_i \neq h$ **then**
  return $\mathsf{dis}(S)$;
else return $\emptyset$;

$\mathsf{CheatDetect}_{\mathrm{S}}(\mathsf{sid})$

$J \Longrightarrow : h_P, m$;
assert $\exists \mathsf{sid}_S : \mathsf{DB}^{\mathsf{acc}}_{\mathrm{S}}[\mathsf{sid}_S] = (h_P, m)$;
$(r_1, \ldots, r_m) \leftarrow$ **values** $(\mathsf{DB}_{\mathrm{S}})$;
$\Longrightarrow J : H_2(m, r_1), \ldots, H_2(m, r_m)$;
return ok;

Fig. 9. Cheat detection subroutine.

Corruptions Whenever any party (except the server and the judge) gets dynamically corrupted, the adversary not only gets full control over the party, but also learns the messages that this party has previously stored. In particular, we assume that the adversary may learn the contents of DB_{U} and DB_{R} (issued tokens), DB_{V} (state of the revocation database), $\mathsf{DB}'_{\mathrm{U}}$ (tokens prepared for verification), and also $\mathsf{DB}^{\mathsf{acc}}_{\mathrm{U}}$, $\mathsf{DB}^{\mathsf{acc}}_{\mathrm{V}}$, $\mathsf{DB}^{\mathsf{acc}}_{\mathrm{R}}$ (accountability records). We assume that the adversary does *not* learn temporary values like the hash h presented to the verifier when it was honest, which would lead to additional leakages.

Protocol properties.

Leakages. The leakages of Π_{rev} are given in Table 2. A leakage can be defined as $\mathcal{L}_f^{\mathcal{C}}(x) = \bigcup_{\mathcal{C}' \subseteq \mathcal{C}} \mathcal{L}_f^{\mathcal{C}'}(x)$, where $\mathcal{C}'$ represents a particular combination of corrupted parties. For convenience, we only provide the minimal set of leakages $\mathcal{L}_f^{\mathcal{C}'}(x)$, sufficient to reconstruct $\mathcal{L}_f^{\mathcal{C}}(x)$. Here $\mathcal{C}' = \emptyset$ is the leakage without corruptions.

Routine	Leakage	Value	Comments
Issue	$\mathcal{L}^{\emptyset}_{issue}(U,R,x)$	U,R	Do not hide the party identities.
	$\mathcal{L}^{\{R\}}_{issue}(U,R,x)$	x	The revoker learns x
	$\mathcal{L}^{\{U\}}_{issue}(U,R,x)$	x	The user learns x
Revoke	$\mathcal{L}^{\emptyset}_{revoke}(R,x)$	R	Do not hide the identity of the revoker R.
		$(R \in \mathcal{R}_x)$ $\wedge (x \notin X^-)$	Leak whether the request is legitimate and x has not been revoked yet.
	$\mathcal{L}^{\{S\}}_{revoke}(R,x)$	$\mathsf{rev\_sids}(x)$	The server detects repeated revocations of x.
	$\mathcal{L}^{\{S,V\}}_{revoke}(R,x)$	$\{x\} \cap \mathsf{pres}(V)$	The server learns x shown to corrupted V.
	$\mathcal{L}^{\mathcal{U}_x}_{revoke}(R,x)$	x	The user of x learns x
Unotify	$\mathcal{L}^{\emptyset}_{unotify}(U,x)$	U	Do not hide the identity of the user U.
		$U \in \mathcal{U}_x$	Leak the fact of illegitimate request.
Vnotify	$\mathcal{L}^{\emptyset}_{vnotify}(V)$	V	Do not hide the identity of the verifier V.
Verify	$\mathcal{L}^{\emptyset}_{verify}(U,V,x)$	$x \notin \mathsf{notif}(U)$	Leak whether the user has previously called a sufficiently fresh notification for x.
	$\mathcal{L}^{\{V\}}_{verify}(U,V,x)$	x	The verifier learns x.
	$\mathcal{L}^{\{S,V\}}_{verify}(U,V,x)$	$\mathsf{rev\_sids}(x)$	Server and verifier learn when x was revoked.

Table 2. Leakages of Π_{rev}. The leakages that can be avoided assuming valid input are coloured gray. The set $\mathsf{pres}(V)$ is the set of tokens x such that $(\mathsf{verify}, \mathsf{sid}, x, V)$ has been input to $\mathcal{F}_{\mathsf{rev}}$. The set $\mathsf{notif}(U)$ is the set of tokens x such that $\mathcal{F}_{\mathsf{rev}}$ has returned $(\mathsf{unotify\text{-}resp\text{-}u}, \mathsf{sid}, \mathsf{true})$ on the *latest* input $(\mathsf{unotify}, \mathsf{sid}, x)$ from the user U, which happened *after* $\mathcal{F}_{\mathsf{rev}}$ sent its last $(\mathsf{vnotify\text{-}resp\text{-}v}, \mathsf{sid}, \mathsf{ok})$ to V. The set $\mathsf{rev\_sids}(x)$ is the set of session identifiers sid such that $(\mathsf{revoke}, \mathsf{sid}, x)$ has been input to $\mathcal{F}_{\mathsf{rev}}$ before.

Notes on (un)linkability. From Table 2, we see that the value x leaked to the verifier in the presentation breaks unlinkability. To get rid of this problem, we could follow the ideas of [27], having a separate revocation list for every verifier. Technically, instead of $H_2(m,r)$, the verifier V would receive $H_2(m,r,V)$, so that verifiers V and V' would not be able to correlate presentations made to them. Also, the value of m included into the hashes could be updated more frequently than with every revocation. However, in this case x could not be directly sent to V, and the check $v(x)$ would need to be a part of the NIZK proof. Its particular efficiency would depend on the embedding credential scheme.

We see our revocation mechanism as an extension to a standard mDL [25] scheme, for which anonymisation techniques relying on computing ECDSA and parsing mDL document format in ZK have been proposed by Frigo and shelat [19]. These components have as well be implemented for the ZK-SecreC library[5] [31]. This allows to do proofs in ZK, without actually showing the signature and the mDL document to the verifier.

While we have not implemented the full mDL-embedded verification, we could estimate its efficiency as follows. The checks $H_1(s,r) = x$ and $H_2(m,r,V) =$

[5] https://zk-secrec.cyber.ee/documentation/4-api/EC/, last visited 7th October, 2025

h for the token x included into the private mDL document would contribute to the total proof circuit size. In the benchmarks of [19], the proof generation took less than 1.5s on the benchmarked devices, and the most expensive portion of the proof corresponds to the SHA256 hash. If we add two more instances of SHA256 to the circut, this would keep user computation time within 5 seconds.

UC-realisation. We want to model the fact that the server does not learn to whom the user has presented its credentials. Hence, we assume that the adversary does not have access to secure communication channels between a an *honest* user and an *honest* verifier. We can then prove the following theorem.

Theorem 1. *The protocol set Π_{rev} of Sec. 4.3 UC-realises $\mathcal{F}_{\mathsf{rev}}$ (depicted in Figures 2, 3) w.r.t. leakages of Table 2 in $\mathcal{F}_{\mathsf{cert}}$ [12], $\mathcal{F}_{\mathsf{wNIZK}}$ [28], $\mathcal{F}_{\mathsf{accSMT}}$ (Fig. 1)-hybrid RO model, assuming a statically covertly corrupted server, a statically actively corrupted judge, and dynamically actively corrupted other parties.*

The full proof of Theorem 1 is given in Sec. E. In the main paper, we informally discuss why it is so.

Privacy (informal). The verifier is not able to check whether a token x has been revoked during m revocations, unless the user has proven it explicitly for the given m. Intuitively, the verifier can detect presence/absence of x in $\mathsf{DB}_{\mathrm{REV}}$ only if it manages to compute $H_2(m, r)$ for a corresponding r. The verifier does not have r and cannot compute it from $x = H_1(s, r)$. It can only obtain $H_2(m, r)$ if the user has sent is a a part of the proof explicitly for the given x and m.

If the server S is corrupted, the adversary *does* have the value r of a revoked token, but cannot relate r to $x = H_1(s, r)$ without knowing s. Only if the verifier and the server are *both* corrupted, together they *can* check presented $h = H_2(m, r)$ against the value r shown to the server during revocation.

Correctness (informal). Upon successful issuing, the value $H_1(s, r)$ (and hence r) is confirmed by the revoker. The value $H_2(m, r)$ is uniquely determined by m and r. Hence, if the server and the revoker are both honest, whenever the token containing a randomness r will be revoked by a legitimate revoker, the quantity $H_2(m, r)$ will be among h_i that are confirmed by the server. Now let us see what happens if the revoker or the server is corrupted.

- The server cannot *unrevoke* a previously revoked token without being accountable for that, as the fact of revocation is confirmed by a message sent through $\mathcal{F}_{\mathsf{accSMT}}$, and verification also depends on information sent through $\mathcal{F}_{\mathsf{accSMT}}$.
- The server cannot *revoke* a token of an honest user without consent of an honest revoker of that token. The hash value $H_2(m, r)$ can be computed only knowing the secret r, which is only known to the revoker and the user.
- The server can *refuse* to revoke a token (as a denial of service).
- The revoker can *revoke* only a token previously issued to it, since revocation requires knowing the randomness r.

– The revoker can *unrevoke* a previously revoked token in collaboration with a corrupted server. In this case, there is no party w.r.t. whom the server would be accountable.

5 Comparison with Related Work

Our work has certain similarity with [27], which assumes a semi-trusted server. It would be possible to extend the result of [27] with accountability using commitments and NIZK proofs of correspondence to these commitments. Proving knowledge of exponent r could be more efficient than proving knowledge of a hash input r, potentially giving a faster result.

In a nutshell, while in our protocol the server revocation list consists of hashes $H(m, r_1), \ldots, H(m, r_m)$, in [27] it consists of group elements $g_m^{r_1}, \ldots, g_m^{r_m}$ for a fresh $g_m \leftarrow\!\!\$\ \mathbb{G}$. In the unlinkability game of [27], the adversary needs to guess whether the presented credential g_m^r is linked to the revocation randomness $r = r_0$ or $r = r_1$. This problem is reduced to DDH (Decisional Diffie-Hellman) problem. An adaptation of this construction to our protocol and the corresponding efficiency benchmarks can be found in Sec. C.

While benchmarks show that the verification would indeed be much faster with DDH proofs, as a trade-off, the revocation becomes significantly slower, as the server will need to perform m group exponentiations to update the revocation list of m elements. Another question is whether the new construction would be *universally composable*. In the game-based definition, the adversary tries to make the guess for *one* presentation and stops. In UC, the simulator needs to continue with the subsequent presentations and revocations. Particular simulation failures happen when r leaks to the adversary either due to adaptive user corruption, or during revocation in a corrupted server case. We still need to rely on DDH in these cases, as we need to ensure privacy *before* these events happen. In these cases, $H(m, r)$ can be *programmed* later, but g_m^r becomes distinguishable from any other value.

While we could not prove that the DDH-based construction would UC-realize $\mathcal{F}_{\mathsf{rev}}$ under DDH, coming up with an alternative ideal functionality or an assumption would be an interesting research question. Especially taking into account that [27] describes a natural way to integrate such a solution into anonymous credentials of [10].

6 Efficiency

6.1 Benchmarks

We have implemented the revocation protocol of Sec. 4.3 in Rust programming language. We have used ECDSA signatures of `ring` library as an implementation of $\mathcal{F}_{\mathsf{cert}}$. We assume SHA256 hash function with 64B input and 32B output. For NIZK proofs, we have used `bellman` library, which in turn uses `groth16` proofs based on [23], which use bilinear pairings. Such proofs have been proven in [7]

Computation times (ms)

m	0	1	10	100	1000	10000	100000	1000000
issue	0.00447	–	–	–	–	–	–	–
revoke	0.108	3.31	3.47	16.5	16.9	23.8	90.8	743
unotify	0.0813	0.0787	0.0833	0.137	0.597	5.57	60.8	673
vnotify	0.0752	0.0772	0.0822	0.137	0.647	6.52	66.2	716
verify	0.0102	2280	2280	2290	2280	2280	2280	2290

Communication (bytes)

m	0	10	100	1000	10000	100000	1000000
issue	48	–	–	–	–	–	–
revoke	160	–	–	–	–	–	–
unotify	114	434	3,314	32,114	320,114	3,200,114	32,000,114
vnotify	114	434	3,314	32,114	320,114	3,200,114	32,000,114
verify	162	–	–	–	–	–	–

Table 3. Revocation protocol benchmarks. Notation "–" represents numbers that are the same as for $m = 0$. The rows represent different subroutines, and columns represent the number m of already revoked tokens.

to UC-realise $\mathcal{F}_{\mathsf{wNIZK}}$ in generic group model. A single NIZK proof has constant size 96B (three group elements). In our benchmarks, a single proof could be generated in 2500ms, and verified in 3ms . We have run the implementation with and Intel(R) Core(TM) i5-10210U CPU @ 1.60GHz, 4 cores, 2 threads per core (all 8 threads have been utilised), and 15GB RAM (peak usage 210MB). Resulting benchmarks are given in Table 3. We have *not* included the overhead of preprocessing phase of Groth16 proofs, which needs to be run only once.

A potential bottleneck is the size of the revocation database in the notification protocols. The user and the verifier will need to have a good network connection to verify revocation status. Alternatively, if we agree not to hide user identity from the verifier (it may be a desirable property in some applications), and agree to allow the issuer to leak whose credenital they revoke (it can be fine if such a revocation by force assumes no collaboration on the user side), we could assume a separate instance of $\mathcal{F}_{\mathsf{rev}}$ for every user. However, the verifier would still need to either download all user data in advance, or would need to do it after encountering the user, which would leak user identity to the server.

7 Conclusion

In this work, we have proposed a single-server solution for credential revocation. We assume that a credential contains a special *revocation token* as an attribute, and propose a protocol of handling revocation of such tokens. We have defined an ideal functionality of revocation, and proposed a protocol that UC-realises it. Assuming a single server inevitably increases adversary's ability to tamper with revocation, and we only may have the server *accountable* for cheating, but we cannot *prevent* cheating. Nevertheless, we find such a solution practical,

assuming a careful server who cares for its reputation and will not cheat if it will be accountable for that.

References

1. Regulation (EU) 2024/1183 of the European Parliament and of the Council of 11 April 2024 amending Regulation (EU) No 910/2014 as regards establishing the European Digital Identity Framework, 2024.
2. Yonatan Aumann and Yehuda Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. *Journal of Cryptology*, 23(2):281–343, 2010.
3. Christian Badertscher, Ran Canetti, Julia Hesse, Björn Tackmann, and Vassilis Zikas. Universal composition with global subroutines: Capturing global setup within plain UC. In Rafael Pass and Krzysztof Pietrzak, editors, *Theory of Cryptography - 18th International Conference, TCC 2020, Durham, NC, USA, November 16-19, 2020, Proceedings, Part III*, volume 12552 of *Lecture Notes in Computer Science*, pages 1–30. Springer, 2020.
4. Foteini Baldimtsi, Jan Camenisch, Maria Dubovitskaya, Anna Lysyanskaya, Leonid Reyzin, Kai Samelin, and Sophia Yakoubov. Accumulators with applications to anonymity-preserving revocation. Cryptology ePrint Archive, 2017/043, 2017.
5. Carsten Baum, Olivier Blazy, Jan Camenisch, Jaap-Henk Hoepman, Eysa Lee, Anja Lehmann, Anna Lysyanskaya, René Mayrhofer, Hart Montgomery, Ngoc Khanh Nguyen, et al. Cryptographers' feedback on the eu digital identity's arf. *Tech. Rep.*, 2024.
6. Jan Bobolz, Pooya Farshim, Markulf Kohlweiss, and Akira Takahashi. The brave new world of global generic groups and uc-secure zero-overhead snarks. In Elette Boyle and Mohammad Mahmoody, editors, *Theory of Cryptography - 22nd International Conference, TCC 2024, Milan, Italy, December 2-6, 2024, Proceedings, Part I*, volume 15364 of *Lecture Notes in Computer Science*, pages 90–124. Springer, 2024.
7. Jan Bobolz, Pooya Farshim, Markulf Kohlweiss, and Akira Takahashi. The brave new world of global generic groups and uc-secure zero-overhead snarks. In *Theory of Cryptography Conference*, pages 90–124. Springer, 2024.
8. Dan Boneh and Hovav Shacham. Group signatures with verifier-local revocation. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 168–177, 2004.
9. Jan Camenisch, Manu Drijvers, Tommaso Gagliardoni, Anja Lehmann, and Gregory Neven. The wonderful world of global random oracles. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part I*, volume 10820 of *Lecture Notes in Computer Science*, pages 280–312. Springer, 2018.
10. Jan Camenisch and Anna Lysyanskaya. An efficient system for non-transferable anonymous credentials with optional anonymity revocation. In Birgit Pfitzmann, editor, *Advances in Cryptology - EUROCRYPT 2001, International Conference on the Theory and Application of Cryptographic Techniques, Proceeding*, volume 2045 of *Lecture Notes in Computer Science*, pages 93–118. Springer, 2001.
11. Ran Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *42nd Annual Symposium on Foundations of Computer Science, FOCS 2001, 14-17 October 2001, Las Vegas, Nevada, USA*, pages 136–145. IEEE Computer Society, 2001.

12. Ran Canetti. Universally composable signature, certification, and authentication. In *17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004), 28-30 June 2004, Pacific Grove, CA, USA*, page 219. IEEE Computer Society, 2004.
13. Ran Canetti. Universally composable security. *Journal of the ACM (JACM)*, 67(5):1–94, 2020.
14. Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. Universally composable security with global setup. In *Theory of Cryptography Conference*, pages 61–85. Springer, 2007.
15. Ran Canetti, Kyle Hogan, Aanchal Malhotra, and Mayank Varia. A universally composable treatment of network time. In *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*, pages 360–375. IEEE, 2017.
16. Ran Canetti, Daniel Shahaf, and Margarita Vald. Universally composable authentication and key-exchange with global pki. In *Public-Key Cryptography–PKC 2016*, pages 265–296. Springer, 2016.
17. David Chaum and Torben Pryds Pedersen. Wallet databases with observers. In *Annual international cryptology conference*, pages 89–105. Springer, 1992.
18. Petr Dzurenda, Jan Hajny, Lukas Malina, and Sara Ricci. Anonymous credentials with practical revocation using elliptic curves. In *SECRYPT*, pages 534–539, 2017.
19. Matteo Frigo et al. Anonymous credentials from ecdsa. *Cryptology ePrint Archive*, 2024.
20. Mike Graf, Ralf Küsters, and Daniel Rausch. Accountability in a permissioned blockchain: Formal analysis of hyperledger fabric. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 236–255. IEEE, 2020.
21. Mike Graf, Ralf Küsters, and Daniel Rausch. Auc: Accountable universal composability. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1148–1167. IEEE, 2023.
22. Mike Graf, Ralf Küsters, Daniel Rausch, Simon Egger, Marvin Bechtold, and Marcel Flinspach. Accountable bulletin boards: Definition and provably secure implementation. In *2024 IEEE 37th Computer Security Foundations Symposium (CSF)*, pages 201–216. IEEE, 2024.
23. Jens Groth. On the size of pairing-based non-interactive arguments. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 305–326. Springer, 2016.
24. Jan Hajny and Lukas Malina. Anonymous credentials with practical revocation. In *2012 IEEE First AESS European Conference on Satellite Telecommunications (ESTEL)*, pages 1–6. IEEE, 2012.
25. ISO Central Secretary. Personal identification — ISO-compliant driving licence — Part 5: Mobile driving licence (mDL) application. Standard ISO/IEC 18013-5:2021, International Organization for Standardization, Geneva, CH, 2021.
26. Jorn Lapon, Markulf Kohlweiss, Bart De Decker, and Vincent Naessens. Analysis of revocation strategies for anonymous idemix credentials. In *Communications and Multimedia Security: 12th IFIP TC 6/TC 11 International Conference, CMS 2011, Proceedings 12*, pages 3–17. Springer, 2011.
27. Wouter Lueks, Gergely Alpár, Jaap-Henk Hoepman, and Pim Vullers. Fast revocation of attribute-based credentials for both users and verifiers. *computers & security*, 67:308–323, 2017.
28. Anna Lysyanskaya and Leah Namisa Rosenbloom. Universally composable sigma-protocols in the global random-oracle model. In Eike Kiltz and Vinod Vaikuntanathan, editors, *Theory of Cryptography - 20th International Conference, TCC 2022, Proceedings, Part I*, volume 13747 of *Lecture Notes in Computer Science*, pages 203–233. Springer, 2022.

29. Fabián Alexis Aburto Moccia, Eduardo Recabarren Domínguez, Gustavo Gatica, Berenice Guerra-Tamara, Germán Herrera-Vidal, Jorge Mario Karam Rozo, Alfonso R Romero-Conrado, and Jairo Coronado-Hernández. Complexity and the transition to post-quantum security: Cryptographic challenges regarding shor's and grover's algorithms. *Procedia Computer Science*, 265:674–680, 2025.
30. Rafael Torres Moreno, Jesús García-Rodríguez, Jorge Bernal Bernabé, and Antonio Skarmeta. A trusted approach for decentralised and privacy-preserving identity management. *IEEE Access*, 9:105788–105804, 2021.
31. Härmel Nestra. Zero-knowledge proofs for mdl authentication, 2022.
32. Ilker Ozcelik, Sai Medury, Justin Broaddus, and Anthony Skjellum. An overview of cryptographic accumulators. *arXiv preprint arXiv:2103.04330*, 2021.
33. Christian Paquin and Greg Zaverucha. U-prove cryptographic specification v1.1 (revision 3), December 2013. Released under the Open Specification Promise (http://www.microsoft.com/openspecifications/en/us/programs/osp/default.aspx).
34. Rosa Pericàs-Gornals, Macià Mut-Puigserver, M Magdalena Payeras-Capellá, Miquel Á Cabot-Nadal, and Jaume Ramis-Bibiloni. Digital credentials management system using rejectable soulbound tokens. *Annals of Telecommunications*, pages 1–13, 2024.
35. Yongjun Ren, Xinyu Liu, Qiang Wu, Ling Wang, and Weijian Zhang. Cryptographic accumulator and its application: A survey. *Security and Communication Networks*, 2022(1):5429195, 2022.
36. Rodrigo Q Saramago, Hein Meling, and Leander N Jehl. A privacy-preserving and transparent certification system for digital credentials. In *26th International Conference on Principles of Distributed Systems (OPODIS 2022)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2023.

A More Background on Universal Composability

A.1 Generalised UC model

In plain UC model, a protocol may only send data to $\mathcal{Z}$ in the form of inputs and outputs of its main parties, and exchange data with $\mathcal{A}$, but there is no other uncovered communication with any other machines. Generalised UC (GUC) and Externalised UC (EUC) [14] allow different protocols to use the same instance of a shared functionality. This allows to model things like PKI [16], where using the same keys in different protocols may lead to additional attacks, omitted in the plain UC model where each protocol uses its own independent keys. It has been shown in [3] that global setup can be captured by plain UC model, making the proofs simpler and compatible with proofs made in plain UC model.

In this work, we will be using clock functionality of [15], which is more correct to model as an external functionality which can be shared by different protocols. We will need it to model accountable message transmission functionality. Additionally, we model hash function as a global random oracle as in [9].

While we could use global PKI for signatures, we will not do it, as we propose having special server keys for the purpose of revocation. Moreover, we emphasise that our protocol is *not* secure if the keys are used in multiple applications, as we have no guarantees that the server will not have to sign claims of exactly the same form in some other protocol.

A.2 Building Block Ideal Functionalities

Secure message transmission In all protocols, we assume secure authenticated communication between parties. The underlying UC functionality $\mathcal{F}_{\mathsf{SMT}}$ behind transmission of network messages is taken from [13], and it is depicted in Figure 10. This functionality allows the adversary to send and receive messages on behalf of corrupted parties, and cause arbitrary network delay i communication of honest parties. In this work, we assume that the leakage l of communication of honest parties is the length of the transmitted message.

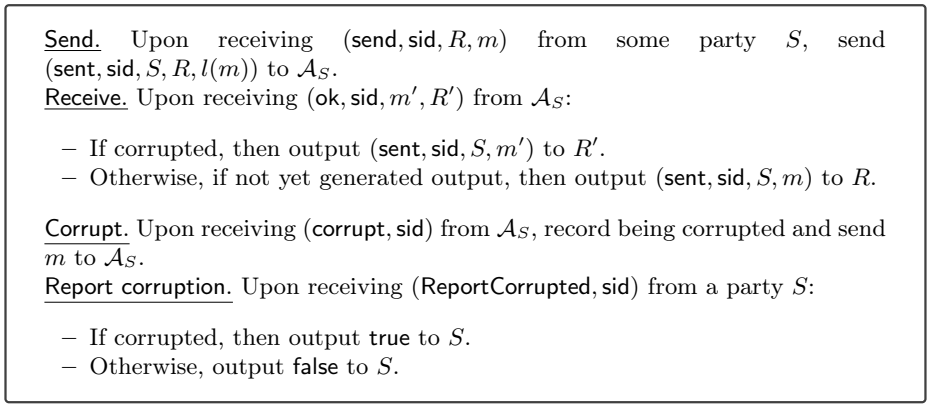

Fig. 10. Secure message transmission functionality $\mathcal{F}_{\mathsf{SMT}}$ from [13] parametrised with leakage function l

Signing For signing, we use certification functionality from Canetti [12], depicted in Figure 11. It provides binding between produced signature and the identity of the signer party. Generation of public keys in is not a part of the functionality. Instead, creating signatures using $\mathcal{F}_{\mathsf{cert}}$ corresponds to sending signatures with *certificates* that bind the verification process to the identity of signer, not the public key.

Clock and Timer The UC functionalities for clock are taken from [15].

There is a perfect reference time functionality $\mathcal{G}_{\mathsf{refClock}}$ (Figure 12), which maintains the global reference time and honestly reports it to all parties. This functionality is not meant to be used by parties directly, but will be needed for the inaccurate clock and the timer functionalities.

There is a more realistic functionality $\mathcal{G}_{\mathsf{clock}}^{P,S,\Sigma}$ that reports *delayed approximate* time to the party P. There are no bounds on the delay time, but the difference between the real and the reported time measurements at the time on

Signature Generation. Upon receiving $(\mathsf{sign}, \mathsf{sid}, m)$ from S, send $(\mathsf{sign}, \mathsf{sid}, m)$ to the $\mathcal{A}$. Upon receiving $(\mathsf{signature}, \mathsf{sid}, m, \sigma)$ from the $\mathcal{A}$, verify that no entry $(m, \sigma, 0)$ is recorded. If it is, then output an error message to S and halt. Else, output $(\mathsf{signature}, \mathsf{sid}, m, \sigma)$ to S, and record the entry $(m, \sigma, 1)$.
Signature Verification. Upon receiving a value $(\mathsf{verify}, \mathsf{sid}, m, \sigma)$ from some party P, send $(\mathsf{verify}, \mathsf{sid}, m, \sigma)$ to the $\mathcal{A}$. Upon receiving $(\mathsf{verified}, \mathsf{sid}, m, \phi)$ from the $\mathcal{A}$, do:

- If $(m, \sigma, 1)$ is recorded then set $f = 1$.
- Else, if the signer is not corrupted, and no entry $(m, \sigma', 1)$ for any σ' is recorded, then set f = 0 and record the entry $(m, \sigma, 0)$.
- Else, if there is an entry (m, σ, f') recorded, then set $f = f'$.
- Else, set $f = \phi$, and record the entry (m, σ', ϕ).

Output $(\mathsf{verified}, \mathsf{sid}, m, f)$ to P.

Fig. 11. The certification functionality $\mathcal{F}_{\mathsf{cert}}$

$\mathcal{G}_{\mathsf{refClock}}$ maintains an integer G corresponding to the reference time. When created, it initialises $G = 0$.
IncrementTime. Upon receiving an IncrementTime request from $\mathcal{Z}$, update $G \leftarrow G + 1$, and send an ok message to $\mathcal{Z}$. Ignore IncrementTime requests sent by any other entity.
GetTime. Upon receiving a GetTime command, return G to the calling entity.

Fig. 12. Global reference time functionality $\mathcal{G}_{\mathsf{refClock}}$ from [15]. It is expected that honest parties do not talk to this functionality directly.

which P receives the response should not exceed the parameter Σ, as far as the clock manager S is honest.

In this work, we assume a trusted clock manager, which allows to omit the party S from the clock functionality. We still allow the adversary to produce delayed inaccurate responses, but the inaccuracy will always be bounded by Σ. This assumption makes the clock functionality simpler, resulting in a functionality $\mathcal{G}_{\mathsf{clock}}^{P,\Sigma}$ (Figure 13).

The timer functionality $\mathcal{F}_{\mathsf{timer}}^{C,\Delta_C,\Sigma_C}$ (Figure 14) allows parties to measure time elapsed between start time and elapsed time queries.

In this work, it is essential that the party C will be able to get a response to its TimeElapsed query instantly. However, in the descriptoin of timer functionality $\mathcal{F}_{\mathsf{timer}}^{C,\Delta_C,\Sigma_C}$ of [15], $\mathcal{Z}$ is allowed to provide a description of a Turing machine M such that the computation of the shift $\sigma_C = M(\mathsf{sid}, G, G')$ takes place *after* the time G has been measured. Even though $\mathcal{Z}$ provides M in advance, and $\mathcal{A}$ is not allowed to produce delays directly, $\mathcal{Z}$ may construct a "slow" M whose computation time introduces an arbitrary delay between the points of time when

The functionality $\mathcal{G}_{\mathsf{clock}}^{P,\Sigma}$ is given the following parameters:

- The clock owner P who may request time measurements.
- The maximum allowable shift Σ from the reference time.

GetTime. Upon receiving input $(\mathsf{GetTime}, \mathsf{sid})$ from party P:

1. Send $(\mathsf{Sleep}, \mathsf{sid})$ to $\mathcal{A}$.
2. Wait for a response of the form $(\mathsf{Wake}, \mathsf{sid}, \sigma)$ from $\mathcal{A}$.
3. If $\sigma = \perp$ or $|\sigma| > \Sigma$, output $(\mathsf{TimeReceived}, \mathsf{sid}, \perp)$ to P and stop.
4. Send $\mathsf{GetTime}$ to $\mathcal{G}_{\mathsf{refClock}}$ to receive G. Compute $T_P \leftarrow G + \sigma$.
5. Output $(\mathsf{TimeReceived}, \mathsf{sid}, T_P)$ to P.

Fig. 13. Global time functionality $\mathcal{G}_{\mathsf{clock}}^{P,\Sigma}$, which is a special case of $\mathcal{G}_{\mathsf{clock}}^{P,S,\Sigma}$ from [15], assuming a trusted clock manager S. There is no limit on the delay of the response. However, the value returned to P, is at most Σ from the true response time.

the end time G was measured, and when the response was output to C. For that reason, we need to be more specific about M. In this work, we assume that there is just some fixed offset σ_C that $\mathcal{Z}$ may provide, which intuitively corresponds to local clock being faster/slower than it should be. This leads to a simplfied functionality depicted in Figure 14, which we will be using in this work.

Restricted Programmable Observable Random Oracle We use ideal functionality $\mathcal{G}_{\mathsf{rpRO}}$ (Figure 15) for Restricted Programmable Observable Global Random Oracle (rpGRO) [9] that enables the GUC-realisation of hash commitments. In the proof, we need to be able to program RO on specific input and observe queries made to the RO (by other parties). With this functionality, adversary is allowed to obtain the list of queries made to the functionality by machines that are not part of session sid (illegitimate queries). Since honest parties make only legitimate queries, then adversary only learns information about the queries made by adversary itself. However, the simulator can see all the queries made through the corrupt machines within the session sid as it is the ideal-world attacker, it will see all the legitimate queries with sid. Additionally, functionality allows adversary to program RO, but parties from session sid can query functionality and check if the RO has been programmed on specific input. In the ideal world, simulator is the only party that can check whether RO has been programmed on specific input. The simulator has extra power (over the real-world adversary) as it is allowed to program RO without being detected. Since our protocol requires multiple random oracles, we will use domain separation by providing additional input to each query of the functionality that identifies which instance of RO is being queried.

The functionality $\mathcal{F}_{\mathsf{timer}}^{C,\Delta_C,\Sigma_C}$ is given the following parameters:

- The timer owner C who may request timer.
- The maximum allowable delay Δ_C between start time and elapsed time.
- The maximum allowable shift Σ_C on the reported time difference.

Upon initialisation, set $\sigma_C = 0$.
SetShift. Upon receiving a command $(\mathsf{SetShift}, \mathsf{sid}, \sigma_C)$ from $\mathcal{Z}$, update local copy of σ_C and send an $(\mathsf{ok}, \mathsf{sid})$ message to $\mathcal{Z}$.
Start. Upon receiving input $(\mathsf{Start}, \mathsf{sid})$ from the party C, if the command $(\mathsf{Start}, \mathsf{sid})$ was previously received, then ignore this request. Otherwise:

1. Send $\mathsf{GetTime}$ to $\mathcal{G}_{\mathsf{refClock}}$. Denote its response as G'.
2. Record (G', sid).
3. Send an $(\mathsf{ok}, \mathsf{sid})$ message to C.

TimeElapsed. Upon receiving input $(\mathsf{TimeElapsed}, \mathsf{sid})$ from the party C, if no previous $(\mathsf{Start}, \mathsf{sid})$ command was received, then ignore this request. Otherwise:

1. Send $\mathsf{GetTime}$ to $\mathcal{G}_{\mathsf{refClock}}$. Denote its response as G.
2. Compute $\delta = G - G' + \sigma_C$ for the previously recorded (G', sid).
3. If $\delta \leq \Delta_C$ and $\sigma_C \leq \Sigma_C$, output (δ, sid) to C.
   Otherwise, output $(\perp, \mathsf{sid})$ to C.

Fig. 14. Timer functionality $\mathcal{F}_{\mathsf{timer}}^{C,\Delta_C,\Sigma_C}$, which is a special case of $\mathcal{F}_{\mathsf{timer}}^{C,\Delta_C,\Sigma_C}$ from [15], assuming a constant Turing machine $M(\mathsf{sid}, G, G') = \sigma_C$. It returns to its owner C the approximate relative time elapsed between the Start and $\mathsf{TimeElapsed}$ commands (using constant accuracy shift σ_C).

Let T be the hash table that is used to store queries to the oracle H. Let $\mathsf{T}_{\mathsf{prog}}$ be a list of programmed values in the oracle H. Both are initially empty. The size of the oracle output is ℓ.
Hash: Upon receiving input $(\mathsf{hash}, i, \mathsf{sid}, m)$, from a party P:

- If there is a record $(i, m, h') \in \mathsf{T}$ for some h', set $h \leftarrow h'$.
- Otherwise, sample random $h' \leftarrow \{0,1\}^{\ell}$ and create a record $(i, m, h') \in \mathsf{T}$, set $h \leftarrow h'$.
- If this query is made by the adversary $\mathcal{A}$ or if sid does not correspond to a session, then add (sid, i, m, h) to the list of illegitimate queries $\mathcal{Q}_{\mathsf{sid}}$ of a session sid.
- Output $(\mathsf{hash\text{-}response}, \mathsf{sid}, i, m, h)$ to party P.

Observe: Upon receiving input $(\mathsf{observe}, \mathsf{sid})$ from the adversary $\mathcal{A}$:

- If $\mathcal{Q}_{\mathsf{sid}}$ does not exist, then set $\mathcal{Q}_{\mathsf{sid}} = \perp$.
- Output $(\mathsf{observed\text{-}list}, Q_{\mathsf{sid}})$.

Program: Upon receiving $(\mathsf{program\text{-}RO}, \mathsf{sid}, i, m, h)$ s.t. $h \in \{0,1\}^{\ell}$ from the adversary $\mathcal{A}$:

- If $\exists h' \in \{0,1\}^{\ell}$ s.t. $(i, m, h') \in \mathsf{T}$ and $h \neq h'$, ignore this query.
- Create a record $(i, m, h) \in \mathsf{T}$ and add an entry of (i, m) to the table $\mathsf{T}_{\mathsf{prog}}$.
- Output $(\mathsf{program\text{-}confirmed})$.

IsProgrammed: Upon receiving $(\mathsf{is\text{-}programmed}, \mathsf{sid}, i, m)$ from a party P:

- If sid does not correspond to current session, ignore this query.
- If $(i, m) \in \mathsf{T}_{\mathsf{prog}}$, set $b = 1$, else $b = 0$.
- Output $(\mathsf{is\text{-}programmed}, b)$

Fig. 15. Restricted Programmable Observable Global Random Oracle $\mathcal{G}_{\mathsf{rpRO}}$

Non-interactive zero-knowledge proofs We use functionality $\mathcal{F}_{\mathsf{wNIZK}}$ (Figure 16). It is based on weak NIZK functionality of Bobolz et. al. [6] which includes the property of malleability, as we are using malleable groth16 proofs in the actual implementation.

Instead of querying the proof and the witness extraction directly from the adversary, as done in [6], we follow the idea by Lysyanskaya and Rosenbloom in [28], where proofs and witness extraction algorithms are given in advance during setup phase. This is because otherwise we would not achieve privacy of presented token for honest prover and honest verifier, while in reality the proof is generated locally by the user, and is *not* observed by the adversary.

Setup. Upon receiving input message $(\mathsf{Setup}, \mathsf{sid})$ from a party P:

1. If this is the first time that $(\mathsf{Setup}, \mathsf{sid})$ was received, send message $(\mathsf{Setup}, \mathsf{sid})$ to the $\mathcal{A}$.
2. Upon receiving $(\mathsf{Algorithms}, \mathsf{sid}, \mathsf{SimProve}, \mathsf{SimExtract})$ from $\mathcal{A}$, store it.

Proof. Upon receiving input message $(\mathsf{Prove}, \mathsf{sid}, x, w)$ from P:

1. If $\mathcal{R}(x, w) = 1$, compute $\pi \leftarrow \mathsf{SimProve}(x)$.
2. Store (x, π) in T_π and output $(\mathsf{Proof}, \mathsf{sid}, \pi)$ to P.

Verify. Upon receiving input message $(\mathsf{Verify}, \mathsf{sid}, x, \pi)$ from V:

1. If $(x, \pi) \in \mathsf{T}_\pi$, output $(\mathsf{Verification}, \mathsf{sid}, x, \pi, 1)$ to V.
2. Otherwise, compute $w \leftarrow \mathsf{SimExtract}(x, \pi)$.
   (a) If $\mathcal{R}(x, w) = 1$, then store (x, π) in T_π.
   (b) If $(w = \mathsf{maul} \wedge (x, \cdot) \in \mathsf{T}_\pi)$ then store (x, π) in T_π.
   (c) If $(x, \pi) \in \mathsf{T}_\pi$, output $(\mathsf{Verification}, \mathsf{sid}, x, \pi, 1)$ to V. Otherwise, output $(\mathsf{Verification}, \mathsf{sid}, x, \pi, 0)$ to V.

Fig. 16. Ideal functionality $\mathcal{F}_{\mathsf{wNIZK}}$

B Real Protocol for Accountable Message Transmission

We propose a real protocol Π_{accSMT} for UC-realisation of $\mathcal{F}_{\mathsf{accSMT}}$ described in Sec. 3. The protocol will use common signatures, for which we use signing functionality $\mathcal{F}_{\mathsf{cert}}$ of [11]. The messages are transmitted between parties using message transmission functionality $\mathcal{F}_{\mathsf{SMT}}$ of [13]. Both functionalities are repeated in Sec. A.2.

In the real protocol, we assume a functionality $\mathcal{G}'_{\mathsf{clock}}$ that outputs consistently the time to all honest parties. We assume that different parties involved into the revocation functionality (issuers, users, verifiers) do not necessarily have a possibility to synchronise their clocks in advance, so we consider the *delayed approximate time* functionality of [15], which is reproduced in Sec. A.2. This functionality is parametrised as $\mathcal{G}_{\mathsf{clock}}^{P,S,\Sigma}$, where P is the owner of functionality who may read time measurements, S is a party whose honesty ensures correctness of measurements, and Σ is the guaranteed bound on the difference in actual and reported measurements. For simplicity, we assume that the party S that may break the time accuracy guarantee is honest, and formally we take $S = P$, i.e. only corrupted P itself may harm accuracy of its own measurements, denoting $\mathcal{G}_{\mathsf{clock}}^{P,\Sigma} = \mathcal{G}_{\mathsf{clock}}^{P,P,\Sigma}$.

Accountable transmission (Fig. 17). Whenever a party R starts listening for a message from S, it waits from the sender S a message msg together with a signed claim that "the message msg has been sent on time t". R checks whether the signed timestamp t is not too different from the current time t' as measured

by R locally, which means that the reported claim is not "too old" in case of a network delay. Later, R can use the signature σ as an evidence.

It is important to set a limit on the session time of R, as otherwise $\mathcal{A}_S$ may delay the termination, causing the party that received an output of $\mathcal{F}_{\mathsf{accSMT}}$ earlier to receive a later timestamp. We will need to fix the choice of parameters of clock-related functionalities before stating and proving that the protocol UC-realises $\mathcal{F}_{\mathsf{accSMT}}$.

For simplicity, we assume that the server is able to measure the time precisely with error $\Sigma = 0$. The reason is that getting a different accuracy error on each measurement would cause an honest server "indeliberately" cheating, putting an earlier timestamp on a message that it has sent earlier. In practice, we can assume that the server clock would *not* be synchronised with some external system inside each session. Even if the local clock is not precise, we assume that it would be a *constant* shift, which excludes such attacks. Since $\mathcal{G}_{\mathsf{clock}}^{S,\Sigma}$ would need to be modified to assume constant shift, we just chose to take $\Sigma = 0$, as in our application the clock is relative anyway.

The checks $t_S \leq t_R + \Sigma$ and $t_S \geq t_R - (\delta + \sigma)$ takes into account possible difference in t_S and t_R due to inaccurately reported t. Even in the case of a delay δ, it includes the delay into allowed difference. As the result, these checks will not affect an honest server.

$\mathsf{send}_S(\mathsf{sid}, R, \mathsf{msg})$	$\mathsf{receive}_R(\mathsf{sid})$
$t_S \leftarrow \mathcal{G}_{\mathsf{clock}}^{S,0}(\mathsf{GetTime}, \mathsf{sid})$;	1 : $\mathsf{ok} \leftarrow \mathcal{F}_{\mathsf{timer}}^{R,\Delta_R,\Sigma_R}(\mathsf{Start}, \mathsf{sid})$;
$\sigma \leftarrow \mathcal{F}_{\mathsf{cert}}(\mathsf{sign}, \mathsf{sid}, (R, \mathsf{msg}, t_S))$;	2 :
$\Longrightarrow R : \mathsf{msg}, t_S, \sigma$;	3 : $S \Longrightarrow : \mathsf{msg}, t_S, \sigma$
	4 : $\mathsf{ok} \leftarrow \mathcal{F}_{\mathsf{cert}}(\mathsf{verify}, \mathsf{sid}, (R, \mathsf{msg}, t_S), \sigma)$;
	5 : $t_R \leftarrow \mathcal{G}_{\mathsf{clock}}^{R,\Sigma}(\mathsf{GetTime}, \mathsf{sid})$;
	6 : $\delta \leftarrow \mathcal{F}_{\mathsf{timer}}^{R,\Delta_R,\Sigma_R}(\mathsf{TimeElapsed}, \mathsf{sid})$;
	7 : **assert** $(t_S \leq t_R + \Sigma) \wedge (t_S \geq t_R - (\delta + \Sigma))$
	8 : $\mathsf{DB}_\mathrm{R}[\mathsf{sid}] \leftarrow (\mathsf{msg}, t_S, \sigma)$
return ok;	9 : **return** msg;

Fig. 17. Sending and receiving message.

Prove fact of transmission . The receiver R wants to prove to J which message msg has been sent to R in a session sid'. For this, they take previously recorded server signature σ on (R, msg, t). The party J checks that σ is a valid signature.

Prove transmission order . The receivers R_1 and R_2 want to prove to J that S has sent to R_1 a message msg_1 in a session sid_1 strictly *before* it has sent to R_2 a message msg_2 i na session sid_2. For this, they look for previously recorded server

signatures σ_1 on $(R_1, \mathsf{msg}_1, t_1)$, and σ_2 on $(R_2, \mathsf{msg}_2, t_2)$. The party J checks that σ_1 and σ_2 are valid signatures, and that $t_1 < t_2$.

$\mathsf{ProveSent}_R(\mathsf{sid}, \mathsf{sid}')$	$\mathsf{ProveSent}_J(\mathsf{sid})$
1 : $(\mathsf{msg}, t, \sigma) \leftarrow \mathsf{DB}_\mathrm{R}[\mathsf{sid}']$	$R \Longrightarrow: (\mathsf{msg}, t, \sigma)$;
2 : $\Longrightarrow J : (\mathsf{msg}, t, \sigma)$	$\mathsf{ok} \leftarrow \mathcal{F}_{\mathsf{cert}}(\mathsf{verify}, \mathsf{sid}, (R, \mathsf{msg}, t), \sigma)$;
3 : **return** ok;	**return** (R, msg);

Fig. 18. Prove that a message has been sent.

$\mathsf{ProveOrder}_{R_i}(\mathsf{sid}, \mathsf{sid}_i)$// $i \in \{1, 2\}$	$\mathsf{ProveOrder}_J(\mathsf{sid})$
1 : $(\mathsf{msg}_i, t_i, \sigma_i) \leftarrow \mathsf{DB}_{\mathrm{R}_i}[\mathsf{sid}_i]$	$R_1 \Longrightarrow: (\mathsf{msg}_1, t_1, \sigma_1)$;
2 : $\Longrightarrow J : (\mathsf{msg}_i, t_i, \sigma_i)$	$R_2 \Longrightarrow: (\mathsf{msg}_2, t_2, \sigma_2)$;
3 :	$\mathsf{ok} \leftarrow \mathcal{F}_{\mathsf{cert}}(\mathsf{verify}, (\mathsf{sid}, 1), (R_1, \mathsf{msg}_1, t_1), \sigma_1)$;
4 :	$\mathsf{ok} \leftarrow \mathcal{F}_{\mathsf{cert}}(\mathsf{verify}, (\mathsf{sid}, 2), (R_2, \mathsf{msg}_2, t_2), \sigma_2)$;
5 :	**assert** $(t_1 < t_2)$
6 : **return** ok;	**return** $(R_1, R_2, \mathsf{msg}_1, \mathsf{msg}_2)$;

Fig. 19. Prove order of sent messages.

Corruptions Whenever a party gets corrupted, the adversary not only gets control over the party, but also learns the messages that it has previously stored. In particular, we assume that the adversary may learn the contents of the tables DB_R that store received messages and signatures on them.

Parameters and assumptions . We need to make some assumptions on the inaccuracy and delay parameters of time functionalities. The parameters Σ and Σ_P depend on clock inaccuracy, and are given in advance. The smaller they are, the better, but they are not adjustable. The parameter Δ_R is the tolerated delay of the timer, and it can be adjusted.

On one hand, while the adversary is potentially able to cause infinite delays in network communication, we still want that the protocol would work in the case of reliable network with limited delay time, which puts a lower bound on Δ_R. Let Δ_{SIGN} and Δ_{VER} be the upper bounds on computing the signing and the verification of $\mathcal{F}_{\mathsf{cert}}$ (these values come from the particular protocols implementing $\mathcal{F}_{\mathsf{cert}}$). Let us now consider optimistic setting, where Δ_{NET} is the maximum network delay of moving packets between R and S, and Δ_{GT} is the maximum delay of the response to the time query. From the protocol of Figure 17, we can count the operations by honest R and S between the timer

start and elapsed time, and see that honest R and S will accept the response if $\Delta_R \geq \Delta_{SIGN} + 2\Delta_{NET} + \Delta_{VER} + 2 \cdot \Delta_{GT}$.

On the other hand, if Δ_R is too large the adversary may use delays to introduce inconsistencies. If two parties query for size within time offset Δ_R, the server will be able to make the party R that terminated earlier to accept a later timestamp t. As stated in the following theorem, this problem can be solved by making the timing between these parties at least $\Delta_R + 2\Sigma_R + 4\Sigma$.

Theorem 2. *Let $l(\mathsf{msg})$ be defined as $|\mathsf{msg}|$, i.e. the leakage is the message length. Assuming environment $\mathcal{Z}$ that makes* receive *requests with time offset at least $\Delta_{ACC} = \Delta_R + 2\Sigma_R + 4\Sigma$, the protocol set Π_{accSMT} GUC-realises $\mathcal{F}_{\mathsf{accSMT}}$ in $\mathcal{F}_{\mathsf{SMT}}$, $\mathcal{F}_{\mathsf{cert}}$, $\mathcal{F}_{\mathsf{timer}}^{R,\Delta_R,\Sigma_R}$, $\mathcal{G}_{\mathsf{clock}}^{R,\Sigma}$-hybrid model.*

The full proof of Theorem 2 can be found in Appendix D. Here we provide a proof sketch.

Privacy. In Π_{accSMT}, we assume that $\mathcal{F}_{\mathsf{SMT}}$ leaks some information $l(\mathsf{msg})$ about the sent message. The same value is output to the adversary in $\mathcal{F}_{\mathsf{accSMT}}$. Assuming that l leaks the message length, the leakage of signature sigma and the timestamp is a constant value.

Correctness of transmission. Similarly to Π_{accSMT}, $\mathcal{F}_{\mathsf{accSMT}}$ allows the corrupted server to choose any output for the client.

Correctness of proof of transmission.

- Suppose that some party R has provided an evidence that has been accepted by J. If the sender is honest, then it will never produce signature on (R, m, t) for a message m that it has not sent to R. Hence, even if R is corrupted, it cannot convince J in a false claim.
- Suppose a corrupted server has sent m to R, and honest R has accepted this messages. This means that R received a valid signature of S on (R, m, t). Such a signature will be accepted by J.

Correctness of proof of ordering.

- Suppose that some parties R_1 and R_2 have provided an evidence that have been accepted by J. If the sender is honest, then it will only produce signatures on (R_1, m_1, t_1) and (R_2, m_2, t_2) for a message m_1 that it indeed has sent to R_1 *before* sending the message m_2 to R_2, assuming that the clock imprecision is constant. Hence, even if R_1 or R_2 is corrupted, they cannot convince J in a false claim.
- Suppose a corrupted server has sent m_1 earlier than m_2, together with some signatures (R_1, m_1, t_1) and (R_2, m_2, t_2), and honest R_1, R_2 have accepted these messages. We show that R_1 and R_2 can convince J that m_1 has been sent before. It is important that an honest party only accepts answers within a delay Δ_R, and $\mathcal{Z}$ will not send requests more frequently than within time

$\Delta_R+2\Sigma_R+4\Sigma$. If the adversary manages to increase the delay of R_1, making it accept a timestamp $t_1 > t_2$, then, even taking into account the sum of imprecisions of all clock measurements $2\Sigma_R + 4\Sigma$, the local timer elapse of R_1 would exceed Δ_R, and message rejected.

C Protocols based on DDH

C.1 The Updated Protocol

In this section, we show how revocation mechanism would look like adopting the DDH mechanism of Lueks et. al. [27]. We use notation g_m for the public exponent of the m-th revocation epoch, which has been computed using a hash function in group $\mathbb{G}$, and can be parametrized as $g_{m,V}$ for a verifier identity V to ensure unlinkability of presentations to different verifiers. We base the DDH protocols on the hash protocols, explicitly striking through the removed pieces of code, and highlighting the new components. The proofs of $\log_{g_1} h_1 = \log_{g_2} h_2$ can be done in zero knowledge, using procedures of Fig. 20 [17].

$\mathsf{DHP}^H[r\restriction^{u,v}_{g,h}]$:	$\mathsf{ChP}^H[\pi\restriction^{u,v}_{g,h}]$:
1 : $s \leftarrow\!\!\$\ \mathbb{Z}_p$	1 : $\alpha \leftarrow g^{\gamma}/u^{\beta}$
2 : $\alpha \leftarrow g^s, \quad \alpha' \leftarrow h^s$	2 : $\alpha' \leftarrow h^{\gamma}/v^{\beta}$
3 : $\beta \leftarrow H(g,h,u,v,\alpha,\alpha',ctx) \in \mathbb{Z}_p$	3 : **assert** $\beta = H(g,h,u,v,\alpha,\alpha',ctx) \in \mathbb{Z}_p$
4 : $\gamma \leftarrow s + r \cdot \beta$	
5 : return $\pi \leftarrow (\beta,\gamma)$	

Fig. 20. NIZK proof that $\log_g u = \log_h v$ [17].

Issuing (Fig. 21) . The token is defined as $x = g^r$. The component s is omitted. Defining $x = g^{sr}$ analogously to $H_1(s,r)$ would not provide additional privacy: for the proofs, we would need to include g^s as well, but g^{sr} would be computable from g^s and r. The additional leakage is that now a corrupted server will learn which x is being revoked.

Revocation (Fig. 22) . In the revocation protocol, hash commitments $H_1(s,r)$ are replaced with pairs (g^s, g^{sr}), and $H_2(m,r)$ with g^r_m.

User notification (Fig. 23) . Instead of $h = H_2(m,r)$, the user computes $h = g^r_m$.

Verifier notification (Fig. 24) . This subroutine is exactly the same as for the original hash-based protocol.

$\mathsf{Issue}_{\mathrm{U}}(\mathsf{sid}, R)$	$\mathsf{Issue}_{\mathrm{R}}(\mathsf{sid}, U)$
$\cancel{s \leftarrow\$ \mathcal{R}_S}$; $r' \leftarrow\$ \mathcal{R}_R$;	
$\Longrightarrow R : \cancel{s}, r'$	$U \Longrightarrow: \cancel{s}, r'$
$r \leftarrow H_1(U, r')$;	$r \leftarrow H_1(U, r')$;
$x \leftarrow \cancel{H_1(s, r)}\ g^r$;	$x \leftarrow \cancel{H_1(s, r)}\ g^r$;
$\mathsf{DB}_{\mathrm{U}}[x] \leftarrow (\cancel{s}, r)$;	**append** $\mathsf{DB}_{\mathrm{R}}[x] \leftarrow (\cancel{s}, r)$;
return x;	**return** x;

Fig. 21. Token issuing subroutine (DDH version).

$\mathsf{Revoke}_{\mathrm{R}}(\mathsf{sid}, x)$	$\mathsf{Revoke}_{\mathrm{S}}(\mathsf{sid})$
$(\_, r) \leftarrow \mathsf{DB}_{\mathrm{R}}[x]$;	$(r_1, \ldots, r_{m-1}) \leftarrow$ **values** $(\mathsf{DB}_{\mathrm{S}})$;
$\Longrightarrow S : r$	$R \Longrightarrow: r$
	$s_R \leftarrow\$ \mathcal{R}_S$;
	$h_R \leftarrow \cancel{H_1(s_R, r)}\ (g^{s_R}, g^{s_R r})$;
$S \overset{\mathsf{acc}}{\Longrightarrow}: h_R$	$\overset{\mathsf{acc}}{\Longrightarrow} P : h_R$
assert $h_R = \cancel{H_1(s_R, r)}\ (g^{s_R}, g^{s_R r})$;	$r_m \leftarrow r$;
$\mathsf{DB}^{\mathsf{acc}}_{\mathrm{R}}[\mathsf{sid}] \leftarrow (h_R, s_R, r)$;	$\mathsf{DB}_{\mathrm{S}}[m] \leftarrow r_m$;
$\mathsf{DB}_{\mathrm{R}}[x] \leftarrow \perp$;	**for** $i = 1..m$ **do** $h_i \leftarrow \cancel{H_2(m, r_i)}\ g_m^{r_i}$;
	$\mathsf{DB}_{\mathrm{REV}} \leftarrow (h_1, \ldots, h_m)$;
return ok;	**return** ok;

Fig. 22. Token revoking subroutine (DDH version).

$\mathsf{UNotify}_{\mathrm{U}}(\mathsf{sid}, x)$	$\mathsf{UNotify}_{\mathrm{S}}(\mathsf{sid})$
	$(h_1, \ldots, h_m) \leftarrow$ **values** $(\mathsf{DB}_{\mathrm{REV}})$;
$S \Longrightarrow: (h_1, \ldots, h_m)$	$\Longrightarrow U : (h_1, \ldots, h_m)$
$S \overset{\mathsf{acc}}{\Longrightarrow}: (h_U, m)$	$\overset{\mathsf{acc}}{\Longrightarrow} U : (h_U = H_3(h_1, \ldots, h_m), m)$
assert $H_3(h_1, \ldots, h_m) = h_U$	$\mathsf{DB}^{\mathsf{acc}}_{\mathrm{S}}[\mathsf{sid}] \leftarrow (h_U, m)$;
$\mathsf{DB}^{\mathsf{acc}}_{\mathrm{U}}[\mathsf{sid}] \leftarrow (h_U, m)$;	
$(\cancel{s}, r) \leftarrow \mathsf{DB}_{\mathrm{P}}[x]$;	
$h \leftarrow \cancel{H_2(m, r)}\ g_m^r$;	
$b \leftarrow h \in \{h_1, \ldots, h_m\}$;	
$\mathsf{DB}'_{\mathrm{U}}[x] \leftarrow (\cancel{s}, r, b, m)$;	
return b;	**return** ok;

Fig. 23. User notification subroutine (DDH verison).

$\mathsf{VNotify}_{\mathrm{V}}(\mathsf{sid})$	$\mathsf{VNotify}_{\mathrm{S}}(\mathsf{sid})$
1 :	$(h_1,\ldots,h_m) \leftarrow$ **values** $(\mathsf{DB}_{\mathrm{REV}})$;
2 : $S \Longrightarrow: (h_1,\ldots,h_m)$	$\Longrightarrow V : (h_1,\ldots,h_m)$;
3 : $S \overset{\mathsf{acc}}{\Longrightarrow}: (h_V, m)$	$\overset{\mathsf{acc}}{\Longrightarrow} V : (h_V = H_3(h_1,\ldots,h_m), m)$;
4 : **assert** $H_3(h_1,\ldots,h_m) = h_V$	$\mathsf{DB}_{\mathrm{S}}^{\mathsf{acc}}[\mathsf{sid}] \leftarrow (h_V, m)$;
5 : $\mathsf{DB}_{\mathrm{V}}^{\mathsf{acc}}[\mathsf{sid}] \leftarrow (h_V, m)$;	
6 : $\mathsf{DB}_{\mathrm{V}} \leftarrow (h_1,\ldots,h_m)$;	
7 : **return** ok;	**return** ok;

Fig. 24. Verifier notification subroutine (DDH version, same as hash version).

$\mathsf{Verify}_{\mathrm{U}}(\mathsf{sid}, x, V)$	$\mathsf{Verify}_{\mathrm{V}}(\mathsf{sid}, v)$
1 : $(s, r, b, m_U) \leftarrow \mathsf{DB}'_{\mathrm{U}}[x]$;	$(h_1,\ldots,h_{m_V}) \leftarrow$ **values** $(\mathsf{DB}_{\mathrm{V}})$;
2 : **assert** $b = \mathsf{false}$;	$\Longrightarrow U : m_V$;
3 : $V \Longrightarrow: m_V$;	
4 : **assert** $m_V \leq m_U$;	
5 : $h \leftarrow$ ~~$H_2(m_V, r)$~~ $g_{m_V}^r$;	$U \Longrightarrow: h, x, \pi$
6 : ~~$\pi \leftarrow \mathcal{F}_{\mathsf{wNIZK}}(\mathsf{Prove}, \mathsf{sid}, (h, x, m_V), (s, r))$~~	~~**assert** $1 = \mathcal{F}_{\mathsf{wNIZK}}(\mathsf{Verify}, \mathsf{sid}, (h, x, m_V), \pi)$~~
7 : $\pi \leftarrow \mathsf{DHP}^H[r \upharpoonright^{x,h}_{g,g_{m_V}}]$	**assert** $1 = \mathsf{ChP}^H[\pi \upharpoonright^{x,h}_{g,g_{m_V}}]$
8 : $\Longrightarrow V : h, x, \pi$	**assert** $h \notin \{h_1,\ldots,h_{m_V}\} \wedge v(x)$;
9 : **return** ok;	**return** ok;

Fig. 25. Token verification subroutine (DDH version).

Verification (Fig. 25) . Instead of a NIZK proof that $h = H_2(m_V, r)$ and $x = H_1(s, r)$ for the same r, the user generates NIZK proofs that $h = g_m^r$ and $x = g^r$ for the same r.

Cheat detection (Fig. 26) . As cheat detection includes generation of revocation list and verification, it inherits changes of the corresponding subroutines.

C.2 Efficiency of the Updated Protocol

The efficiency benchmarks for the DDH based version can be found in Table 4. We have used p256 library of Rust for elliptic curve operations. Comparing these results against Table 3 of the hash version, we see that:

- The issuing overhead is slightly larger, as hash computation is faster than elliptic curve scalar multiplication, but it is fast anyway.
- The notification runtimes are similar. Indeed, there is no difference in the verifier notification, and the user notification includes an additional group exponentiation.

$\mathsf{CheatDetect_P}(\mathsf{sid}, \mathsf{sid}_P)$ // $\{P, P'\} \in \{V, U\}$

1 : $(h_P, m) \leftarrow \mathsf{DB}_{\mathrm{P}}^{\mathsf{acc}}[\mathsf{sid}_P]$;
2 : $\mathsf{ok} \leftarrow \mathcal{F}_{\mathsf{accSMT}}(\mathsf{prove\text{-}order\text{-}2}, \mathsf{sid}, \mathsf{sid}_P)$
3 : $\Longrightarrow R : m$;
4 :
5 :
6 :
7 :
8 : **return** ok;

$\mathsf{CheatDetect_R}(\mathsf{sid}, \mathsf{sid}_R)$

$(h_R, s, r) \leftarrow \mathsf{DB}_{\mathrm{R}}^{\mathsf{acc}}[\mathsf{sid}_R]$;
$\mathsf{ok} \leftarrow \mathcal{F}_{\mathsf{accSMT}}(\mathsf{prove\text{-}order\text{-}1}, \mathsf{sid}, \mathsf{sid}_R)$
$P \Longrightarrow : m$;
$h \leftarrow H_2(m, r)$;
~~$\pi \leftarrow \mathcal{F}_{\mathsf{wNIZK}}(\mathsf{Prove}, \mathsf{sid}, (h, h_R, m), (s, r))$~~
$\pi \leftarrow \mathsf{DHP}^H[r \,|\, {}^{h_R, h}_{g, g_m}|]$
$\Longrightarrow J : \pi, h$;
return ok;

$\mathsf{CheatDetect_J}(\mathsf{sid})$

1 : $(R, P, h_R, (h_P, m)) \leftarrow \mathcal{F}_{\mathsf{accSMT}}(\mathsf{prove\text{-}order}, \mathsf{sid})$
2 : $R \Longrightarrow : \pi, h$;
3 : ~~**assert** $1 = \mathcal{F}_{\mathsf{wNIZK}}(\mathsf{Verify}, \mathsf{sid}, (h, h_R, m), \pi)$~~
4 : **assert** $1 = \mathsf{ChP}^H[\pi \,|\, {}^{h_R, h}_{g, g_m}|]$
5 : $\Longrightarrow S : h_P, m$;
6 : $S \Longrightarrow : (h_1, \ldots, h_m)$;
7 : **if** $h_P \neq H_3(h_1, \ldots, h_m)$ **or** $\forall i : h_i \neq h$ **then**
8 : **return** $\mathsf{dis}(S)$;
9 : **else return** $\emptyset$;

$\mathsf{CheatDetect_S}(\mathsf{sid})$

$J \Longrightarrow : h_P, m$;
assert $\exists \mathsf{sid}_S : \mathsf{DB}_{\mathrm{S}}^{\mathsf{acc}}[\mathsf{sid}_S] = (h_P, m)$;
$(r_1, \ldots, r_m) \leftarrow$ **values** $(\mathsf{DB}_{\mathrm{S}})$;
~~$\Longrightarrow J : H_2(m, r_1), \ldots, H_2(m, r_m)$;~~
$\Longrightarrow J : g_m^{r_1}, \ldots, g_m^{r_m}$;
return ok;

Fig. 26. Cheat detection subroutine (DDH version).

- The verification runtime is much faster, as the DDH proof generation is much cheaper, and these proofs are as compact as groth16 proofs.
- The revocation is slower for DDH, and overhead increases with the number of revoked tokens. This is because, for m revoked elements, the server needs to perform m group exponentiations to re-compute the entire revocation database. In out experiments, a single revocation takes ca 42 minutes for $m = 1000000$, while it takes less than second for hash-based approach. The current results are comparable to those reported in the non-optimised test application of [27]. It is possible that we would get better times for DDH using another library or some additional optimisations.

A visual comparison of some benchmarks of the two approaches is given in Figure 27.

C.3 Security of the Updated Protocol

There are at least the following additional weaknesses compared to the hash version:

Computation times (ms)

m	0	1	10	100	1000	10000	100000	1000000
issue	0.393	–	–	–	–	–	–	–
revoke	1.28	1.57	3.63	23.7	228	2120	20300	150000
unotify	0.550	0.575	0.573	0.632	1.33	8.20	54.8	617
vnotify	0.116	0.122	0.131	0.197	0.887	5.90	54.2	616
verify	0.156	1.04	1.04	1.04	1.04	1.07	1.49	5.67

Communication (bytes)

m	0	1	10	100	1000	10000	100000	1000000
issue	32	–	–	–	–	–	–	–
revoke	240	–	–	–	–	–	–	–
unotify	146	210	786	6,546	64,146	640,146	6,400,146	64,000,146
vnotify	146	210	786	6,546	64,146	640,146	6,400,146	64,000,146
verify	194	–	–	–	–	–	–	–

Table 4. Revocation protocol benchmarks for DDH version. Notation "–" represents numbers that are the same as for $m = 0$.

- During revocation, a corrupted server can compute x from r. This induces an additional leakage $\mathcal{L}^{\{S\}}_{revoke}(R, x) = \{x\}$. In practice, it would mean that collaborating issuer and server would learn what has been revoked.
- While DDH based solution provides desired unlinkability in stand-alone model [27], is more difficult to come up with a proof in UC model. During adaptive user corruption, as well as during revocation in a corrupted server case, the adversary learns r. While in the hash-based protocol we also assume that corrupted server and corrupted RP can match presentations and revocations, in the proof we could *program* the hashes to be able to proceed with the simulation. In the DDH solution, we cannot re-program the previously simulated values g_m^r. If they have been random, the adversary sees that they are not equal g_m^r, which leads to simulation failure. This is the main reason behind giving up with a UC proof for the DDH based version.
- The DDH (and the discrete logarithm) problem is not hard for quantum computers, while for hash functions no attacks are known yet [29].

Nevertheless, it has been shown in [27] the DDH-based approach is compatible with [10], providing a natural way to achieve unlinkability. So both solutions have their pros and cons.

D Full Proof of Accountable Transmission Functionality (Theorem 2)

Let us fix an arbitrary adversary $\mathcal{A}$. We describe the simulator $\mathcal{S}_{\mathsf{accSMT}}$ who is mediating communication between $\mathcal{A}$ and the adversarial interface of $\mathcal{F}_{\mathsf{accSMT}}$, and is trying to convince $\mathcal{A}$ that it interacts with a real protocol Π_{accSMT}, simulating messages that $\mathcal{A}$ would obtain from Π_{accSMT}. We prove that no $\mathcal{Z}$ can distinguish whether it is interacting with the real protocol Π_{accSMT}, or ideal functionality $\mathcal{F}_{\mathsf{accSMT}}$ extended with the simulator $\mathcal{S}_{\mathsf{accSMT}}$.

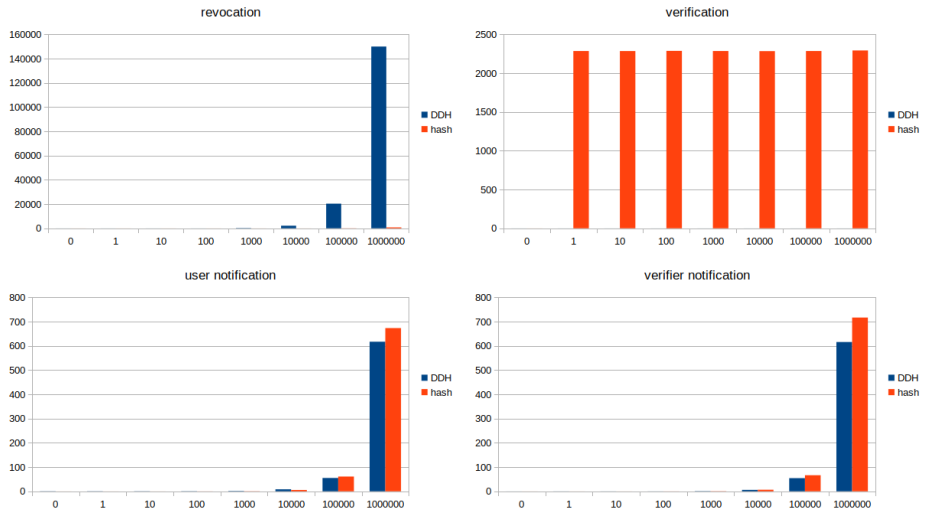

Fig. 27. Comparison of running times of DDH and hash based solutions. The x-axis represents the number of revoked tokens, and y-axis the time in milliseconds.

D.1 The Simulator

In the beginning, $\mathcal{S}_{\mathsf{accSMT}}$ initialises the set of corrupted parties $\mathcal{C} = \emptyset$.

General notes. If all involved parties are corrupted, there is nothing to simulate to $\mathcal{A}$. Moreover, there is nothing to exchange with $\mathcal{F}_{\mathsf{accSMT}}$ as well, as the internal state of $\mathcal{F}_{\mathsf{accSMT}}$ does not change in such cases. Hence, we only consider simulation of cases where at least one involved party is honest.

Simulator summary. Since $\mathcal{F}_{\mathsf{accSMT}}$ sends all inputs to $\mathcal{A}$ (there is no privacy in $\mathcal{F}_{\mathsf{accSMT}}$), the simulator $\mathcal{S}_{\mathsf{accSMT}}$ just follows the routines of Π_{accSMT}, forwarding inputs of corrupted parties to $\mathcal{F}_{\mathsf{accSMT}}$. It then needs to be proved that the outputs of $\mathcal{F}_{\mathsf{accSMT}}$ and the simulated protocol are the same. The difference may come from (1) server providing a different output due to asynchronous read and write requests, and (2) different verdicts of the judge. This is where assumptions like not writing during active reading sessions, and particular correlations between the timeout, the clock/timer shifts and the access time of $\mathcal{Z}$ become important.

Corrupt. Corruption messages are forwarded to $\mathcal{F}_{\mathsf{accSMT}}$ and recorded by $\mathcal{S}_{\mathsf{accSMT}}$.

1. On message $(\mathsf{corrupt}, P)$ from $\mathcal{A}$, forward $(\mathsf{corrupt}, P)$ to $\mathcal{F}_{\mathsf{accSMT}}$.
2. Add P to the set $\mathcal{C}$.

After P gets corrupted, the functionality $\mathcal{F}_{\mathsf{accSMT}}$ sends to $\mathcal{S}_{\mathsf{accSMT}}$ all inputs that P received from $\mathcal{Z}$, and all outputs that P received from $\mathcal{Z}$.$\mathcal{S}_{\mathsf{accSMT}}$ receives

control over the inputs and outputs of P. The adversary expects to receive also the contents of DB_P, containing all the signatures that the receiver P has received from honest senders so far, which $\mathcal{S}_{\mathsf{accSMT}}$ has simulated on behalf of an honest sender already when the message has been sent.

Initialisation. Whenever the initialisation phase is to be started in the real protocol:

1. Create a local table $\mathsf{T}_{\mathrm{sign}}$ for storing signatures of $\mathcal{F}_{\mathsf{cert}}$.
2. Create a local table T_{R} for the log of successful outputs of $\mathcal{F}_{\mathsf{accSMT}}$.

At any time after the initialisation. Whenever $\mathcal{A}$ sends an input $(\mathsf{sign}, \mathsf{sid}, m)$ or $(\mathsf{verify}, \mathsf{sid}, m, \sigma)$ to $\mathcal{F}_{\mathsf{cert}}$ on behalf of corrupted party P, simulate $\mathcal{F}_{\mathsf{cert}}$ by asking adversary to provide signature and verify signature while consulting internal table $\mathsf{T}_{\mathrm{sign}}$.

Send. Only a honest sender needs to be simulated.
Honest sender ($S \notin \mathcal{C}$).
On input $(\mathsf{send\text{-}req}, \mathsf{sid}, R, \mathsf{msg})$ from $\mathcal{F}_{\mathsf{accSMT}}$:

1. Simulate $\mathcal{G}_{\mathsf{clock}}^{S,0}$ on input $(\mathsf{GetTime}, \mathsf{sid})$ from S.
   (a) Send $(\mathsf{Sleep}, \mathsf{sid})$ to $\mathcal{A}$.
   (b) Wait for a response of the form $(\mathsf{Wake}, \mathsf{sid}, \sigma')$ from $\mathcal{A}$.
   (c) If $\sigma' = \bot$ or $|\sigma'| > 0$, set $t_S = \bot$ and stop the simulation.
   (d) Send $\mathsf{GetTime}$ to $\mathcal{G}_{\mathsf{refClock}}$ to receive G. Compute $t_S \leftarrow G + \sigma'$.
2. Simulate $\mathcal{F}_{\mathsf{cert}}$ on input $(\mathsf{sign}, \mathsf{sid}, (R, \mathsf{msg}, t_S))$ from S.
   Get the output σ of $\mathcal{F}_{\mathsf{cert}}$.
3. Simulate S sending $(\mathsf{msg}, t_S, \sigma)$ to P.
4. Write $\mathsf{T}_{\mathrm{W}}[\mathsf{sid}] \leftarrow (R, \mathsf{msg}, t_S)$.

Receive. Only a honest receiver needs to be simulated.
Honest receiver ($R \notin \mathcal{C}$).
On input $(\mathsf{receive\text{-}req}, \mathsf{sid})$ from $\mathcal{F}_{\mathsf{accSMT}}$:

1. Simulate $\mathcal{F}_{\mathsf{timer}}^{R,\Delta_R,\Sigma_R}$ on input $(\mathsf{Start}, \mathsf{sid})$ from R:
   (a) Send $\mathsf{GetTime}$ to $\mathcal{G}_{\mathsf{refClock}}$, getting time G'.
   (b) Record (G', sid).
2. Simulate P receiving $(\mathsf{msg}, t_S, \sigma)$ (potentially chosen by $\mathcal{A}$) from S.
3. Simulate $\mathcal{F}_{\mathsf{cert}}$ on input $(\mathsf{verify}, \mathsf{sid}, (R, \mathsf{msg}, t_S), \sigma)$ from P.
4. Simulate $\mathcal{G}_{\mathsf{clock}}^{R,\Sigma}$ on input $(\mathsf{GetTime}, \mathsf{sid})$ from R.
   (a) Send $(\mathsf{Sleep}, \mathsf{sid})$ to $\mathcal{A}$.
   (b) Wait for a response of the form $(\mathsf{Wake}, \mathsf{sid}, \sigma')$ from $\mathcal{A}$.
   (c) If $\sigma' = \bot$ or $|\sigma'| > \Sigma$, set $t_R = \bot$ and stop the simulation.
   (d) Send $\mathsf{GetTime}$ to $\mathcal{G}_{\mathsf{refClock}}$ to receive G. Compute $t_R \leftarrow G + \sigma'$.
5. Simulate $\mathcal{F}_{\mathsf{timer}}^{R,\Delta_R,\Sigma_R}$ on input $(\mathsf{TimeElapsed}, \mathsf{sid})$ from R:

   (a) Send $\mathsf{GetTime}$ to $\mathcal{G}_{\mathsf{refClock}}$. Denote its response as G.
   (b) Compute $\delta = G - G' + \sigma_C$ for the previously recorded (G', sid).
   (c) If the assertion $\delta \leq \Delta_C$ and $\sigma_C \leq \Sigma_C$ fails, take $\delta \leftarrow \perp$.
6. Assert $t_S \leq t_R + \Sigma$ and $t_S \geq t_R - (\delta + \sigma)$.
   If the assertion fails, return $\perp$.
   // It may fail even for honest S and R, due to network delay.
7. Send $(\mathsf{receive\text{-}ok}, \mathsf{sid}, \mathsf{msg})$ to $\mathcal{F}_{\mathsf{accSMT}}$.
8. Write $\mathsf{T}_{\mathrm{R}}[\mathsf{sid}] \leftarrow (R, \mathsf{msg}, t_S, \sigma)$.

Prove message sent. As there are reliable channels between R and J, the adversary is not notified of communication if all involved parties are honest, so we only consider simulation where $R \in \mathcal{C}$. We will nevertheless consider the other cases in the proof of indistinguishability, as we need to ensure that the outputs will be the same in the real and the simulated protocols.

Honest judge and corrupted receiver ($J \notin \mathcal{C}$ and $R \in \mathcal{C}$).
On input $(\mathsf{prove\text{-}sent\text{-}req}, \mathsf{sid}, \_)$ from $\mathcal{F}_{\mathsf{accSMT}}$:

1. Simulate J receiving $(\mathsf{msg}, t, \sigma)$ from R (chosen by $\mathcal{A}$).
2. Simulate $\mathcal{F}_{\mathsf{cert}}$ on input $(\mathsf{verify}, \mathsf{sid}, (R, \mathsf{msg}, t), \sigma)$ from J.
   If verification fails, send $(\mathsf{prove\text{-}sent\text{-}ok}, \mathsf{sid}, \perp)$ to $\mathcal{F}_{\mathsf{accSMT}}$.
3. Send $(\mathsf{prove\text{-}sent\text{-}ok}, \mathsf{sid}, \mathsf{msg})$ to $\mathcal{F}_{\mathsf{accSMT}}$.

Corupted judge and honest receiver ($J \in \mathcal{C}$ and $R \notin \mathcal{C}$).
On input $(\mathsf{prove\text{-}sent\text{-}req}, \mathsf{sid}, \mathsf{sid}')$ from $\mathcal{F}_{\mathsf{accSMT}}$:

1. Take an entry $(R, \mathsf{msg}, t, \sigma)$ from $\mathsf{T}_{\mathrm{R}}[\mathsf{sid}']$.
2. Simulate R sending $(\mathsf{msg}, t, \sigma)$ to J.
3. Send $(\mathsf{prove\text{-}sent\text{-}ok}, \mathsf{sid}, \mathsf{msg})$ to $\mathcal{F}_{\mathsf{accSMT}}$.

Prove message order. Similarly to the previous case, omit all-honest-parties case. However, now we assume that either R_1 or R_2 (or both) is corrupted.

Honest judge and corrupted receiver ($J \notin \mathcal{C}$ and ($R_1 \in \mathcal{C}$ or $R_2 \in \mathcal{C}$)).
On input $(\mathsf{prove\text{-}order\text{-}req}, \mathsf{sid}, \mathsf{sid}_1, \mathsf{sid}_2)$ from $\mathcal{F}_{\mathsf{accSMT}}$:

1. If $R_i \in \mathcal{C}$, let $\mathcal{A}$ choose $(\mathsf{msg}_i, t_i, \sigma_i)$.
   Otherwise, take an entry $(R_i, \mathsf{msg}_i, t_i, \sigma_i)$ from $\mathsf{T}_{\mathrm{R}}[\mathsf{sid}_i]$.
2. Let $(\mathsf{msg}_1, t_1, \sigma_1)$ and $(\mathsf{msg}_2, t_2, \sigma_2)$ be received by J.
3. Simulate $\mathcal{F}_{\mathsf{cert}}$ on input $(\mathsf{verify}, \mathsf{sid}, (R_i, \mathsf{msg}_i, t_i), \sigma_i)$ from J for $i \in \{1, 2\}$.
   If either verification fails, send $(\mathsf{prove\text{-}order\text{-}ok}, \mathsf{sid}, \perp)$ to $\mathcal{F}_{\mathsf{accSMT}}$.
4. Send $(\mathsf{prove\text{-}sent\text{-}ok}, \mathsf{sid}, \mathsf{msg}_1, \mathsf{msg}_2)$ to $\mathcal{F}_{\mathsf{accSMT}}$.

Corupted judge and at least one honest receiver ($J \in \mathcal{C}$ and ($R_1 \notin \mathcal{C}$ or $R_2 \notin \mathcal{C}$)).
On input $(\mathsf{prove\text{-}sent\text{-}req}, \mathsf{sid}, \mathsf{sid}')$ from $\mathcal{F}_{\mathsf{accSMT}}$:

1. If $R_i \in \mathcal{C}$, let $\mathcal{A}$ choose $(\mathsf{msg}_i, t_i, \sigma_i)$.
   Otherwise, take an entry $(R_i, \mathsf{msg}_i, t_i, \sigma_i)$ from $\mathsf{T}_{\mathrm{R}}[\mathsf{sid}_i]$.
2. Simulate R_i sending $(\mathsf{msg}_i, t_i, \sigma_i)$ to J.
3. Send $(\mathsf{prove\text{-}sent\text{-}ok}, \mathsf{sid}, \mathsf{msg}_1, \mathsf{msg}_2)$ to $\mathcal{F}_{\mathsf{accSMT}}$.

D.2 Indistinguishability

To prove indistinguishability of the real protocol and the simulation, we first state and prove some helpful lemmas.

Lemma 1. *Let R_1 and R_2 be two honest receivers. Assume $\mathcal{Z}$ makes* receive *requests with time offset $> \Delta_R$. Then, if R_1 makes* receive *request* before R_2, *and responses of both R_1 and R_2 are not $\perp$, then R_1 gets the response* before R_2. *This holds for both the real and the simulated execution.*

Proof. Assuming that time delays between executed code lines is negligible, the receiver R_1 sends Start command to the timer functionality at the time $t^0_{R_1}$ at which an honest party R_1 made a query to $\mathcal{F}_{\mathsf{accSMT}}$ from $\mathcal{Z}$.

Execution of R_1 can only be successful if the time elapsed from $t^0_{R_1}$ does not exceed Δ_R. Hence, R_1 gets a (successful) response latest at the time $t^0_{R_1} + \Delta_R$. On the other hand, by assumption, R_2 may only access $\mathcal{F}_{\mathsf{accSMT}}$ at time $t^0_{R_2} > t^0_{R_1} + \Delta_R$, so it returns a response later, even if everything happens without any delays for it.

In a simulation, assuming that time delay in communication between $\mathcal{F}_{\mathsf{accSMT}}$ and $\mathcal{S}_{\mathsf{accSMT}}$ is negligible, the simulator sends Start command as soon as R_1 has provided input to $\mathcal{F}_{\mathsf{accSMT}}$, and returns approval message to $\mathcal{F}_{\mathsf{accSMT}}$ at time $t^0_{R_1} + \Delta_R$ latest. Assuming that the time spent on the subsequent check made by $\mathcal{F}_{\mathsf{accSMT}}$ is negligible, the output is returned to R_2 at time $t^0_{R_1} + \Delta_R$ latest as well. □

Lemma 2. *Let R_1 and R_2 be two honest parties. Assume $\mathcal{Z}$ makes* receive *requests with time offset $\Delta_{ACC} > \Delta_R + 2\Sigma_R + 4\Sigma$. Let $\mathsf{T}'_{\mathrm{R}} = [(P, \mathit{msg}, t, \sigma) \in \mathsf{T}_{\mathrm{R}} \mid P \in \{R_1, R_2\}]$ for T_{R} constructed by the simulator. Then, the entries of T'_{R} comes in ascending order by value t.*

Proof. Consider two consequent records $(R_1, \mathsf{msg}, t_1, \sigma_1)$ and $(R_2, \mathsf{msg}, t_2, \sigma_2)$ of T'_{R}. Let $t^0_{R_1}$ and $t^0_{R_2}$ be the actual starting times of R_1 and R_2 respectively. By Lemma 1, the party that accesses $\mathcal{F}_{\mathsf{accSMT}}$ earlier will also get the response earlier, so we have $t^0_{R_1} < t^0_{R_2}$.

For the party R_1, we find an *upper* bound on t_1. During simulation, $\mathcal{S}_{\mathsf{accSMT}}$ has sent TimeElapsed command to the timer functionality instantly after measuring the time t_{R_1}. Let $t^1_{R_1}$ be the actual time of measurement for the party R_1. Taking into account clock error Σ and timer error Σ_R (which allows that the *actual* elapsed time $t^1_{R_1} - t^0_{R_1}$ was up to $\Delta_R + \Sigma_R$), we get $t_{R_1} \le t^1_{R_1} + \Sigma \le t^0_{R_1} + \Delta_R + \Sigma_R + \Sigma$. Additionally, $\mathcal{S}_{\mathsf{accSMT}}$ has verified that $t_1 \le t_{R_1} + \Sigma$ for the recorded timestamp t_1. We get $t_1 \le t^0_{R_1} + \Delta_R + \Sigma_R + 2\Sigma$.

For the party R_2, we will find a *lower* bound on t_2. The simulator $\mathcal{S}_{\mathsf{accSMT}}$ has verified that $t_2 \ge t_{R_2} - (\delta + \Sigma)$ for the recorded timestamp t_2. Since $t_{R_2} \ge t^1_{R_2} - \Sigma$, and $t^1_{R_2} \ge t^0_{R_2} + \delta - \Sigma_R$, we have $t_2 \ge t^1_{R_2} - \Sigma - \delta - \Sigma \ge t^0_{R_2} + \delta - \Sigma_R - \Sigma - \delta - \Sigma \ge t^0_{R_2} - \Sigma_R - 2\Sigma$.

By assumption, $t^0_{R_2} - t^0_{R_1} \ge \Delta_{ACC}$. From the upper bound on t_1, we get $t_1 \le t^0_{R_2} - \Delta_{ACC} + \Delta_R + \Sigma_R + 2\Sigma$. From the lower bound on t_2, we get $t^0_{R_2} \le t_2 +$

$\Sigma_R+2\Sigma$. Overall, $t_1 \leq t_2-\Delta_{ACC}+\Delta_R+2\Sigma_R+4\Sigma$. For $\Delta_{ACC} > \Delta_R+2\Sigma_R+4\Sigma$, get $t_1 < t_2$. □

We now prove that no $\mathcal{Z}$ that makes receive requests on behalf of honest parties with time offset $\Delta_{ACC} > \Delta_P+2\Sigma_P+4\Sigma$ can distinguish whether it interacts with the real protocol Π_{accSMT} or the ideal functionality $\mathcal{F}_{\mathsf{accSMT}}$ extended with the simulator $\mathcal{S}_{\mathsf{accSMT}}$ described in Sec. D.1.

Send and Receive $\mathcal{S}_{\mathsf{accSMT}}$ internally runs the routines of the protocol Π_{accSMT} in a straightforward manner, getting all the inputs from $\mathcal{F}_{\mathsf{accSMT}}$. Hence, the messages that $\mathcal{S}_{\mathsf{accSMT}}$ exchanges with $\mathcal{A}$ are the same in the real protocol and the simulation. Moreover, the final outputs obtained by $\mathcal{S}_{\mathsf{accSMT}}$ for the *simulated* protocol are the same as in the *real* protocol. Additionally, we need to ensure that the subsequent checks of $\mathcal{F}_{\mathsf{accSMT}}$ succeed as well.

The simulator covers all failures of the real protocol (like too long time delay or failed signature check) by letting $\mathcal{F}_{\mathsf{accSMT}}$ simultaneously fail using the adversary's interface. Hence, it suffices to show that all successful runs of Π_{accSMT} will be successful in the simulation.

Honest sender. For an honest sender, the real protocol will only terminate if the message has actually been sent to the receiver, meaning that the input $(\mathsf{send\text{-}req}, \mathsf{sid}, R, \mathsf{msg})$ has come from $\mathcal{Z}$. An honest R starts listening for a message only if $(\mathsf{receive}, \mathsf{sid}, R)$ has come from $\mathcal{Z}$. The simulator simulates receiving iff it has received notifications about both such inputs from $\mathcal{F}_{\mathsf{accSMT}}$ (or, if R is controlled by the adversary, only the first one).

In the end, $\mathcal{S}_{\mathsf{accSMT}}$ will only send a confirmation message $(\mathsf{receive\text{-}ok}, \mathsf{sid}, \mathsf{msg})$ to $\mathcal{F}_{\mathsf{accSMT}}$ if a message msg has indeed been sent by the honest sender. Assuming that the underlying communication functionality of the real protocol does not allow repetition attacks, the message has not been output by $\mathcal{F}_{\mathsf{accSMT}}$ to R yet. Hence, all checks by $\mathcal{F}_{\mathsf{accSMT}}$ succeed, and it outputs msg to R.

Corrupted sender. Only the case of an honest receiver R is relevant for the simulation. In this case, in the real protocol, R should only terminate if $\mathcal{A}$ timely sends an appropriate message to it. $\mathcal{F}_{\mathsf{accSMT}}$ does not set any additional restrictions on msg' chosen by the simulator in a corrupted sender case.

As the result, regardless of the sender corruption, the table T_R of $\mathcal{S}_{\mathsf{accSMT}}$ stores records $(R, \ldots, \mathsf{msg}, \ldots)$ of exactly those messages msg that have been received by an honest receiver R.

Prove transmission We split the proof into mutually exclusive cases which correspond to intuitive actual situations. Assume that $(\mathsf{prove\text{-}send}, \mathsf{sid}, \ldots)$ has been provided as input of some honest party. The outputs to $\mathcal{Z}$ are only relevant for an honest J.

Transmissions to an honest party are accountable. Let R be honest. Assume that $(\mathsf{sent}, \mathsf{sid}', \mathsf{msg})$ has indeed been output to R. In this case, $\mathcal{F}_{\mathsf{accSMT}}$ outputs ok to an honest J. In the real protocol, J outputs ok if all of the following succeeds:

1. An honest R has found a suitable message in its database.
   // An honest party would not output $(\mathsf{sent}, \mathsf{sid}', \mathsf{msg})$ otherwise.
2. J has successfully received a proper message from R.
   // This is ensured by reliable channels between the parties and the judge.
3. Server signature on the message provided by R verifies correctly.
   // An honest receiver has verified the signature in advance.

Hence, J will also output ok in the real protocol.

If the message $(\mathsf{sent}, \mathsf{sid}', \mathsf{msg})$ has *not* been output to R, then the first check by R fails in the real protocol. In this case, $\mathcal{F}_{\mathsf{accSMT}}$ outputs $\perp$ as well.

Only actual transmissions can proven. Let S be honest. Assume that there has been no actual message transmission. That is, a message $(\mathsf{send}, \mathsf{sid}', R, \mathsf{msg})$ has not been input by S before. In this case, $\mathcal{F}_{\mathsf{accSMT}}$ outputs $\perp$ to J.

- Let R be honest. It can only proceed with the database record extraction if a corresponding message $(\mathsf{sent}, \mathsf{sid}', \mathsf{msg})$ has been previously output to R before. As S is honest, this is possible only in the case when $(\mathsf{send}, \mathsf{sid}', R, \mathsf{msg})$ has been input by S before. This leads to a contradiction, so J outputs $\perp$ in the real protocol.
- Let R be corrupted. In the simulated protocol, $\mathcal{F}_{\mathsf{accSMT}}$ will definitely output $\perp$ to J. In the real protocol, J will output ok if it receives a message (R, msg, t) with a valid signature σ of S. An honest server only signs (R, msg, t) if a message $(\mathsf{send}, \mathsf{sid}', R, \mathsf{msg})$ has been input to S. This leads to a contradiction, so J outputs $\perp$ in the real protocol.

Other transactions. In all other cases, $\mathcal{F}_{\mathsf{accSMT}}$ allows the simulator to choose the output for the party J, so that $\mathcal{F}_{\mathsf{accSMT}}$ outputs the same value as would be output in the real protocol. In particular, a corrupted R may always choose to fail the check even if a message has actually been transmitted, and a corrupted S may always generate the signatures for a corrupted R in advance.

Prove Order We split the proof into mutually exclusive cases which correspond to intuitive actual situations. Assume that $(\mathsf{prove\text{-}order}, \mathsf{sid}, \ldots)$ has been provided as input of some honest party. The outputs to $\mathcal{Z}$ are only relevant for an honest J.

Transmissions to honest parties are accountable. Let R_1 and R_2 be honest. Assume that $(\mathsf{sent}, \mathsf{sid}_1, \mathsf{msg}_1)$ has indeed been output to R_1 before $(\mathsf{sent}, \mathsf{sid}_2, \mathsf{msg}_2)$ has been output to R_2. In this case, $\mathcal{F}_{\mathsf{accSMT}}$ outputs ok to an honest J. In the real protocol, J outputs ok if all of the following succeeds:

1. Honest R_1 and R_2 have found a suitable message in its database.
   // An honest party would not output $(\mathsf{sent}, \mathsf{sid}_i, \mathsf{msg}_i)$ otherwise.

2. J has successfully received proper messages from R_i.
   // This is ensured by reliable channels between the parties and the judge.
3. Server signatures on the message provided by R_i verify correctly.
   // An honest receiver has verified the signature in advance.
4. The inequality $t_1 \leq t_2$ holds.
   // By Lemma 2, we have $t_1 \leq t_2$ iff R_1 has received its output before R_2.

Hence, J will also output ok in the real protocol.

If either message $(\mathsf{sent}, \mathsf{sid}_i, \mathsf{msg}_i)$ has *not* been output to R_i, then the first check by R fails in the real protocol. If R_1 has received its output later than R_2, then the check $t_1 \leq t_2$ fails. in these cases, $\mathcal{F}_{\mathsf{accSMT}}$ outputs $\perp$ as well.

Only actual transmissions can proven. Let S be honest. Assume that there has been no suitable message transmission. That is, no message $(\mathsf{send}, \mathsf{sid}_1, R_1, \mathsf{msg}_1)$ has been input by S *before* a message $(\mathsf{send}, \mathsf{sid}_2, R_2, \mathsf{msg}_2)$. In this case, $\mathcal{F}_{\mathsf{accSMT}}$ outputs $\perp$ to J.

- Let both R_1 and R_2 be honest. They can only proceed with the database record extraction if a corresponding message $(\mathsf{sent}, \mathsf{sid}_i, \mathsf{msg}_i)$ has been previously output to R_i before. As S is honest, this is possible only in the case when $(\mathsf{send}, \mathsf{sid}_i, R_i, \mathsf{msg}_i)$ has been input by S before. However, even if such messages would have been sent, passing the condition $t_1 \leq t_2$ would violate the assumption. This leads to a contradiction, so J outputs $\perp$ in the real protocol.
- Let either R_1 or R_2 be corrupted. In the simulated protocol, $\mathcal{F}_{\mathsf{accSMT}}$ will definitely output $\perp$ to J. In the real protocol, J will output ok if it:
  1. Receives a message $(R_i, \mathsf{msg}_i, t_i)$ with a valid signature σ_i of S.
     // An honest S only signs $(R_i, \mathsf{msg}_i, t_i)$ if a message $(\mathsf{send}, \mathsf{sid}, R_i, \mathsf{msg}_i)$ has been input to S.
  2. The inequality $t_1 \leq t_2$ holds.
     // By Lemma 2, we have $t_1 \leq t_2$ iff R_1 has received its output before R_2. Together with the previous, this leads to contradiction with assumption.

  Hence, J outputs $\perp$ in the real protocol.

Other transactions. In all other cases, $\mathcal{F}_{\mathsf{accSMT}}$ allows the simulator to choose the output for the party J, so that $\mathcal{F}_{\mathsf{accSMT}}$ outputs the same value as would be output in the real protocol. In particular, even a single corrupted R_i may always choose to fail the check even if appropriate messages have actually been transmitted, and a corrupted S may always generate the signatures for a corrupted R_i in advance.

E Full Proof of Revocation Mechanism (Theorem 3)

Formally, we assume that all parties of $\mathcal{F}_{\mathsf{rev}}$ communicate through the message transmission functionality $\mathcal{F}_{\mathsf{SMT}}$ of [13], which we assume to be corrupted if either the sender or the receiver is corrupted. First of all, we provide a more detailed variant of Theorem 1, which we will actually prove.

Theorem 3. *The protocol set Π_{rev} of Sec. 4.3 UC-realises $\mathcal{F}_{rev}$ (depicted in Figures 2, 3) w.r.t. leakages of Table 2 in $\mathcal{F}_{SMT}$, $\mathcal{F}_{cert}$, $\mathcal{F}_{accSMT}$, $\mathcal{F}_{wNIZK}$-hybrid model in presence of $\mathcal{G}_{\mathsf{rpRO}}$.*

First of all, we note that, assuming a particular implementation Π_{accSMT} of $\mathcal{F}_{\mathsf{accSMT}}$, Theorem 3 implies Corollary 1. Intuitively, due to imprecision of clocks and the need to accept network delays, Π_{rev} that is built upon Π_{accSMT} allows corrupted server to conceal the fact of revocation if it has happened slightly *before* the verifier did its notification request, claiming that it took place slightly *after* the verifier request.

Corollary 1. *Assuming environment $\mathcal{Z}$ that keeps time offset between* revoke *and* vnotify *requests at least $\Delta_{ACC} = \Delta_P + 2\Sigma_P + 4\Sigma$, the protocol set Π_{rev} that uses implementation Π_{accSMT} of $\mathcal{F}_{accSMT}$ UC-realises $\mathcal{F}_{rev}$ w.r.t. leakages of Table 2 in $\mathcal{F}_{SMT}$, $\mathcal{F}_{cert}$, $\mathcal{F}_{wNIZK}$, $\mathcal{F}_{\mathsf{timer}}^{P,\Delta_P,\Sigma}$-hybrid model in presence of $\mathcal{G}_{\mathsf{rpRO}}$ and $\mathcal{G}_{\mathsf{clock}}^{P,\Sigma}$.*

Let us fix an arbitrary adversary $\mathcal{A}$. We describe the simulator $\mathcal{S}_{\mathsf{rev}}$ who is mediating communication between $\mathcal{A}$ and the adversarial interface of $\mathcal{F}_{\mathsf{rev}}$, and is trying to convince $\mathcal{A}$ that it interacts with a real protocol Π_{rev}, simulating messages that $\mathcal{A}$ would obtain from Π_{rev}. We prove that no $\mathcal{Z}$ can distinguish whether it is interacting with the real protocol Π_{rev}, or ideal functionality $\mathcal{F}_{\mathsf{rev}}$ extended with the simulator $\mathcal{S}_{\mathsf{rev}}$.

Throughout the proof, let $\mathsf{domain}(f)$ and $\mathsf{range}(f)$ denote the input and the output sets of a function f respectively.

E.1 The Simulator

In the beginning, $\mathcal{S}_{\mathsf{rev}}$ initialises the set of corrupted parties $\mathcal{C} = \emptyset$. The functionality $\mathcal{F}_{\mathsf{rev}}$ is instantiated with a parameter N, and with the leakage functions of Table 2. The simulator is aware of the way in which $\mathcal{F}_{\mathsf{rev}}$ is instantiated.

General notes. Let *involved* parties be those parties who provide inputs to a subroutine of $\mathcal{F}_{\mathsf{rev}}$. If all involved parties are corrupted, there is nothing to simulate to $\mathcal{A}$. Moreover, there is nothing to exchange with $\mathcal{F}_{\mathsf{rev}}$ as well, as the internal state of $\mathcal{F}_{\mathsf{rev}}$ does not change in such cases. Hence, we only consider simulation of cases where at least one involved party is honest.

- If $S \notin \mathcal{C}$, then the simulator focuses on maintaining the hashes $h_1, \ldots, h_m$ in such a way that $h_i = H_2(m, r)$ for $x = H_1(s, r)$ for which the simulator (and the adversary) knows that x has been revoked on the i-th revocation. For unknown x (tokens shared between an honest user and an honest revoker), the simulator generates $h_i \leftarrow\!\$\ \mathsf{range}(H_2)$ during revocation, and $h \leftarrow\!\$\ \mathsf{range}(H_2)$ during presentation. If the same x has been presented to a corrupted verifier repeatedly for the same m, then the simulator uses the same h.
  The simulator interacts with $\mathcal{F}_{\mathsf{rev}}$ in such a way that its inner set X^- is consistent with revocations of corrupted revokers, i.e. it chooses the inputs

x of corrupted revokers as they would be in a real protocol. In some cases, it is impossible, as $\mathcal{A}$ may provide an input r for which no token $x = H_1(s, r)$ has been issued yet. in this case, $\mathcal{S}_{\mathsf{accSMT}}$ does not add anything to X^-, but instead remembers to fail any matching x during verification as an additional failure.
- If $S \in \mathcal{C}$, the simulator focuses on locally maintaining a set X^- of tokens x whose revocation has had effect on at least one honest party. It immediately adds to X^- all tokens revoked by honest revokers. If the revoker is corrupted, even if x has been legitimately added to X^- of $\mathcal{F}_{\mathsf{rev}}$ during a revocation session, a server may still retract it as far as this x has not had effect on any honest party. The latter happens when x gets included into the values shown to an honest user or an honest verifier during their notification sessions. In such cases, simulator adds x to $\mathcal{F}_{\mathsf{rev}}$, triggering revocation by providing inputs on behalf of corrupted R and S.

Initialisation. Whenever the initialisation phase is to be started in the real protocol:

1. Initialise empty table T_{R} for $\mathcal{F}_{\mathsf{accSMT}}$.
2. Send $(\mathsf{Setup}, \mathsf{sid})$ to $\mathcal{A}$ through $\mathcal{F}_{\mathsf{wNIZK}}$. Upon receiving the message $(\mathsf{Algorithms}, \mathsf{sid}, \mathsf{SimProve}, \mathsf{SimExtract})$ from $\mathcal{A}$, store it.
3. Initialise empty table T_{π} for $\mathcal{F}_{\mathsf{wNIZK}}$.
4. For all $1 \leq j \leq N$ generate $x_j \leftarrow\!\!\$\ \mathsf{range}(H_1)$. Let $\mathcal{X} = \{x_1, \ldots, x_N\}$.
5. Send $(\mathsf{init}, \mathsf{sid}, \mathcal{X})$ to $\mathcal{F}_{\mathsf{rev}}$.
6. Initialise local empty tables DB_{P}, $\mathsf{DB}'_{\mathrm{P}}$, $\mathsf{DB}^{\mathsf{acc}}_{\mathrm{P}}$ for all parties $P \in \mathcal{P}$ that will participate in the protocol, and $\mathsf{DB}_{\mathrm{REV}}$ for the server.

Corruption messages are forwarded to $\mathcal{F}_{\mathsf{rev}}$ and recorded by $\mathcal{S}_{\mathsf{rev}}$.

1. On message $(\mathsf{corrupt}, P)$ from $\mathcal{A}$, forward $(\mathsf{corrupt}, P)$ to $\mathcal{F}_{\mathsf{rev}}$.
2. Add P to the set $\mathcal{C}$.

At any time after the initialisation.

Queries to $\mathcal{G}_{\mathsf{rpRO}}$. The adversary may make queries to $\mathcal{G}_{\mathsf{rpRO}}$, to evaluate oracles H_1 and H_2 on any input of its choice, updating the RO tables. Additionally, it can make $\mathsf{Observe}$ and $\mathsf{Program}$ queries to $\mathcal{G}_{\mathsf{rpRO}}$. Since $\mathcal{A}$ is allowed to make programming quires, for the hash values received from the adversary, simulator verifies whether this value was programmed or not. This gives simulator ability to reject adversarially generated values.

Queries to $\mathcal{F}_{cert}$. Whenever $\mathcal{A}$ sends an input $(\mathsf{sign}, \mathsf{sid}, m)$ or $(\mathsf{verify}, \mathsf{sid}, m, \sigma)$ to $\mathcal{F}_{\mathsf{cert}}$ on behalf of corrupted party P, simulate by asking adversary to provide corresponding signature or to verify signature message pair using internal table $\mathsf{T}_{\mathrm{sign}}$.

Queries to $\mathcal{F}_{wNIZK}$. The adversary may make $\mathsf{Prove}, \mathsf{Verify}$ queries to $\mathcal{F}_{\mathsf{wNIZK}}$ on behalf of a corrupted party. If $P = U \in \mathcal{C}$, simulate this interaction as follows:

1. Receive $(\mathsf{Prove}, \mathsf{sid}, x, w)$ from $\mathcal{A}$ on behalf of corrupted P as input to $\mathcal{F}_{\mathsf{wNIZK}}$.
2. If $\mathcal{R}(x, w) = 1$, compute $\pi \leftarrow \mathsf{SimProve}(x)$.
3. Save (x, π) to T_π. Output $(\mathsf{Proof}, \mathsf{sid}, \pi)$ to $\mathcal{A}$.

If $V \in \mathcal{C}$, simulate this interaction as follows:

1. Receive $(\mathsf{Verify}, \mathsf{sid}, x, \pi)$ from $\mathcal{A}$ on behalf of corrupted V as input to $\mathcal{F}_{\mathsf{wNIZK}}$.
2. If $(x, \pi) \in \mathsf{T}_\pi$, output $(\mathsf{Verification}, \mathsf{sid}, x, \pi, 1)$ to $\mathcal{A}$.
3. Otherwise, compute $w \leftarrow \mathsf{SimExtract}(x, \pi)$.
   (a) If $\mathcal{R}(x, w) = 1$, then store (x, π) in T_π.
   (b) If $(w = \mathsf{maul} \wedge (x, \cdot) \in \mathsf{T}_\pi)$ then store (x, π) in T_π.
   (c) If $(x, \pi) \in \mathsf{T}_\pi$, output $(\mathsf{Verification}, \mathsf{sid}, x, \pi, 1)$ to $\mathcal{A}$.
   Otherwise, output $(\mathsf{Verification}, \mathsf{sid}, x, \pi, 0)$ to $\mathcal{A}$.

Issuing. The simulator acts depending on the corruption of U and R.

Honest U and honest R ($U \notin \mathcal{C}$, $R \notin \mathcal{C}$).

1. Wait for $(\mathsf{issue\text{-}req}, \mathsf{sid}, (U, R))$ from $\mathcal{F}_{\mathsf{rev}}$.
2. Generate random $s \leftarrow\!\!\$\, \mathcal{R}_S$ and $r' \leftarrow\!\!\$\, \mathcal{R}_R$.
3. Query $\mathcal{G}_{\mathsf{rpRO}}$ on $(\mathsf{hash}, \mathsf{sid}, (1, U, r'))$ to obtain r.
4. Send $(\mathsf{issue\text{-}req\text{-}ok}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{rev}}$.
5. Simulate U sending a message s, r' to R.
6. Upon receiving $(\mathsf{issue\text{-}resp\text{-}u}, \mathsf{sid})$ from $\mathcal{F}_{\mathsf{rev}}$, send $(\mathsf{issue\text{-}resp\text{-}u\text{-}ok}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{rev}}$.
7. Simulate R receiving a message s, r' from U.
8. Upon receiving $(\mathsf{issue\text{-}resp\text{-}r}, \mathsf{sid})$ from $\mathcal{F}_{\mathsf{rev}}$, send $(\mathsf{issue\text{-}resp\text{-}r\text{-}ok}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{rev}}$.

The adversary sees only the length of the message sent from U to R, so the simulated s and r' may further be ignored.

Corrupted U and honest R ($U \in \mathcal{C}$, $R \notin \mathcal{C}$).

1. Get from $\mathcal{A}$ instruction for U to establish an issuing session sid with R.
2. Send $(\mathsf{issue}, \mathsf{sid}, R)$ to $\mathcal{F}_{\mathsf{rev}}$ as input of U.
3. Wait for $(\mathsf{issue\text{-}req}, \mathsf{sid})$ from $\mathcal{F}_{\mathsf{rev}}$.
4. Wait until $\mathcal{A}$ chooses message s, r' for R on behalf of U.
5. Simulate R receiving a message s, r' from U.
6. Query $\mathcal{G}_{\mathsf{rpRO}}$ on $(\mathsf{hash}, \mathsf{sid}, (1, U, r'))$ to obtain r and on input $(\mathsf{sid}, (1, s, r))$ to obtain x.
7. Store $\mathsf{rnd}_C[x] \leftarrow r$.
   // Having $x = x'$ and $\mathsf{rnd}_C[x] \neq \mathsf{rnd}_C[x']$ would mean finding the second preimage, so rnd_C is well-defined.
8. Send $(\mathsf{issue\text{-}req\text{-}ok}, \mathsf{sid}, x)$ to $\mathcal{F}_{\mathsf{rev}}$.
9. Upon receiving $(\mathsf{issue\text{-}resp\text{-}r}, \mathsf{sid})$ from $\mathcal{F}_{\mathsf{rev}}$, send $(\mathsf{issue\text{-}resp\text{-}r\text{-}ok}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{rev}}$.

Honest U and corrupted R ($U \notin \mathcal{C}$, $R \in \mathcal{C}$).

1. Get from $\mathcal{A}$ instruction for R to establish an issuing session sid with U.
2. Send $(\mathsf{issue}, \mathsf{sid}, U)$ to $\mathcal{F}_{\mathsf{rev}}$ as input of R.
3. Wait for $(\mathsf{issue\text{-}req}, \mathsf{sid}, x)$ from $\mathcal{F}_{\mathsf{rev}}$.
4. Generate random $s \leftarrow\!\$\, \mathcal{R}_S$ and $r' \leftarrow\!\$\, \mathcal{R}_R$.
5. Query $\mathcal{G}_{\mathsf{rpRO}}$ on $(\mathsf{sid}, (1, U, r'))$ to obtain r and program $\mathcal{G}_{\mathsf{rpRO}}$ by giving it input $(\mathsf{program\text{-}RO}, \mathsf{sid}, (1, s, r), x)$. This should succeed with big probability as s and r are freshly and uniformly randomly chosen.
6. Simulate U sending a message s, r' to R.
7. Send $(\mathsf{issue\text{-}req\text{-}ok}, \mathsf{sid}, x)$ to $\mathcal{F}_{\mathsf{rev}}$.
8. Store $\mathsf{rnd}_H[x] \leftarrow r$.
   // Having $x = x'$ and $\mathsf{rnd}_H[x] \neq \mathsf{rnd}_H[x']$ would mean finding the second preimage, so rnd_H is well-defined.
9. Upon receiving $(\mathsf{issue\text{-}resp\text{-}u}, \mathsf{sid})$ from $\mathcal{F}_{\mathsf{rev}}$, send $(\mathsf{issue\text{-}resp\text{-}u\text{-}ok}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{rev}}$.

Since s, r' are freshly generated by $\mathcal{S}_{\mathsf{rev}}$, and $\mathcal{F}_{\mathsf{rev}}$ chooses x as a unique element of $\mathcal{X}$, it is unlikely that H_1 has already been programmed for the input s, r or the output x.

Revocation. The simulator acts depending on the corruption of R and S.

Honest server and honest revoker ($S \notin \mathcal{C}$, $R \notin \mathcal{C}$).

1. Wait for input $(\mathsf{revoke\text{-}req}, \mathsf{sid}, L)$ from $\mathcal{F}_{\mathsf{rev}}$ such that L contains $R \notin \mathcal{C}$.
2. Extract R and a bit $b = (x \notin X^-) \wedge (R \in \mathcal{R}_x)$ from L.
3. Assert $b = \mathsf{true}$. This implies that R would proceed in the real protocol.
4. Check whether x is included into L.
   - If it is, since $S \notin \mathcal{C}$, it could only have leaked via $\mathcal{L}^U_{revoke}$ for $U \in \mathcal{U}_x \cap \mathcal{C}$. Since $R \notin \mathcal{C}$ and $R \in \mathcal{R}_x$, there exists $r \leftarrow \mathsf{rnd}_C[x]$.
     // An honest revoker R revokes a token x of a corrupted user U.
   - Otherwise, take $r \leftarrow \perp$.
     // An honest revoker R revokes a token of an honest user U.
5. Store $\mathsf{r}[m] \leftarrow r$.
6. Simulate R sending r to S.
7. Simulate S receiving r from R.
8. Let $m \leftarrow \mathbf{size}\ (\mathsf{DB}_{\mathrm{REV}}) + 1$.
9. Sample $s_R \leftarrow\!\$\, \mathcal{R}_S$.
10. Compute h_R:
    - If $\mathsf{r}[m] \neq \perp$, query $\mathcal{G}_{\mathsf{rpRO}}$ on input $(\mathsf{hash}, \mathsf{sid}, (1, s_R, r))$ to obtain h_R
    - Otherwise, sample $h_R \leftarrow\!\$\, \mathsf{range}(H_1)$
11. Simulate sending s_R to R.
12. Simulate sending h_R to R using $\mathcal{F}_{\mathsf{accSMT}}$.
13. For all $1 \leq i \leq m$:
    - If $\mathsf{r}[i] \neq \perp$, query $\mathcal{G}_{\mathsf{rpRO}}$ on input $(\mathsf{hash}, \mathsf{sid}, (2, m, \mathsf{r}[i]))$ to obtain h_i
    - Otherwise, sample $h_i \leftarrow\!\$\, \mathsf{range}(H_2)$
14. Rewrite contents of $\mathsf{DB}_{\mathrm{REV}}$ to $(h_1, \ldots, h_m)$.
15. Send $(\mathsf{revoke\text{-}req\text{-}ok}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{rev}}$.
16. Simulate R receiving s_R from S.

17. Simulate R receiving h_R from S via $\mathcal{F}_{\mathsf{accSMT}}$.
18. Simulate positive assertion by honest R without actually evaluating $H_1(s_R, r)$.
19. Assign $\mathsf{DB}_{\mathrm{R}}^{\mathsf{acc}}[\mathsf{sid}] \leftarrow h_R$.
    // Note that s_R and r are not stored, as these are unknown.
20. Upon receiving $(\mathsf{revoke\text{-}resp\text{-}r}, \mathsf{sid})$ from $\mathcal{F}_{\mathsf{rev}}$, send $(\mathsf{revoke\text{-}resp\text{-}r\text{-}ok}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{rev}}$.

Honest server and corrupted revoker ($S \notin \mathcal{C}$, $R \in \mathcal{C}$).

1. Get from $\mathcal{A}$ instruction for R to establish a revocation session sid.
2. Wait until $\mathcal{A}$ chooses message r for S on behalf of R.
3. Check if there is any $x \in \mathsf{domain}(\mathsf{rnd}_H)$ such that $\mathsf{rnd}_H[x] = r$.
   - If there is such an x, then it is unique, since every output of rnd_H (value r) is computed by querying $\mathcal{G}_{\mathsf{rpRO}}$ on input $(\mathsf{sid}, 1, U, r')$ for a freshly sampled $r' \leftarrow\!\!\$\, \mathcal{R}_R$.
     // A corrupted revoker (on any corrupted party with whom that revoker has shared r) revokes x generated by an honest user.
   - If there is no such x, then r has *not* been generated by an honest user. Take $x \leftarrow \emptyset$.
     // A corrupted party either revokes x generated by a corrupted user, or provided r that does not correspond to any token (yet). The simulator ensures that failures of corrupted users will be covered in the verification phase as an additional failure, even if the corresponding x does not belong to X^- of $\mathcal{F}_{\mathsf{rev}}$.
4. Send $(\mathsf{revoke}, \mathsf{sid}, x)$ to $\mathcal{F}_{\mathsf{rev}}$ as input of R.
5. Wait for $(\mathsf{revoke\text{-}req}, \mathsf{sid}, L)$ from $\mathcal{F}_{\mathsf{rev}}$.
6. Simulate S receiving r (chosen by $\mathcal{A}$) from R.
7. Let $m \leftarrow \mathbf{size}\ (\mathsf{DB}_{\mathrm{REV}}) + 1$.
8. Store $\mathsf{r}[m] \leftarrow r$.
9. Sample $s_R \leftarrow\!\!\$\, \mathcal{R}_S$.
10. Query $\mathcal{G}_{\mathsf{rpRO}}$ on input $(\mathsf{hash}, \mathsf{sid}, (1, s_R, \mathsf{r}[m]))$ to obtain h_R
11. Simulate sending s_R to R.
12. Simulate sending h_R to R using $\mathcal{F}_{\mathsf{accSMT}}$.
13. For all $1 \le i \le m$:
    - If $\mathsf{r}[i] \neq \perp$, query $\mathcal{G}_{\mathsf{rpRO}}$ on input $(\mathsf{hash}, \mathsf{sid}, (2, m, \mathsf{r}[i]))$ to obtain h_i
    - Otherwise, sample $h_i \leftarrow\!\!\$\, \mathsf{range}(H_2)$.
14. Rewrite contents of $\mathsf{DB}_{\mathrm{REV}}$ to $(h_1, \ldots, h_m)$.
15. Send $(\mathsf{revoke\text{-}req\text{-}ok}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{rev}}$.

Corrupted server and honest revoker ($S \in \mathcal{C}$, $R \notin \mathcal{C}$).

1. Get from $\mathcal{A}$ an instruction for S to establish a revocation session sid.
2. Send $(\mathsf{revoke}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{rev}}$ as input of S.
3. Wait for $(\mathsf{revoke\text{-}req}, \mathsf{sid}, L)$ from $\mathcal{F}_{\mathsf{rev}}$ such that L contains $R \notin \mathcal{C}$.
4. Extract R and a bit $b = (x \notin X^-) \wedge (R \in \mathcal{R}_x)$ from L.
5. Assert $b = \mathsf{true}$. This implies that R would proceed in the real protocol.
6. If x can be extracted from L:

- If $\mathsf{rnd}_C[x]$ is defined, take $r \leftarrow \mathsf{rnd}_C[x]$.
  // In this case, x has been chosen by a corrupted user for an honest revoker $R \in \mathcal{R}_x$ who is now trying to revoke it.
- Otherwise, if $\mathsf{rev\_sids}(x)$ can be extracted from L, take $r \leftarrow \mathsf{opened}[\mathsf{sid}]$ that has already been simulated for any $\mathsf{sid} \in \mathsf{rev\_sids}(x)$.
  // In this case, the revoker has already tried to revoke x before, and has shown r to the server, but the revocation has failed.
- Otherwise, sample $s \leftarrow\!\!\$\ \mathcal{R}_S$ and $r \leftarrow\!\!\$\ \mathcal{R}_R$. Program $\mathcal{G}_{\mathsf{rpRO}}$ by sending input $(\mathsf{program\text{-}RO}, \mathsf{sid}, (1, s, r), x)$. This should succeed with big probability as s and r are freshly and uniformly randomly chosen. Assign $\mathsf{opened}[\mathsf{sid}] = r$.
  // In this case, an honest revoker revokes an honest user's token, which has already been shown to a corrupted verifier. It is the first time r is simulated to $\mathcal{A}$.
  For all $h = \mathsf{h}[x, m']$ defined for some $1 \leq m' < m$, program $\mathcal{G}_{\mathsf{rpRO}}$ by sending input $(\mathsf{program\text{-}RO}, \mathsf{sid}, (2, m', r), h)$.
  // Related hashes of the form $H_2(1, r), \ldots, H_2(m-1, r)$ have already been presented to some corrupted verifier, and need to be adjusted. The way in which $\mathsf{h}[x, m']$ are constructed is more precisely described in the simulation of verification.

7. Otherwise, if x is unknown:
   - If $\mathsf{rev\_sids}(x)$ can be extracted from L, take $r \leftarrow \mathsf{opened}[\mathsf{sid}]$ that has already been simulated for any $\mathsf{sid} \in \mathsf{rev\_sids}(x)$.
   - Otherwise, $r \leftarrow\!\!\$\ \mathcal{R}_R$. Assign $\mathsf{opened}[\mathsf{sid}] = r$.

   // In this case, an honest revoker revokes an honest user's token, which has not been presented to a corrupted party yet.
8. Simulate R sending r to S.
9. Assign $S_{ID}^{revoke} \leftarrow S_{ID}^{revoke} \cup \{\mathsf{sid}\}$.
10. Send $(\mathsf{revoke\text{-}req\text{-}ok}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{rev}}$.
11. Simulate R receiving s_R from S.
12. Simulate R receiving h_R from S via $\mathcal{F}_{\mathsf{accSMT}}$.
13. Perform assertion $h_R = H_1(s_R, r)$ on behalf of R.
    For this, make query to $\mathcal{G}_{\mathsf{rpRO}}$ on input $(\mathsf{sid}, 1, s_R, r)$ and verify that the result is equal to h_R. Additionally, verify that $(\mathsf{sid}, 1, s_R, r)$ has not been programmed by querying $\mathcal{G}_{\mathsf{rpRO}}$ on input $(\mathsf{IsProgrammed}, (\mathsf{sid}, 1, s_R, r))$ and aborting if the result is not $(\mathsf{IsProgrammed}, 0)$. If $H_1(s_R, r)$ has not been programmed yet, then adversary unlikely comes up with a matching h_R. Therefore, probability that R aborts is negligible.
14. Assign $\mathsf{DB}_{\mathrm{R}}^{\mathsf{acc}}[\mathsf{sid}] \leftarrow h_R$.
15. Upon receiving $(\mathsf{revoke\text{-}resp\text{-}r}, \mathsf{sid})$ from $\mathcal{F}_{\mathsf{rev}}$, send $(\mathsf{revoke\text{-}resp\text{-}r\text{-}ok}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{rev}}$.

User notification. The simulator acts depending on the corruption of U and S.

Honest server and honest user ($S \notin \mathcal{C}$, $U \notin \mathcal{C}$).

1. Wait for input $(\mathsf{unotify\text{-}req}, \mathsf{sid}, (U, b))$ from $\mathcal{F}_{\mathsf{rev}}$.
2. Take $(h_1, \ldots, h_m) \leftarrow \mathbf{values}\ (\mathsf{DB}_{\mathrm{REV}})$.
3. Simulate S sending $h_1, \ldots, h_m$ to U.
4. Send $(\mathsf{unotify\text{-}req\text{-}ok}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{rev}}$.
5. Simulate U receiving $h_1, \ldots, h_m$ from S.
6. Simulate U receiving (h_U, m) from S through $\mathcal{F}_{\mathsf{accSMT}}$.
7. Simulate a successful hash check of h_U, add $\mathsf{DB}_{\mathrm{U}}^{\mathsf{acc}}[\mathsf{sid}] \leftarrow h_U$.
8. Verify that $b = \mathsf{true}$, i.e. x has been previously issued to U by $\mathcal{F}_{\mathsf{rev}}$.
9. Simulate local operations by U for a randomly chosen value r, without actually evaluating $H_2(m, r)$.
10. Assign $\mathsf{m}[U] \leftarrow m$.
11. Upon receiving $(\mathsf{unotify\text{-}resp\text{-}u}, \mathsf{sid})$ from $\mathcal{F}_{\mathsf{rev}}$, send $(\mathsf{unotify\text{-}resp\text{-}u\text{-}ok}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{rev}}$.

Honest server and corrupted user ($S \notin \mathcal{C}$, $U \in \mathcal{C}$).

1. Get from $\mathcal{A}$ instruction for U to establish a notification session sid.
2. Send $(\mathsf{unotify}, \mathsf{sid}, x)$ to $\mathcal{F}_{\mathsf{rev}}$ as input of U for arbitrary $x \in \mathcal{X}$.
3. Wait for $(\mathsf{unotify\text{-}req}, \mathsf{sid}, L)$ from $\mathcal{F}_{\mathsf{rev}}$.
4. Take $(h_1, \ldots, h_m) \leftarrow \mathbf{values}\ (\mathsf{DB}_{\mathrm{REV}})$.
5. Simulate S sending $h_1, \ldots, h_m$ to U.
6. Simulate S sending $(H_3(h_1, \ldots, h_m), m)$ to U through $\mathcal{F}_{\mathsf{accSMT}}$.
7. Send $(\mathsf{unotify\text{-}req\text{-}ok}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{rev}}$.

Corrupted server and honest user ($S \in \mathcal{C}$, $U \notin \mathcal{C}$).

1. Get from $\mathcal{A}$ instruction for S to establish a notification session sid.
2. Send $(\mathsf{unotify}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{rev}}$ as input of S.
3. Wait for $(\mathsf{unotify\text{-}req}, \mathsf{sid}, (U, b))$ from $\mathcal{F}_{\mathsf{rev}}$.
4. Wait until $\mathcal{A}$ chooses message $h_1, \ldots, h_m$ for U on behalf of S.
5. Simulate U receiving $h_1, \ldots, h_m$ (chosen by $\mathcal{A}$) from S.
6. Simulate U receiving (h_U, m) (chosen by $\mathcal{A}$) from S through $\mathcal{F}_{\mathsf{accSMT}}$.
7. Verify $h_U = H_3(h_1, \ldots, h_m)$. Add $\mathsf{DB}_{\mathrm{U}}^{\mathsf{acc}}[\mathsf{sid}] \leftarrow h_U$.
8. Verify that $b = \mathsf{true}$, i.e. x has been previously issued to U by $\mathcal{F}_{\mathsf{rev}}$.
9. Simulate local operations by U for a randomly chosen value r, without actually evaluating $H_2(m, r)$.
10. Assign $\mathsf{m}[U] \leftarrow m$.
11. Let $S_{ID} \leftarrow S_{ID}^{revoke}$.
    // Take S_{ID}-s of all revocations by honest revokers made so far. We initially assume that all revocations are concealed, and then remove one by one those actually covered by h_i.
12. For all $r \in \mathsf{range}(\mathsf{rnd}_H) \cap \mathsf{range}(\mathsf{opened})$:
    (a) Check if $h_i = H_2(m, r)$ for some i. This is done by querying $\mathcal{G}_{\mathsf{rpRO}}$ on $(\mathsf{sid}, 2, m, r)$ and checks that the result is equal to h_i. Check that $(\mathsf{sid}, 2, m, r)$ is not programmed by giving $\mathcal{G}_{\mathsf{rpRO}}$ input $(\mathsf{IsProgrammed}, (\mathsf{sid}, 2, m, r))$ and ignore this x if the result is not $(\mathsf{IsProgrammed}, 0)$. If $H_2(m, r)$ has not been programmed yet, then there is negligible probability that adversary comes up with a matching h_i.

(b) If such h_i does exist:
- If $\mathsf{opened}[\mathsf{sid}'] = r$ is defined for some sid', remove sid' from S_{ID}.
  // This means that this record is not con
- Otherwise, if $U \in \mathcal{U}_x$ and $\mathsf{rnd}_H[x] = r$, take $\mathsf{sid}' \leftarrow (\mathsf{sid}, x)$, $R \in \mathcal{R}_x \cap \mathcal{C}$, and send to $\mathcal{F}_{\mathsf{rev}}$ an input $(\mathsf{revoke}, \mathsf{sid}', x)$ of R, and input $(\mathsf{revoke}, \mathsf{sid}')$ of S.

13. For all $\mathsf{sid}' \in S_{ID}$, if there exists $\mathsf{DB}_{\mathrm{R}}^{\mathsf{acc}}[\mathsf{sid}'] \neq \perp$ for some $R \notin \mathcal{C}$, send $(\mathsf{brokenCons}, U, R)$ to $\mathcal{F}_{\mathsf{rev}}$.
    // Any concealed revocation by an honest revoker is accountable.
14. Send $(\mathsf{unotify\text{-}req\text{-}ok}, \mathsf{sid}, S_{ID})$ to $\mathcal{F}_{\mathsf{rev}}$.
15. Upon receiving $(\mathsf{unotify\text{-}resp\text{-}u}, \mathsf{sid})$ from $\mathcal{F}_{\mathsf{rev}}$, send $(\mathsf{unotify\text{-}resp\text{-}u\text{-}ok}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{rev}}$.

Verifier notification The simulator acts depending on the corruption of V and S.

Honest server and honest verifier ($S \notin \mathcal{C}$, $V \notin \mathcal{C}$).

1. Wait for input $(\mathsf{vnotify\text{-}req}, \mathsf{sid}, V)$ from $\mathcal{F}_{\mathsf{rev}}$.
2. Take $(h_1, \ldots, h_m) \leftarrow \mathbf{values}\ (\mathsf{DB}_{\mathrm{REV}})$.
3. Simulate S sending $h_1, \ldots, h_m$ to V.
4. Simulate V receiving $h_1, \ldots, h_m$ from S.
5. Simulate V receiving (h_V, m) from S through $\mathcal{F}_{\mathsf{accSMT}}$.
6. Simulate a successful hash check of h_V, add $\mathsf{DB}_{\mathrm{V}}^{\mathsf{acc}}[\mathsf{sid}] \leftarrow h_V$.
7. Assign $\mathsf{m}[V] \leftarrow m$.
8. Assign $\mathsf{DB}_{\mathrm{V}} \leftarrow (h_1, \ldots, h_m)$.
9. Send $(\mathsf{vnotify\text{-}req\text{-}ok}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{rev}}$.
10. Upon receiving $(\mathsf{vnotify\text{-}resp\text{-}v}, \mathsf{sid})$ from $\mathcal{F}_{\mathsf{rev}}$, send $(\mathsf{vnotify\text{-}resp\text{-}v\text{-}ok}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{rev}}$.

Honest server and corrupted verifier ($S \notin \mathcal{C}$, $V \in \mathcal{C}$).

1. Get from $\mathcal{A}$ instruction for V to establish a notification session sid.
2. Send $(\mathsf{vnotify}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{rev}}$ as input of V.
3. Wait for $(\mathsf{vnotify\text{-}req}, \mathsf{sid}, L)$ from $\mathcal{F}_{\mathsf{rev}}$.
4. Take $(h_1, \ldots, h_m) \leftarrow \mathbf{values}\ (\mathsf{DB}_{\mathrm{REV}})$.
5. Simulate S sending $h_1, \ldots, h_m$ to V.
6. Simulate S sending $(H_3(h_1, \ldots, h_m), m)$ to U through $\mathcal{F}_{\mathsf{accSMT}}$.
7. Send $(\mathsf{vnotify\text{-}req\text{-}ok}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{rev}}$.

Corrupted server and honest verifier ($S \in \mathcal{C}$, $V \notin \mathcal{C}$).

1. Get from $\mathcal{A}$ instruction for S to establish a notification session sid.
2. Send $(\mathsf{vnotify}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{rev}}$ as input of S.
3. Wait for $(\mathsf{vnotify\text{-}req}, \mathsf{sid}, V)$ from $\mathcal{F}_{\mathsf{rev}}$.
4. Wait until $\mathcal{A}$ chooses message $h_1, \ldots, h_m$ for V on behalf of S.
5. Simulate V receiving $h_1, \ldots, h_m$ (chosen by $\mathcal{A}$) from S.
6. Simulate V receiving (h_V, m) (chosen by $\mathcal{A}$) from S through $\mathcal{F}_{\mathsf{accSMT}}$.
7. Verify $h_V = H_3(h_1, \ldots, h_m)$. Add $\mathsf{DB}_{\mathrm{V}}^{\mathsf{acc}}[\mathsf{sid}] \leftarrow h_V$.

8. Assign $\mathsf{DB}_{\mathrm{V}} \leftarrow (h_1, \ldots, h_m)$.
9. Assign $\mathsf{m}[V] \leftarrow m$.
10. Let $S_{ID} \leftarrow S_{ID}^{revoke}$.
    // Take S_{ID}-s of all revocations by honest revokers made so far. We initially assume that all revocations are concealed, and then remove one by one those actually covered by h_i.
11. For all $r \in \mathsf{range}(\mathsf{rnd}_H) \cap \mathsf{range}(\mathsf{opened})$:
    (a) Check if $h_i = H_2(m, r)$ for some i. This is done by querying $\mathcal{G}_{\mathsf{rpRO}}$ on $(\mathsf{sid}, 2, m, r)$ and checks that the result is equal to h_i. Check that $(\mathsf{sid}, 2, m, r)$ is not programmed by giving $\mathcal{G}_{\mathsf{rpRO}}$ input $(\mathsf{IsProgrammed}, (\mathsf{sid}, 2, m, r))$ and ignore this x if the result is not $(\mathsf{IsProgrammed}, 0)$. If $H_2(m, r)$ has not been programmed yet, then there is negligible probability that adversary comes up with a matching h_i.
    (b) If such h_i does exist:
        – If $\mathsf{opened}[\mathsf{sid}'] = r$ is defined for some sid', remove sid' from S_{ID}.
        – Otherwise, if $U \in \mathcal{U}_x$ and $\mathsf{rnd}_H[x] = r$, take $\mathsf{sid}' \leftarrow (\mathsf{sid}, x)$, $R \in \mathcal{R}_x \cap \mathcal{C}$, and send to $\mathcal{F}_{\mathsf{rev}}$ an input $(\mathsf{revoke}, \mathsf{sid}', x)$ of R, and input $(\mathsf{revoke}, \mathsf{sid}')$ of S.
12. For all $\mathsf{sid}' \in S_{ID}$, if there exists $\mathsf{DB}_{\mathrm{R}}^{\mathsf{acc}}[\mathsf{sid}'] \neq \perp$ for some $R \notin \mathcal{C}$, send $(\mathsf{brokenCons}, U, R)$ to $\mathcal{F}_{\mathsf{rev}}$.
    // Any concealed revocation by an honest revoker is accountable.
13. Send $(\mathsf{vnotify\text{-}req\text{-}ok}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{rev}}$.
14. Upon receiving $(\mathsf{vnotify\text{-}resp\text{-}v}, \mathsf{sid})$ from $\mathcal{F}_{\mathsf{rev}}$, send $(\mathsf{vnotify\text{-}resp\text{-}v\text{-}ok}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{rev}}$.

Verification In this routine, the leakage contains a bit b such that $b = \mathsf{true}$ iff the latest $(\mathsf{unotify}, \mathsf{sid}, x)$ input of U to $\mathcal{F}_{\mathsf{rev}}$ produced an output $(\mathsf{unotify}, \mathsf{sid}, \mathsf{false})$ *after* the latest $(\mathsf{vnotify}, \mathsf{sid}, \mathsf{ok})$ output to V.

Since $\mathcal{A}$ does not have any access to channels between honest parties (excluding honest server S), their interaction does not need to be simulated to $\mathcal{A}$. However, interaction between $\mathcal{S}_{\mathsf{rev}}$ and $\mathcal{F}_{\mathsf{rev}}$ takes place in any case.

Honest user and honest verifier ($U \notin \mathcal{C}$, $V \notin \mathcal{C}$).

1. Wait for input $(\mathsf{verify\text{-}req}, \mathsf{sid}, b)$ from $\mathcal{F}_{\mathsf{rev}}$.
2. Verify that $b = \mathsf{true}$.
3. Take an arbitrary token $x \in \mathcal{X}$ and simulate corresponding h as in the case of corrupted verifier..
   // The only effect that this randomly chosen x has on $\mathcal{Z}$ is whether proofs succeeds of fail. Here we assume that the probabiliy of failure for an honest prover is negligible, i.e. the following check always succeeds regardless of x.
4. Let $x_\pi = (h, x, m)$. Compute $\pi \leftarrow \mathsf{SimProve}(x_\pi)$. Save $(\mathsf{sid}, x_\pi, \pi)$ to T_π. // Simulate a NIZK proof.
5. Send $(\mathsf{verify\text{-}req\text{-}ok}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{rev}}$.
6. Upon receiving $(\mathsf{verify\text{-}resp\text{-}u}, \mathsf{sid})$ from $\mathcal{F}_{\mathsf{rev}}$, send $(\mathsf{verify\text{-}resp\text{-}u\text{-}ok}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{rev}}$.

7. Upon receiving $(\mathsf{verify\text{-}resp\text{-}v}, \mathsf{sid})$ from $\mathcal{F}_{\mathsf{rev}}$, send $(\mathsf{verify\text{-}resp\text{-}v\text{-}ok}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{rev}}$.

Honest user and corrupted verifier ($U \notin \mathcal{C}$, $V \in \mathcal{C}$).

1. Get from $\mathcal{A}$ instruction for V to establish a verification session sid.
2. Send $(\mathsf{verify}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{rev}}$ as input of V.
3. Wait for $(\mathsf{verify\text{-}req}, \mathsf{sid}, L)$ from $\mathcal{F}_{\mathsf{rev}}$.
4. Since $V \in \mathcal{C}$, the token x is included into L. Extract x from L.
5. Simulate V sending m (chosen by $\mathcal{A}$) to the user.
6. If $b = \mathsf{true}$, simulate successful database read and assertion by user.
7. Simulate the value h as follows.
   - If $\mathsf{h}[x, m]$ exists, take $h \leftarrow \mathsf{h}[x, m]$.
     // If the token x has already been presented with the same m, the hash h will be exactly the same.
   - Otherwise, if $\mathsf{rnd}_H[x] = r$ has already been defined, query $\mathcal{G}_{\mathsf{rpRO}}$ on $(\mathsf{hash}, 2, \mathsf{sid}, (m, r))$ to get h. Write $\mathsf{h}[x, m] \leftarrow h$
     // If the value r has already been leaked to a corrupted revoker, use the same r.
   - Otherwise, if $S \in \mathcal{C}$ extract S^x_{ID} from L. If $S^x_{ID} \neq \emptyset$, extract sid' from S^x_{ID}. As $\mathsf{rnd}_H[x]$ has not been defined, x has been issued to an honest revoker, who revokes a token only once, so $S^x_{ID} = \{\mathsf{sid}'\}$, and $\mathsf{opened}[\mathsf{sid}'] = r$ has been defined upon revocation. Query $\mathcal{G}_{\mathsf{rpRO}}$ on $(\mathsf{hash}, 2, \mathsf{sid}, (m, r))$ to get h. Write $\mathsf{h}[x, m] \leftarrow h$.
     // If the value r has already been leaked to a corrupted server during a revocation, use the same r.
   - Otherwise, sample $h \leftarrow\!\!\$\ \mathsf{range}(H_2)$. Write $\mathsf{h}[x, m] \leftarrow h$.
     // In this case, x has been shared between honest U and R, and has not been revoked yet. Whenever $x = H_1(s, r)$ will eventually be programmed (this happens during revocation), the values $\mathsf{h}[x, m]$ will be updated accordingly as described in simulation of revocation.
8. Let $x_\pi = (h, x, m)$. Compute $\pi \leftarrow \mathsf{SimProve}(x_\pi)$. Save $(\mathsf{sid}, x_\pi, \pi)$ to T_π. // Simulate a NIZK proof.
9. Simulate user sending h, x, π to V.
10. Send $(\mathsf{verify\text{-}req\text{-}ok}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{rev}}$.
11. Upon receiving $(\mathsf{verify\text{-}resp\text{-}u}, \mathsf{sid})$ from $\mathcal{F}_{\mathsf{rev}}$, send $(\mathsf{verify\text{-}resp\text{-}u\text{-}ok}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{rev}}$.

Corrupted user and honest verifier ($U \in \mathcal{C}$, $V \notin \mathcal{C}$).

1. Get from $\mathcal{A}$ instruction for U to establish a verification session sid with V.
2. Wait until $\mathcal{A}$ chooses message h, x, π for V on behalf of U.
3. Send $(\mathsf{verify}, \mathsf{sid}, x, V)$ to $\mathcal{F}_{\mathsf{rev}}$ as input of U.
4. Wait for $(\mathsf{verify\text{-}req}, \mathsf{sid}, L)$ from $\mathcal{F}_{\mathsf{rev}}$.
5. Simulate V receiving h, x, π (chosen by $\mathcal{A}$) from U.
6. Let $x_\pi = (h, x, m)$. Simulate verification of $\mathcal{F}_{\mathsf{wNIZK}}$ as follows:
   (a) If $(x_\pi, \pi) \in \mathsf{T}_\pi$, simulate success and proceed.

(b) Otherwise, compute $w \leftarrow \mathsf{SimExtract}(x_\pi, \pi)$.
   i. If $\mathcal{R}(x_\pi, w) = 1$, then store (x_π, π) in T_π.
   ii. If $(w = \mathsf{maul} \wedge (x_\pi, \cdot) \in \mathsf{T}_\pi)$ then store (x_π, π) in T_π.
   iii. If $(x_\pi, \pi) \in \mathsf{T}_\pi$, simulate success and proceed.
   Otherwise, simulate failure and stop simulation of this routine call.

// Verify NIZK proof

7. Simulate comparing h against $h_1, \ldots, h_{m_V} \leftarrow \mathsf{DB}_\mathrm{V}$. // Here corrupted server may have introduced *additional* revocations of corrupted user, which are not covered by $\mathcal{F}_{\mathsf{rev}}$, so we need to inform additional failures to $\mathcal{F}_{\mathsf{rev}}$
8. If all checks succeed, send $(\mathsf{verify\text{-}req\text{-}ok}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{rev}}$.
9. Upon receiving $(\mathsf{verify\text{-}resp\text{-}v}, \mathsf{sid})$ from $\mathcal{F}_{\mathsf{rev}}$, send $(\mathsf{verify\text{-}resp\text{-}v\text{-}ok}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{rev}}$.

Corruption Upon getting $(\mathsf{corrupt})$ message from $\mathcal{A}$ to a party P, if $P \notin \{S, J\}$:

1. Send $(\mathsf{corrupt}, P)$ message to $\mathcal{F}_{\mathsf{rev}}$.
2. Receive from $\mathcal{F}_{\mathsf{rev}}$ all messages input and output by P.
3. Add P to $\mathcal{U}_x$ for $x \in X_U = \{x : (\mathsf{issue\text{-}resp\text{-}u}, \mathsf{sid}, x)$was output by $\mathcal{F}_{\mathsf{rev}}$ to $P\}$.
4. Add P to $\mathcal{R}_x$ for $x \in X_R = \{x : (\mathsf{issue\text{-}resp\text{-}r}, \mathsf{sid}, x)$was output by $\mathcal{F}_{\mathsf{rev}}$ to $P\}$.
5. Receive from $\mathcal{F}_{\mathsf{rev}}$ the set $X_0^- = \{x : P \in \mathcal{U}_x \wedge (\mathsf{revoke}, \mathsf{sid}, x)$was input to $\mathcal{F}_{\mathsf{rev}}\}$.
6. Let $X_1^- = \{x : P \in \mathcal{R}_x \wedge (\mathsf{revoke}, \mathsf{sid}, x)$was input to $\mathcal{F}_{\mathsf{rev}}$ by $P\}$. // Now X_0^- are the tokens revoked *for* P, and X_1^- are the tokens revoked *by* P.
7. For all $x \in X_U \cup X_R$, if $\mathsf{rnd}_H[x]$ or $\mathsf{rnd}_C[x]$ have not been defined yet, the random oracle $\mathcal{G}_{\mathsf{rpRO}}$ needs to be programmed for $x = H(s, r)$ (the issued token), and $h = H_2(m_V, r)$ (hashes presented to a corrupted verifier):
   - Sample fresh randomness $s \in \mathcal{R}_S$.
   - If a message $(\mathsf{revoke}, \mathsf{sid}, x)$ has been input for sid such that $\mathsf{opened}[\mathsf{sid}] = r$ is defined, take this r. Otherwise, sample fresh randomness $r \in \mathcal{R}_R$.
   - Send $(\mathsf{program\text{-}RO}, \mathsf{sid}, (1, s, r), x)$ to $\mathcal{G}_{\mathsf{rpRO}}$.
   - For all $h = \mathsf{h}[x, m_V]$ defined for some m_V, send $(\mathsf{program\text{-}RO}, \mathsf{sid}, (2, m', r), h)$ to $\mathcal{G}_{\mathsf{rpRO}}$.
     // This hash is already programmed in the case $\mathsf{opened}[\mathsf{sid}] = r$ is defined
   - Construct $\mathsf{DB}_\mathrm{P}[x] = (s, r)$.

   There are some additional values that need to be programmed for the given x. If $x \in X_0^- \cup X_1^-$, look for the index i of the revocation on which x has been revoked. Now the random oracle $\mathcal{G}_{\mathsf{rpRO}}$ needs to be programmed for the hashes $h_j = H_2(j, r)$ for $j \in \{1, \ldots, i\}$ (revocation list), and $h'_i = H_1(s_i, r)$ (revocation approval sent to revoker). This is done as follows:
   - Sample fresh randomness s_i.
   - Send $(\mathsf{program\text{-}RO}, \mathsf{sid}, (1, s_i, r), h'_i)$ to $\mathcal{G}_{\mathsf{rpRO}}$.
   - For all $j \in \{1, \ldots, i\}$, send $(\mathsf{program\text{-}RO}, \mathsf{sid}, (2, j, r), h_j)$ to $\mathcal{G}_{\mathsf{rpRO}}$.
     // These hashes are already programmed in the case $\mathsf{opened}[\mathsf{sid}] = r$ is defined
   - Construct $\mathsf{DB}_\mathrm{P}^{\mathsf{acc}}[\mathsf{sid}] = (h'_i, s_i, r)$ (for P acting as a revoker, for a successful revocation sid).

8. For the latest input $(\mathsf{unotify}, \mathsf{sid}, x)$ followed by $(\mathsf{unotify\text{-}resp\text{-}u}, \mathsf{sid}, b)$, assign $\mathsf{DB}'_{\mathrm{P}}[x] = (s, r, b, \mathsf{m}[P])$.
9. Note that the values $\mathsf{DB}^{\mathsf{acc}}_{\mathrm{U}}[\mathsf{sid}]$, $\mathsf{DB}^{\mathsf{acc}}_{\mathrm{V}}[\mathsf{sid}]$, and DB_{V} have already been simulated for honest U and V during the previous user and verifier notification sessions.
10. Send DB_{P}, $\mathsf{DB}'_{\mathrm{P}}$, $\mathsf{DB}^{\mathsf{acc}}_{\mathrm{P}}$ to $\mathcal{A}$.

Detect Cheating On input $(\mathsf{cheat\text{-}detect\text{-}req}, \mathsf{sid}, V, R)$ from $\mathcal{F}_{\mathsf{rev}}$, the simulator acts depending on the corruption of V, R, S, J.

Honest user/verifier ($P \notin \mathcal{C}$).

1. Look for an entry $(h_P, m) = \mathsf{DB}^{\mathsf{acc}}_{\mathrm{P}}[\mathsf{sid}_P]$ simulated for P before. Simulate failure of J if such an entry does not exist.
2. Simulate running $\mathcal{F}_{\mathsf{accSMT}}$ on input $(\mathsf{prove\text{-}order\text{-}2}, \mathsf{sid}, \mathsf{sid}_P)$.
3. Simulate sending m to R.

Honest revoker ($R \notin \mathcal{C}$).

1. Look for an entry $h_R = \mathsf{DB}^{\mathsf{acc}}_{\mathrm{R}}[\mathsf{sid}_R]$ simulated for R before. Simulate failure of J if such an entry does not exist.
2. Simulate running $\mathcal{F}_{\mathsf{accSMT}}$ on input $(\mathsf{prove\text{-}order\text{-}1}, \mathsf{sid}, \mathsf{sid}_R)$.
3. Simulate receiving m from R. If $R \in \mathcal{C}$, receive m from $\mathcal{A}$.
4. Simulate h as follows:
   - If $S \in \mathcal{C}$, take r that has been simulated during revocation session sid_R and compute $h \leftarrow H_2(m, r)$.
     // An honest R reaches this step only if such a revocation has taken place.
   - If $S \notin \mathcal{C}$, take h_i from the list of hashes $(h_1, \ldots, h_m)$ shown to P by honest S on session sid_P, where i has been previously simulated to $\mathcal{A}$ during the revocation session sid_R.
5. Simulate a NIZK proof $\pi = \mathsf{SimProve}(h, h_R, m)$.
6. Simulate sending π, h to J.

Honest server ($S \notin \mathcal{C}$).

1. Simulate receiving (h_P, m) from J.
2. Take $(h_1, \ldots, h_m)$ that have been simulated to $\mathcal{A}$ during the revocation session sid_S, on which $\mathsf{DB}^{\mathsf{acc}}_{\mathrm{S}}[\mathsf{sid}_S] = h_R = H_3(h_1, \ldots, h_m)$ has been recorded. If there is no such session, simulate failure of S.
   // An honest judge will not send (h_P, m) to S unless having verified a signature of S on it. In the real protocol, the server should stop if a corrupted judge attempts to cause some extra leakage by providing malicious h_P and m.
3. Simulate sending $h_1, \ldots, h_m$ to J.

Honest judge ($J \notin \mathcal{C}$).

1. Simulate running $\mathcal{F}_{\mathsf{accSMT}}$ on input $(\mathsf{prove\text{-}order}, \mathsf{sid})$. Let $(R, P, h_R, (h_P, m))$ be the corresponding output.
   // Here the output may depend on inputs of corrupted parties.

2. Simulate receiving π, h from R. If $R \in \mathcal{C}$, this message is chosen by the adversary.
3. Simulate running $\mathcal{F}_{\mathsf{wNIZK}}$ on values h, h_R, m, π. Proceed only if the proof verifies.
4. If $h_p \neq H_3(h_1, \ldots, h_m)$ or $\forall i : h_i \neq h$, then send $(\mathsf{detect\text{-}cheat\text{-}ok}, \mathsf{sid}, \mathsf{dis}(S))$ to $\mathcal{F}_{\mathsf{accSMT}}$. Otherwise, send $(\mathsf{detect\text{-}cheat\text{-}ok}, \mathsf{sid}, \emptyset)$ to $\mathcal{F}_{\mathsf{accSMT}}$.

E.2 Indistinguishability

Programming RO. The simulator $\mathcal{S}_{\mathsf{rev}}$ internally runs the routines of the protocol Π_{rev} in a straightforward manner, using inputs obtained from $\mathcal{F}_{\mathsf{rev}}$. However, there are some values that depend on inputs of honest parties, which $\mathcal{S}_{\mathsf{rev}}$ has not received from $\mathcal{F}_{\mathsf{rev}}$. In particular, the simulator has to replace actual evaluation of certain hashes with random values $h \leftarrow \mathsf{range}(H_2)$ and $h \leftarrow \mathsf{range}(H_1)$, which it can program later through querying $\mathcal{G}_{\mathsf{rpRO}}$. This may happen in the following cases:

- for an honest revoker R, an honest user U, and honest server S in the revocation phase;
- for an honest user U in the verification phase.

We need to be sure that all these hashes can be later programmed correctly. Let us assume that a token x, which belongs to an honest user, has been revoked at the revocation epoch m.

Let $S \notin \mathcal{C}$.

- During revocation of x, the simulator has sampled a random hash instead of making hash evaluation query $H_2(m, r)$ to the $\mathcal{G}_{\mathsf{rpRO}}$. During upcoming revocations, S will sample random hashes instead of making hash evaluation queries $H_2(m+1, r), H_2(m+2, r), \ldots$ as well. Due to the check $r_m \notin (r_1, \ldots, r_{m-1})$ by an honest server, it will not publish repeating values $h_i = H_2(m, r)$ and $h_j = H_2(m, r)$ for $i \neq j$, so all these hashes are derived independently from each other.
- During verification, the check $b = \mathsf{false}$ ensures that an honest U never proceeds with x that has already been revoked by the m-th revocation for which the user makes the proof (note that the server will not lie here, as it is honest). Hence, the hashes related to x that could be released here are $H_2(1, r), \ldots, H_2(m-1, r)$.
- Whenever r gets leaked upon dynamic party corruption, the simulator makes programming queries to the $\mathcal{G}_{\mathsf{rpRO}}$ for all the previously released values $\mathsf{h}[x, m]$, h_i, h'_i to keep their correspondence with r.

Let $S \in \mathcal{C}$.

- During revocation, if the server only learns r, the simulator makes hash evaluation query to $\mathcal{G}_{\mathsf{rpRO}}$ to get $H_2(m, r)$.
- During revocation, if the server learns both x and the corresponding r, the simulator makes programming query to $\mathcal{G}_{\mathsf{rpRO}}$ to set $x = H_1(s, r)$.

– During verification, the simulator only samples a fresh random hash h if (1) neither $\mathsf{rnd}_H[x] = r$ nor $\mathsf{opened}[\mathsf{sid}] = r$ has defined, i.e. r has not been shown to the adversary yet, and (2) x has not been presented for this m yet.
  - If the user repeatedly presents the same x for the same m, the simulator ensures that exactly the same $h = \mathsf{h}[x, m]$ is used each time.
  - Whenever r gets leaked to a corrupted server during the issuing or revocation, or upon dynamic party corruption, the simulator makes programming queries to the $\mathcal{G}_{\mathsf{rpRO}}$ for all the previously released values $\mathsf{h}[x, m]$ to keep their correspondence with r.

In both cases, as m increases with every revocation, the adversary needs to distinguish independently sampled hashes $h_1, h_2, h_3, \ldots$ from the true evaluations of hashes $H_2(1, r), H_2(2, r), H_2(3, r), \ldots$ (released either during revocation or during verification). As H_2 is modelled as global random oracle, the values are independent from each other. Whenever r gets leaked to the adversary, the simulator makes programming query of $h_i = H_2(i, r)$ to $\mathcal{G}_{\mathsf{rpRO}}$. The simulator only releases randomness to the adversary *after* the RO has been programmed according to that randomness. Programming of an RO entry may fail if the adversary has evaluated that entry by chance.

Another possible inconsistency related to hashes is the output of the issuing phase. For honest users, $\mathcal{F}_{\mathsf{rev}}$ samples a random token $x \leftarrow\!\$\ \mathcal{X}$, which is essentially $x \leftarrow\!\$\ \mathsf{range}(H_1)$. In the real protocol, it should be $x = H_1(s, r)$ for freshly sampled s and r. As H_1 is modelled as global random oracle, these values are indistinguishable, as far as s are r not leaked to adversary. Whenever they are leaked, the simulator makes corresponding programming queries (of x) to $\mathcal{G}_{\mathsf{rpRO}}$. Programming of $\mathcal{G}_{\mathsf{rpRO}}$ succeeds unless the environment makes a random oracle query on (s, r), which are uniformly randomly chosen variables. Similar holds for the commitment $h_R = H_1(s_R, r)$ that the revoker gets from the server, and which is sampled randomly at the moment of revocation by an honest revoker and an honest server.

Overall, if $n_{\mathsf{ISS}} < N$ is the total number of issued tokens, and n_{REV} is the total number of revocations, then for every token there are up to n_{REV} hashes to be simulated for the revocation list and the presentations altogether, and the probabiility of attacker hitting one of these hashes by chance is $\frac{n_{\mathsf{ISS}} \cdot n_{\mathsf{REV}}}{\mathcal{R}_R}$. In addition, there are $x = H(s, r)$ and $h_R = H_1(s_R, r)$ for which simulation fails if the attacker guesses s (or s_i) by chance, which happens with probability at most $\frac{n_{\mathsf{ISS}}}{\mathcal{R}_S}$. Assuming that the environment can make up to q queries to RO, the upper bound on probability of simulation failure becomes $\frac{q \cdot n_{\mathsf{ISS}} \cdot n_{\mathsf{REV}}}{\mathcal{R}_R} + \frac{q \cdot n_{\mathsf{ISS}}}{\mathcal{R}_S}$.

Correctness of Simulation Let us now assume that the programming of RO has succeeded. We now prove that the outputs to $\mathcal{Z}$ and $\mathcal{A}$ are indistiguihable in the real and the simulated protocols. We go through all subroutines one by one, discussing where something may fail, seeing the simulation of all routines as a whole.

Issuing. $\mathcal{F}_{\mathsf{rev}}$ checks whether $U' \in \mathcal{U}_x$ for an honest user U', which may introduce additional failures. However, there is no such check in the real protocol. Assume that, for an honest U', the issuing has finished successfully with an output x'. In this case, it should be $x' = H_1(s', r')$ for fresh s' and r'. Now assume that the issuing has been run again for some user U.

- Let U be honest. It chooses $x = H_1(s, r)$ for fresh s and r. The probability that x' and x coincide by chance is $1/|\mathsf{range}(H_1)| = 1/|\mathcal{R}_R|$.
- Let U be corrupted.
  - Let R be honest For the fixed x', the adversary may come up with s and r such that $H_1(s, r) = x'$ either by chance with probability $1/|\mathcal{R}_R|$, or if suitable s and r have been explicitly leaked to the adversary. In the latter case, as an honest R takes $r = H_1(U, r')$, the adversary needs to come up with a suitable r'. It may already know r'' sampled by honest U' such that $r = H_1(U', r'')$, but satisfying $r = H_1(U, r')$ for $U \neq U'$ would mean finding the second preimage. So a suitable r' can only be found by chance with probability $1/|\mathcal{R}_R|$. Altogether, $\mathcal{A}$ may succeed in $x = x'$ with probability at most $2/|\mathcal{R}_R|$.
  - If R is corrupted, then we have the case where both U and R are corrupted. In this case, the outputs of U and R do not depend on the success or failure of $\mathcal{F}_{\mathsf{rev}}$.

Putting aside RO programming related failures, for n_{ISS} issued tokens, the upper bound on probability of simulation failure in the issuing phase is at most $\frac{2n_{\mathsf{ISS}}}{\mathcal{R}_R}$.

Revocation. The simulator ensures that the set X^- of $\mathcal{F}_{\mathsf{rev}}$ contains pairs (x, R) such that revocation of x by R has been approved either by S (if $S \notin \mathcal{C}$), or by R (if $S \in \mathcal{C}$).

- If the revoking party R is honest and has all rights to revoke x, then an entry (x, R) is always added to X^- of $\mathcal{F}_{\mathsf{rev}}$.
- If the revoking party R is corrupted, the simulator needs to reconstruct x from the partial information r that it gets from $\mathcal{A}$, to make revocation of $\mathcal{F}_{\mathsf{rev}}$ consistent with the real protocol. It only considers the case $\mathsf{rnd}_H[x] = r$, i.e. where r was generated by an honest user to a corrupted revoker, and in this case (x, R) is added to X^- of $\mathcal{F}_{\mathsf{rev}}$. Otherwise, the simulator ensures that no token is added to X^- of $\mathcal{F}_{\mathsf{rev}}$. We need to justify that we may ignore the other options besides $\mathsf{rnd}_H[x] = r$:
  - If x has been shared between honest U and R', the probability that $\mathcal{A}$ guesses underlying r by chance is $1/|\mathcal{R}_R|$.
  - If a corrupted user has generated r, then the same value r can be used to prove revocation of many different tokens x by providing a different s. Even if $\mathsf{rnd}_C[x] = r$ has been defined, such x is not unique. In particular, a corrupted user may fail their own verification for *any* x of the form $x = H_1(s, r)$, by choosing an appropriate s. This would potentially cover

the entire set $\mathcal{X}$ of all tokens, and the simulator cannot guess in advance what will be presented. Hence, it will just ensure that $\mathcal{F}_{\mathsf{rev}}$ fails whenever a corrupted user uses the same r in the verification, even if $x \notin X^-$.

Putting aside RO programming related failures, for n_{REV} revocations, the upper bound on probability of simulation failure in the revocation phase is at most $\frac{n_{\mathsf{REV}}}{\mathcal{R}_R}$.

User notification. We prove a helpful lemma first.

Lemma 3. *Let $x \in \mathcal{X}$ be such that $\mathcal{U}_x \cap \mathcal{C} = \emptyset$, and let $x = H_1(s, r)$ hold in the real protocol. Let $h_1, \ldots, h_m$ be generated by the adversary. Let $\overline{X}$ be the set constructed by the simulator in the corrupted server case for the user and the verifier notifications. Then, we have*

$$\exists \mathsf{sid}', R : (X^-[\mathsf{sid}'] = (x, R)) \wedge (\mathsf{sid}' \notin S_{ID}) \iff \exists i : h_i = H_2(m, r) \ .$$

Proof. Let us prove both directions separately.

- Assume that there is no i such that $h_i = H_2(m, r)$. In this case, the check $(X^-[\mathsf{sid}'] = (x, R)) \wedge (\mathsf{sid}' \notin S_{ID})$ fails: even if $X^-[\mathsf{sid}'] = (x, R)$ is true, sid' has not been removed from S_{ID}, as removal only takes place on condition $\exists i : h_i = H_2(m, r)$.
- Assume that $h_i = H_2(m, r)$ for some i. We show that, due to the assumption $\mathcal{U}_x \cap \mathcal{C} = \emptyset$, this is only possible if either $\mathsf{rnd}_H[x] = r$ or $\mathsf{opened}[\mathsf{sid}'] = r$ has been defined in the simulation. In any other case, the adversary would not be able to come up with such h_i:
  - During issuing to a corrupted revoker, the simulator samples $r' \leftarrow \mathcal{R}_R$ and programs $r = H(U', r')$ for a certain user U'. The adversary guesses the same r in advance (before $\mathsf{rnd}_H[x]$ is assigned) with probability $1/|\mathcal{R}_R|$.
  - During revocation with a corrupted server, $r \leftarrow\!\!\$\ \mathcal{R}_R$ and $s \leftarrow\!\!\$\ \mathcal{R}_S$ are sampled by the simulator. The adversary guesses the same r in advance (before $\mathsf{opened}[\mathsf{sid}]$ is assigned) with probability $1/|\mathcal{R}_R|$.

  Now assume $\mathsf{rnd}_H[x] = r$ or $\mathsf{opened}[\mathsf{sid}'] = r$. In this case, $\exists \mathsf{sid}', R : X^-[\mathsf{sid}'] = (x, R)$ is true, as x has already been revoked if $\mathsf{opened}[\mathsf{sid}'] = r$, and the simulator has performed an extra revocation in the case $\mathsf{rnd}_H[x] = r$. In both cases simulator has *not* included corresponding sid' to S_{ID}, so $x \notin S_{ID}$. □

Now let us proceed with the proof of user notification. The output b of $\mathcal{F}_{\mathsf{rev}}$ to an honest user U should be the same as in the real protocol. If $U \notin \mathcal{U}_x$, then both the real and the simulated protocols return $\perp$. Let $U \in \mathcal{U}_x$. Since $\mathcal{F}_{\mathsf{rev}}$ does not allow dishonest user to reuse an honest user's token, we have $\mathcal{U}_x \cap \mathcal{C} = \emptyset$, so we can apply Lemma 3.

Assume that U outputs $b = \mathsf{false}$ in the real protocol. In this case, there is no $h_i = H_2(m, r)$. By Lemma 3, we have $\exists \mathsf{sid}', R : (X^-[\mathsf{sid}'] = (x, R)) \wedge (\mathsf{sid}' \notin S_{ID}) = \mathsf{false}$, so $\mathcal{F}_{\mathsf{rev}}$ also outputs $b = \mathsf{false}$.

Assume that U outputs $b = \mathsf{true}$ in the real protocol. In this case, $h_i = H_2(m, r)$ for some i. By Lemma 3, we have $\exists \mathsf{sid}', R : (X^-[\mathsf{sid}'] = (x, R)) \wedge (\mathsf{sid}' \notin S_{ID}) = \mathsf{true}$, so $\mathcal{F}_{\mathsf{rev}}$ also outputs $b = \mathsf{true}$.

The functionality $\mathcal{F}_{\mathsf{rev}}$ requires that, if the revocation of session sid' is concealed, it should be accountable by the corresponding revoker R. Whenever some sid' of a revocation session by honest R gets concealed, simulator assigns $\mathsf{brokenCons}(U, R) = \mathsf{true}$. We will prove in the indistinguishability of cheating detection that this is indeed correct.

Putting aside RO programming related failures, for n_{REV} revocations, and n_{UNOT} user notifications, the upper bound on probability of simulation failure in the user notification phase is at most $\frac{n_{\mathsf{REV}} \cdot n_{\mathsf{UNOT}}}{\mathcal{R}_R}$.

Verifier notification. Since this subroutine produces no outputs, discussion of properties of X_V^- can be postponed to the verification. Similarly to user notification, whenever some sid' of a revocation session by honest R gets concealed, simulator assigns $\mathsf{brokenCons}(V, R) = \mathsf{true}$. We will prove in the indistinguishability of cheating detection that this is indeed correct.

Putting aside RO programming related failures, for n_{REV} revocations, and N_{VNOT} verifier notifications, the upper bound on probability of simulation failure in the verifier notification phase is at most $\frac{n_{\mathsf{REV}} \cdot N_{\mathsf{VNOT}}}{\mathcal{R}_R}$.

Verification. In the real protocol, an honest V checks whether provided h is included into hashes $h_1, \ldots, h_{m_V}$ of $\mathsf{DB_V}$ that has been constructed during the last notification session.

- Let U be honest. If there has been no prior positive user notification that happened after the the latest verifier notification (satisfying $m_V \leq m_U$), this is reported by $\mathcal{F}_{\mathsf{rev}}$ as a part of leakage L, and a failure is simulated.
  We need to ensure that the subsequent steps of Π_{rev} will fail iff $x \in X_V^-$.
  - The NIZK proofs for the given x using h computed in such a way that they do not equal any of the h_i (they are sampled independently from $\mathsf{range}(H)$ and remain unprogrammed until the r gets revealed), so the proof verification always succeeds.
  - Assuming the proof has been verified, both $\mathcal{F}_{\mathsf{rev}}$ and the real protocol make this comparison based on the previously reported X_V^-.
    * If $S \notin \mathcal{C}$, it has honestly reported to U that $x \notin X^-$. It has also reported $X_V^- = X^-$ to V. Now, since $m_V \leq m_U$, the revocation of verifier was earlier, and token x has not been included into X_V^-, so $x \notin X_V^-$. Simulator has computed $h \leftarrow\!\$\ \mathsf{range}(H_2)$, so h equals some h_i by chance with probability at most $m_V / |\mathcal{R}_R|$.
    * If $S \in \mathcal{C}$, then $X_V^- = \{x | X^-[\mathsf{sid}'] = (x, R) \wedge (\mathsf{sid}' \notin S_{ID})\}$. By Lemma 3 this is true iff $\exists i : h_i = H_2(m, r)$.
- If U is corrupted and V is honest, the simulator performs checks on behalf of an honest V.
  - If at least one revoker $R \in \mathcal{R}_x$ is honest, then $\mathsf{rnd}[x] = r$ has already been programmed. The adversary may cheat if it finds s' and r' such

that $x = H_1(s', r')$, which allows it to present a valid NIZK proof for a hash $h' = H_2(m_V, r')$. However, it would break collision-resistance, and happens with probability $1/|\mathcal{R}_R|$. So we assume $h = H_2(m_V, r)$.
  * If $S \notin \mathcal{C}$, then $X_V^- \leftarrow X^-$. On the other hand, in the real protocol, the server has computed $h_i = H_2(m_V, r)$ for $\mathsf{rnd}[x] = r$ iff $(x, R) \in X^-$ has been true at the time V made its notification query.
  * If $S \in \mathcal{C}$, then $X_V^- = \{x | X^-[\mathsf{sid}'] = (x, R) \wedge (\mathsf{sid}' \notin S_{ID})\}$. By Lemma 3 this is true iff $\exists i : h_i = H_2(m, r)$.

  In both cases, the real and the simulated protocol fail simultaneously.
- Otherwise, the adversary may fix an arbitrary r and construct a proof for $h = H_2(m_V, r)$, making it either pass of fail, depending on the choice of r. Such a proof would work for any $x = H_1(s, r)$, depending on the choice of x. For such x, it can be $x \notin X_V^-$, as x has not been programmed at the moment when X_V^- was created. If $\mathcal{A}$ chooses to make a match $h = h_i$ for such x, the simulator just reports it as additional failure to $\mathcal{F}_{\mathsf{rev}}$.

Putting aside RO programming related failures, for n_{REV} revocations, and n_{VER} verification sessions, the upper bound on probability of simulation failure in the verifier notification phase is at most $\frac{n_{\mathsf{REV}} \cdot n_{\mathsf{VER}}}{\mathcal{R}_R} + n_{\mathsf{VER}} \cdot P_{NIZK}$, where P_{NIZK} is the probability of faking a NIZK proof.

Probability of failure. Overall, in the main phase of the protocol (without cheating detection), assuming that the environment makes at most q queries to $\mathcal{G}_{\mathsf{rpRO}}$, the probability of simulation failure is at most

$$q \cdot \left(\frac{n_{\mathsf{ISS}} \cdot n_{\mathsf{REV}}}{\mathcal{R}_R} + \frac{n_{\mathsf{ISS}}}{\mathcal{R}_S} \right) + \frac{2 \cdot n_{\mathsf{ISS}} + n_{\mathsf{REV}} \cdot (1 + n_{\mathsf{UNOT}} + N_{\mathsf{VNOT}} + n_{\mathsf{VER}})}{\mathcal{R}_R} + n_{\mathsf{VER}} \cdot P_{NIZK}$$

Detect Cheating We split the proof into mutually exclusive cases which correspond to intuitive actual situations.

Cheating is accountable. Assume that $\mathsf{brokenCons}(P, R) = \mathsf{true}$ holds in the simulated protocol. In this case, $\mathcal{F}_{\mathsf{rev}}$ outputs $\mathsf{dis}(S)$, regardless of the choice $\mathcal{C}' \in \{\{S\}, \emptyset\}$ of the simulator. Note that $\mathcal{F}_{\mathsf{rev}}$ allows to assign $\mathsf{brokenCons}(P, R) = \mathsf{true}$ only for $S \in \mathcal{C}$. Also, we may only require accountability in the case $P, R \notin \mathcal{C}$, so it suffices to consider these cases here. In the real protocol, J outputs $\mathsf{dis}(S)$ if all of the following succeeds:

1. J has successfully received $(R, P, h_R, (h_P, m))$ from $\mathcal{F}_{\mathsf{accSMT}}$. For this, the following should have happened in turn for $P, R \notin \mathcal{C}$:
   - There is an entry $\mathsf{DB_R}[\mathsf{sid}_R] = h_R$ previously recorded.
   - There is an entry $\mathsf{DB_P}[\mathsf{sid}_P] = (h_P, m)$ previously recorded.
   - The party R received a message h_R from $\mathcal{F}_{\mathsf{accSMT}}$ *before* the party P received a message (h_P, m) from $\mathcal{F}_{\mathsf{accSMT}}$.

   // As brokenCons(P, R) = true is only defined during a notification session of P, when a token from some *previous* revocation by R is silenced, such entries do exist.

2. J has successfully received a message (π, h) from R.
   // This is ensured by reliable channels between R and J during accountability.
3. The proof π verifies on (h, h_R, m).
   // An honest R takes $h = H_2(m, r)$ for the silenced token randomness r, which satisfies $h_R = H(s_R, r)$ for some s_R that R has used as a witness.
4. One of the following holds:
   (a) The server S has not come up with any $(h_1, \ldots, h_m)$ satisfying $h_P = H_3(h_1, \ldots, h_m)$.
   (b) The server S has provided $(h_1, \ldots, h_m)$, but none of h_i matches h.
   // Hash check against h_P ensures that these are the same values $(h_1, \ldots, h_m)$ that have been shown to the party P. In the simulated protocol, brokenCons$(P, R) =$ true is only possible if $H_2(m, r)$ has not been included into any of the h_i, as is manually checked by the simulator.

The probability that the server cheats without being accountable is hence at most $\frac{n_{\mathsf{REV}}}{\mathcal{R}_R}$, i.e. it has by chance come up with hashes $h_1, \ldots, h_m$ such that $h_i = h$ and $h_P = H_3(h_1, \ldots, h_m)$.

An honest party is never blamed. Assume that S is honest. In this case, only $\mathsf{brokenCons}(P, R) = \mathsf{false}$ in possible the simulated protocol. The functionality $\mathcal{F}_{\mathsf{rev}}$ will not output $\mathsf{dis}(S)$, as this is possible only in the case $\mathsf{brokenCons}(P, R) = \mathsf{true}$, which contradicts the assumption.

In the real protocol, J will output $\mathsf{dis}(S)$ only if it:

1. Receives $(R, P, h_R, (h_P, m))$ from $\mathcal{F}_{\mathsf{accSMT}}$.
   // This means that a revocation by R has indeed happened before a notification of P, whereas $h_R = H_3(s_R, r)$ and $h_P = H_3(h_1, \ldots, h_m)$ for the revoked r and $h_1, \ldots, h_m$ that *do include* the value $h_i = H_2(m, r)$, as honest S does not conceal revocations.
2. J has successfully received a message (π, h) from R, and the proof π verifies on (h, h_R, m).
   // This means that $h = H_2(m, r)$ for the same value r that is committed by h_R.
3. Either $h_P \neq H_3(h_1, \ldots, h_m)$, or $\forall i : h_i \neq h$.
   // An honest server constructs $h_i = H(m, r_i)$, which are exactly the values that it has shown to P during the notification, so $h_P = H_3(h_1, \ldots, h_m)$. Since notification happened after the revocation, $r = r_i$ for some i. Since validity of π ensures that $h = H_2(m, r)$, it should be the case $h = h_i$.

Hence, J outputs $\emptyset$ in this case.

The probability that the honest party will be blamed is at most $P_{NIZK} + P_{SIGN}$, where P_{NIZK} is the probability of getting a fake NIZK proof, and P_{SIGN} the probability of getting a fake signature of the server.

A corrupted party is not protected. If $\mathsf{brokenCons}(P, R) = \mathsf{false}$, but S is corrupted, then there are no guarantees in which verdict simulation ends up, as S may accuse itself, providing signatures on wrong information. Since $\mathcal{S}_{\mathsf{rev}}$ internally runs the routines of the protocol Π_{rev} in a straightforward manner, the set of accused parties $\mathcal{C}'$ (in this case, either $\mathcal{C}' = \{S\}$ or $\mathcal{C}' = \emptyset$) that it informs to $\mathcal{F}_{\mathsf{rev}}$ is the same as the output of J in the real protocol. As by assumption $\mathsf{brokenCons}(P, R) = \mathsf{false}$, $\mathcal{F}_{\mathsf{rev}}$ will also output $\mathsf{dis}(\mathcal{C}')$ to J.