Multi-Party Revocation in Sovrin: Performance through Distributed Trust*

Lukas Helminger[1,2], Daniel Kales[1], Sebastian Ramacher[3][0000−0003−1957−3725], and Roman Walch[1,2]

[1] Graz University of Technology, Graz, Austria
{lukas.helminger,daniel.kales,roman.walch}@iaik.tugraz.at
[2] Know-Center GmbH, Graz, Austria
[3] AIT Austrian Institute of Technology, Vienna, Austria
sebastian.ramacher@ait.ac.at

Abstract. Accumulators provide compact representations of large sets and compact membership witnesses. Besides constant-size witnesses, public-key accumulators provide efficient updates of both the accumulator itself and the witness. However, bilinear group based accumulators come with drawbacks: they require a trusted setup and their performance is not practical for real-world applications with large sets.
In this paper, we introduce multi-party public-key accumulators dubbed *dynamic (threshold) secret-shared accumulators*. We present an instantiation using bilinear groups having access to more efficient witness generation and update algorithms that utilize the shares of the secret trapdoors sampled by the parties generating the public parameters. Specifically, for the q-SDH-based accumulators, we provide a maliciously-secure variant sped up by a secure multi-party computation (MPC) protocol (IMACC'19) built on top of SPDZ and a maliciously secure threshold variant built with Shamir secret sharing. For these schemes, a performant proof-of-concept implementation is provided, which substantiates the practicability of public-key accumulators in this setting.
We explore applications of dynamic (threshold) secret-shared accumulators to revocation schemes of group signatures and credentials system. In particular, we consider it as part of Sovrin's system for anonymous credentials where credentials are issued by the foundation of trusted nodes.

Keywords: multiparty computation, dynamic accumulators, distributed trust, threshold accumulators

1 Introduction

Digital identity management systems become an increasingly important corner stone of digital workflows. Self-sovereign identity (SSI) systems such as Sovrin[4]

* This is the full version of a paper which appears in CT-RSA 2021 – The Cryptographers' Track at the RSA Conference 2021, San Francisco, CA, USA, May 17-21, 2021.

[4] https://sovrin.org/

are of central interest as underlined by a recent push in the European Union for a cross-border SSI system.[5] But all these systems face a similar issue, namely that of efficient revocation. Regardless of whether they are built from signatures, group signatures or anonymous credentials, such systems have to consider mechanisms to revoke a user's identity information. Especially for identity management systems with a focus on privacy, revocation may threaten those privacy guarantees. As such various forms of privacy-preserving revocations have emerged in the literature including approaches based on various forms of deny- or allowlists including [NFHF09, Ver16, BS01, BL11, BL12, ATSM09, CL02, FHM11, NKHF05, NS04, GGM14] among many others.[6]

One promising approach regarding efficiency is based on denylists (or allowlists) via cryptographic accumulators which were introduced by Benaloh and de Mare [BdM93]. They allow one to accumulate a finite set $\mathcal{X}$ into a succinct value called the accumulator. For every element in this set, one can efficiently compute a witness certifying its membership, and additionally, some accumulators also support efficient non-memberships witnesses. However, it should be computationally infeasible to find a membership witness for non-accumulated values and a non-membership witness for accumulated values, respectively. Accumulators facilitate privacy-preserving revocation mechanisms, which is especially relevant for privacy-friendly authentication mechanisms like group signatures and credentials. For a denylist approach, the issuing authority accumulates all revoked users and users prove in zero-knowledge that they know a non-membership witness for their credential. Alternatively, for a allowlist approach, the issuing authority accumulates all users and users then prove in zero-knowledge that they know a membership witness. As both approaches may involve large lists, efficient accumulator updates as well as efficient proofs are important for building an overall efficient system. For example, in Sovrin [KL16] and Hyperledger Indy[7] such an accumulator-based approach with allowlists following the ideas of [GGM14] is used. Their credentials contain a unique revocation ID attribute, i_R, which are accumulated. Each user obtains a membership witness proving that their i_R is contained in the accumulator. Once a credential is revoked, the corresponding i_R gets removed from the accumulator and all users have to update their proofs accordingly. The revoked user is no longer able to prove knowledge of a verifying witness and thus verification fails.

Accumulators are an important primitive and building block in many cryptographic protocols. In particular, Merkle trees [Mer89] have seen many applications in both the cryptographic literature but also in practice. For example, they have been used to implement Certificate Transparency (CT) [Lau14] where all issued certificates are publicly logged, i.e., accumulated. Accumulators also find application in credentials [CL02], ring, and group signatures [LLNW16, DRS18], anonymous cash [MGGR13], authenticated hash tables [PTT08], among many

[5] https://essif-lab.eu/

[6] For a discussion of approaches for group signatures, see, e.g., [SSU16].

[7] https://hyperledger-indy.readthedocs.io/projects/hipe/en/latest/text/0011-cred-revocation/README.html

others. When looking at accumulators deployed in practice, many systems rely on Merkle trees. Most prominently we can observe this fact in CT. Even though new certificates are continuously added to the log, the system is designed around a Merkle tree that gets recomputed all the time instead of updating a dynamic public-key accumulator. The reason is two-fold: first, for dynamic accumulators to be efficiently computable, knowledge of the secret trapdoor used to generate the public parameters is required. Without this information, witness generation and accumulator updates are simply too slow for large sets (cf. [KOR19]). Secondly, in this setting it is of paramount importance that the log servers do not have access to the secret trapdoor. Otherwise malicious servers would be able to present membership witnesses for every certificate even if it was not included in the log.

The latter issue can also be observed in other applications of public-key accumulators. The approaches due to Garman et al. [GGM14] and the one used in Sovrin rely on the Strong-RSA and q-SDH accumulators, respectively. Both these accumulators have trapdoors: in the first case the factorization of the RSA modulus and in the second case a secret exponent. Therefore, the security of the system requires those trapdoors to stay secret. Hence, these protocols require to put significant trust in the parties generating the public parameters. If they would act maliciously and not delete the secret trapdoors, they would be able to break all these protocols in one way or another. To circumvent this problem, Sander [San99] proposed a variant of an RSA-based accumulator from RSA moduli with unknown factorization. Alternatively, secure multi-party computation (MPC) protocols enable us to compute the public parameters and thereby replace the trusted third party. As long as a large enough subset of parties is honest, the secret trapdoor is not available to anyone. Over the years, efficient solutions for distributed parameter generation have emerged, e.g., for distributed RSA key generation [FLOP18, CCD+20, CHI+20], or distributed ECDSA key generation [LN18].

Based on the recent progress in efficient MPC protocols, we ask the following question: *what if the parties kept their shares of the secret trapdoor?* Are the algorithms of the public-key accumulators exploiting knowledge of the secret trapdoor faster if performed within an (maliciously-secure) MPC protocol than their variants relying only on the public parameters?

1.1 Our Techniques

We give a short overview of how our construction works which allows us to positively answer this question for accumulators in the discrete logarithm setting. Let us consider the accumulator based on the q-SDH assumption which is based on the fact that given powers $g^{s^i} \in \mathbb{G}$ for all i up to q where $s \in \mathbb{Z}_p$ is unknown, it is possible to evaluate polynomials $f \in \mathbb{Z}_p[X]$ up to degree q at s in the exponent, i.e., $g^{f(s)}$. This is done by taking the coefficients of the polynomial, i.e., $f = \sum_{i=0}^{q} a_i X^i$, and computing $g^{f(s)}$ as $\prod_{i=0}^{q} (g^{s^i})^{a_i}$. The accumulator is built by defining a polynomial with the elements as roots and evaluating this polynomial at s in the exponent. A witness is simply the corresponding factor canceled out,

i.e., $g^{f(s)(s-x)^{-1}}$. Verification of the witness is performed by checking whether the corresponding factor and the witness match $g^{f(s)}$ using a pairing equation.

If s is known, all computations are more efficient: $f(s)$ can be directly evaluated in $\mathbb{Z}_p$ and the generation of the accumulator only requires one exponentiation in $\mathbb{G}$. The same is true for the computation of the witness. For the latter, the asymptotic runtime is thereby reduced from $\mathcal{O}(|\mathcal{X}|)$ to $\mathcal{O}(1)$. This improvement comes at a cost: if s is known, witnesses for non-members can be produced.

On the other hand, if multiple parties first produce s in an additively secret-shared fashion, these parties can cooperate in a secret-sharing based MPC protocol. Thereby, all the computations can still benefit from the knowledge of s. Indeed, the parties would compute their share of $g^{f(s)}$ and $g^{f(s)(s-x)^{-1}}$ respectively and thanks to the partial knowledge of s could still perform all operations – except the final exponentiation – in $\mathbb{Z}_p$. Furthermore, all involved computations are generic enough to be instantiated with MPC protocols with different trust assumptions. These include the dishonest majority protocol SPDZ [DKL+13, DPSZ12] and honest majority threshold protocols based on Shamir secret sharing [Sha79]. While in SPDZ, an honest party can always detect malicious behavior, Shamir adds robustness against parties dropping out or failing to provide their shares.

1.2 Our Contribution

Starting from the very recent treatment of accumulators in the UC model [Can01] by Baldimtsi et al. [BCY20], we introduce the notion of *(threshold) secret-shared accumulators*. As the name suggests, it covers accumulators where the trapdoor is available in a (potentially full) threshold secret-shared fashion with multiple parties running the parameter generation as well as the algorithms that profit from the availability of the trapdoor. Since the MPC literature discusses security in the UC model, we also chose to do so for our accumulators.

Based on recent improvements on distributed key generation of discrete logarithms, we provide dynamic public-key accumulators without trusted setup. During the parameter generation, the involved parties keep their shares of the secret trapdoor. Consequently, we present MPC protocols secure in the semi-honest and the malicious security model, respectively, implementing the algorithms for accumulator generation, witness generation, and accumulator updates exploiting the shares of the secret trapdoor. Specifically, we give such protocols for q-SDH accumulators [Ngu05, DHS15], which can be build from dishonest-majority full-threshold protocols (e.g., SPDZ [DKL+13, DPSZ12]) and from honest-majority threshold MPC protocols (e.g., Shamir secret sharing [Sha79]). In particular, our protocol enables updates to the accumulator independent of the size of the accumulated set. For increased efficiency, we consider this accumulator in bilinear groups of Type-3. Due to their structure, the construction nicely generalizes to any number of parties.

We provide a proof-of-concept implementation of our protocols in two MPC frameworks, MP-SPDZ [Kel20] and FRESCO.[8] We evaluate the efficiency of our protocols and compare them to the performance of an implementation, having no access to the secret trapdoors as usual for the public-key accumulators. We evaluate our protocol in the LAN and WAN setting in the semi-honest and malicious security model for various choices of parties and accumulator sizes. For the latter, we choose sizes up to 2^{14}. Specifically, for the q-SDH accumulator, we observe the expected $\mathcal{O}(1)$ runtimes for witness creation and accumulator updates, which cannot be achieved without access to the trapdoor. Notably, for the tested numbers of up to 5 parties, the MPC-enabled accumulator creation algorithms are faster for 2^{10} elements in the LAN setting than its non-MPC counterpart (without access to the secret trapdoor). For 2^{14} elements the algorithms are also faster in the WAN setting.

Finally, we discuss how our proposed MPC-based accumulators might impact revocation in distributed credential systems such as Sovrin [KL16]. In this scenario, the trust in the nodes run by the Sovrin foundation members can further be reduced. In addition, this approach generalizes to any accumulator-based revocation scheme and can be combined with threshold key management systems. We also discuss applications to CT and its privacy-preserving extension [KOR19]. In particular, the size of the witnesses stored in certificates or sent as part of the TLS handshake is significantly reduced without running into performance issues.

1.3 Related Work

When cryptographic protocols are deployed that require the setup of public parameters by a trusted third party, issues similar to those mentioned for public-key accumulators may arise. As discussed before, especially cryptocurrencies had to come up with ways to circumvent this problem for accumulators but also the common reference string (CRS) of zero-knowledge SNARKs [CGGN17]. Here, trust in the CRS is of paramount importance on the verifier side to prevent malicious provers from cheating. But also provers need to trust the CRS as otherwise zero-knowledge might not hold. We note that there are alternative approaches, namely subversion-resilient zk-SNARKS [BFS16] to reduce the trust required in the CRS generator. However, subversion-resilient soundness and zero-knowledge at the same time has been shown to be impossible by Bellare et al. Abdolmaleki et al. [ABLZ17] provided a construction of zk-SNARKS, which was later improved by Fuchsbauer [Fuc18], achieving subversion zero-knowledge by adding a verification algorithm for the CRS. As a result, only the verifier needs to trust the correctness of the CRS. Groth et al. [GKM+18] recently introduced the notion of an updatable CRS where first generic compilers [ARS20] are available to lift any zk-SNARK to an updatable simulation sound extractable zk-SNARK. There the CRS can be updated and if the initial generation or one of the updates was done honestly, neither soundness nor zero-knowledge can be subverted. In

[8] https://github.com/aicis/fresco

the random oracle model (ROM), those considerations become less of a concern and the trust put into the CRS can be minimized, e.g., as done in the construction of STARKs [BBHR19].

Approaches that try to fix the issue directly in the formalization of accumulators and corresponding constructions have also been studied. For example, Lipmaa [Lip12] proposed a modified model tailored to the hidden order group setting. In this model, the parameter setup is split into two algorithms, Setup and Gen where the adversary can control the trapdoors output by Setup, but can neither influence nor access the randomness used by Gen. However, constructions in this model so far have been provided using assumptions based on modules over Euclidean rings, and are not applicable to the efficient standard constructions we are interested in. More recently, Boneh et al. [BBF19] revisited the RSA accumulator without trapdoor which allows the accumulator to be instantiated from unknown order groups without trusted setup such as class groups of quadratic imaginary orders [HM00] and hyperelliptic curves of genus 2 or 3 [DG20].

The area of secure multiparty computation has seen a lot of interest both in improving the MPC protocols itself to a wide range of practical applications. In particular, SPDZ [DPSZ12, DKL+13] has seen a lot of interest, improvements and extensions [KOS15, KOS16, CDE+18, OSV20]. This interest also led to multiple MPC frameworks, e.g., MP-SPDZ [Kel20], FRESCO and SCALE-MAMBA,[9] enabling easy prototyping for researchers as well as developers. For practical applications of MPC, one can observe first MPC-based systems turned into products such as Unbound's virtual hardware security model (HSM).[10] For such a virtual HSM, one essentially wants to provide distributed key generation [FLOP18] together with threshold signatures [DK01] allowing to replace a physical HSM. Similar techniques are also interesting for securing wallets for the use in cryptocurrencies, where especially protocols for ECDSA [GG18, LN18] are of importance to secure the secret key material. Similarly, such protocols are also of interest for securing the secret key material of internet infrastructure such as DNSSEC [DOK+20]. Additionally, addressing privacy concerns in machine learning algorithms has become increasingly popular recently, with MPC protocols being one of the building blocks to achieve private classification and private model training as in [WGC19] for example. Recent works [SA19] also started to generalize the algorithms that are used as parts of those protocols allowing group operations on elliptic curve groups with secret exponents or secret group elements.

2 Preliminaries

In this section, we introduce cryptographic primitives we use as building blocks. For notation and assumptions, we refer to Appendix A.

[9] https://homes.esat.kuleuven.be/~nsmart/SCALE/
[10] https://www.unboundtech.com/usecase/virtual-hsm/

2.1 UC security and ABB

In this paper, we mainly work in the UC model first introduced by Canetti [Can01]. The success of the UC model stems from its universal composition theorem, which, informally speaking, states that it is safe to use a secure protocol as a sub-protocol in a more complex one. This strong statement enables one to analyze and proof the security of involved protocols in a modular way, allowing us to build upon work that was already proven to be secure in the UC model. In preparation for the security analysis of our MPC accumulators, we recall the definition of the UC model [Can01].

Definition 1. *Let $EXEC_{\Pi,\mathcal{A},\mathcal{E}}$ respectively $EXEC_{\mathcal{F},\mathcal{S},\mathcal{E}}$ denote the random variables describing the output of environment $\mathcal{E}$ when interacting with an adversary $\mathcal{A}$ and parties performing protocol Π, respectively when interacting with a simulator $\mathcal{S}$ and an ideal functionality $\mathcal{F}$. Protocol Π UC emulates the ideal functionality $\mathcal{F}$ if for any adversary $\mathcal{A}$ there exists a simulator $\mathcal{S}$ such that, for any environment $\mathcal{E}$ the distribution of $EXEC_{\Pi,\mathcal{A},\mathcal{E}}$ and $EXEC_{\mathcal{F},\mathcal{S},\mathcal{E}}$ are indistinguishable.*

The importance of the UC model for secure multiparty computation stems from the arithmetic black box (ABB)as introduced by Damgård and Nielsen [DN03]. The ABB models a secure general-purpose computer in the UC model. It allows performing arithmetic operations on private inputs provided by the parties. The result of these operations is then revealed to all parties. Working with the ABB provides us with a tool of abstracting arithmetic operations, including addition and multiplication in fields.

2.2 SPDZ, Shamir, and Derived Protocols

Our protocols build upon SPDZ [DPSZ12, DKL+13] and Shamir secret sharing [Sha79], concrete implementations of the abstract ABB. SPDZ itself is based on an additive secret-sharing over a finite field $\mathbb{F}_p$ with information-theoretic MACs making the protocol statistically UC secure against an active adversary corrupting all but one player. On the other hand, Shamir secret sharing is a threshold sharing scheme where $k \leq n$ out of n parties are enough to evaluate the protocol correctly. Therefore, it is naturally robust against parties dropping out during the computation; however, it assumes an honest-majority amongst all parties for security. Shamir secret sharing can be made maliciously UC secure in the honest-majority setting using techniques from [CGH+18] or [LN17].

We will denote the ideal functionality of the online protocol of SPDZ and Shamir secret sharing by $\mathcal{F}_{\text{Abb}}$. For an easy use of these protocols later in our accumulators, we give a high-level description of the functionality together with an intuitive notation. We assume that the computations are performed by n (or k) parties and we denote by $\langle s \rangle \in \mathbb{F}_p$ a secret-shared value between the parties in a finite field with p elements, where p is prime. The ideal functionality $\mathcal{F}_{\text{Abb}}$ provides us with the following basis operations: Addition $\langle a + b \rangle \leftarrow \langle a \rangle +$

$\langle b \rangle$ (can be computed locally), multiplication $\langle ab \rangle \leftarrow \langle a \rangle \cdot \langle b \rangle$ (interactive 1-round protocol), sampling $\langle r \rangle \xleftarrow{R} \mathbb{F}_p$, and opening a share $\langle a \rangle$. For convenience, we assume that we have also access to the inverse function $\langle a^{-1} \rangle$. Computation of the inverse can be efficiently implemented using a standard form of masking as first done in [BB89]. Given an opening of $\langle z \rangle = \langle r \cdot a \rangle$, the inverse of $\langle a \rangle$ is then equal to $z^{-1}\langle r \rangle$. However, there is a small failure probability if either a or r is zero. In our case, the field size is large enough that the probability of a random element being zero is negligible.

There is one additional sub-protocol which we will often need. Recent work [SA19] introduced protocols – in particular based on SPDZ – for group operations of elliptic curve groups supporting secret exponents and secret group elements. The high-level idea is to use the original SPDZ in the exponent group and for the authentication of the shares of an elliptic curve point a similar protocol as in SPDZ. For this work, we only need the protocol for exponentiation of a public point with a secret exponent. Let $\mathbb{G}$ be a cyclic group of prime order p and $g \in \mathbb{G}$. Further, let $\langle a \rangle \in \mathbb{F}_p$ be a secret-shared exponent.

$\mathsf{Exp}_{\mathbb{G}}(\langle a \rangle, g)$: The parties locally compute $\langle g^a \rangle \leftarrow g^{\langle a \rangle}$.

Since the security proof of this sub-protocol in [SA19] does not use any exclusive property of an elliptic curve group, it applies to any cyclic group of prime order.

All protocols discussed so far are secure in the UC model, making them safe to use in our accumulators as sub-protocols. Therefore, we will refer to their ideal functionality as $\mathcal{F}_{\mathrm{ABB+}}$. As a result, our protocols become secure in the UC model as long as we do not reveal any intermediate values.

2.3 Accumulators

We rely on the formalization of accumulators by Derler et al. [DHS15]. We start with the definitions of static and dynamic accumulators.

Definition 2 (Static Accumulator). *A static accumulator is a tuple of PPT algorithms* $(\mathsf{Gen}, \mathsf{Eval}, \mathsf{WitCreate}, \mathsf{Verify})$ *which are defined as follows:*

$\mathsf{Gen}(1^\kappa, q)$: *This algorithm takes a security parameter* κ *and a parameter* q*. If* $q \neq \infty$*, then* q *is an upper bound on the number of elements to be accumulated. It returns a key pair* $(\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda)$*, where* $\mathsf{sk}_\Lambda = \emptyset$ *if no trapdoor exists. We assume that the accumulator public key* pk_Λ *implicitly defines the accumulation domain* D_Λ*.*

$\mathsf{Eval}((\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda), \mathcal{X})$: *This algorithm takes a key pair* $(\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda)$ *and a set* $\mathcal{X}$ *to be accumulated and returns an accumulator* $\Lambda_\mathcal{X}$ *together with some auxiliary information* aux*.*

$\mathsf{WitCreate}((\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda), \Lambda_\mathcal{X}, \mathsf{aux}, x_i)$: *This algorithm takes a key pair* $(\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda)$*, an accumulator* $\Lambda_\mathcal{X}$*, auxiliary information* aux *and a value* x_i*. It returns* $\perp$*, if* $x_i \notin \mathcal{X}$*, and a witness* wit_{x_i} *for* x_i *otherwise.*

$\mathsf{Verify}(\mathsf{pk}_\Lambda, \Lambda_\mathcal{X}, \mathsf{wit}_{x_i}, x_i)$: *This algorithm takes a public key* pk_Λ*, an accumulator* $\Lambda_\mathcal{X}$*, a witness* wit_{x_i} *and a value* x_i*. It returns* 1 *if* wit_{x_i} *is a witness for* $x_i \in \mathcal{X}$ *and* 0 *otherwise.*

Definition 3 (Dynamic Accumulator). *A dynamic accumulator is a static accumulator with PPT algorithms* $(\mathsf{Add}, \mathsf{Delete}, \mathsf{WitUpdate})$ *defined as follows:*

$\mathsf{Add}((\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda), \Lambda_\mathcal{X}, \mathsf{aux}, x)$: *This algorithm takes a key pair* $(\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda)$*, an accumulator* $\Lambda_\mathcal{X}$*, auxiliary information* aux*, as well as an element* x *to be added. If* $x \in \mathcal{X}$*, it returns* $\perp$*. Otherwise, it returns the updated accumulator* $\Lambda_{\mathcal{X}'}$ *with* $\mathcal{X}' \leftarrow \mathcal{X} \cup \{x\}$ *and updated auxiliary information* aux'*.*

$\mathsf{Delete}((\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda), \Lambda_\mathcal{X}, \mathsf{aux}, x)$: *This algorithm takes a key pair* $(\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda)$*, an accumulator* $\Lambda_\mathcal{X}$*, auxiliary information* aux*, as well as an element* x *to be added. If* $x \notin \mathcal{X}$*, it returns* $\perp$*. Otherwise, it returns the updated accumulator* $\Lambda_{\mathcal{X}'}$ *with* $\mathcal{X}' \leftarrow \mathcal{X} \setminus \{x\}$ *and updated auxiliary information* aux'*.*

$\mathsf{WitUpdate}((\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda), \mathsf{wit}_{x_i}, \mathsf{aux}, x)$: *This algorithm takes a key pair* $(\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda)$*, a witness* wit_{x_i} *to be updated, auxiliary information* aux *and an* x *which was added to/deleted from the accumulator, where* aux *indicates addition or deletion. It returns an updated witness* wit'_{x_i} *on success and* $\perp$ *otherwise.*

This formalization of accumulators gives access to a trapdoor if it exists and sk_Λ is set to $\emptyset$ if it is not available. We recall collision freeness:

Definition 4 (Collision Freeness). *A cryptographic accumulator is* collision-free*, if for all PPT adversaries* $\mathcal{A}$ *there is a negligible function* $\varepsilon(\cdot)$ *such that:*

$$\Pr\left[\begin{array}{l}(\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda) \leftarrow \mathsf{Gen}(1^\kappa, q), (\mathsf{wit}_{x_i}{}^\star, x_i{}^\star, \mathcal{X}^\star, r^\star) \leftarrow \mathcal{A}^\mathcal{O}(\mathsf{pk}_\Lambda) : \\ \mathsf{Verify}(\mathsf{pk}_\Lambda, \Lambda^\star, \mathsf{wit}_{x_i}{}^\star, x_i{}^\star) = 1 \wedge x_i{}^\star \notin \mathcal{X}^\star\end{array}\right] \leq \varepsilon(\kappa),$$

where $\Lambda^\star \leftarrow \mathsf{Eval}((\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda), \mathcal{X}^\star; r^\star)$ *and the adversary gets access to the oracles* $\mathcal{O} = \{\mathsf{Eval}((\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda), \cdot), \mathsf{WitCreate}((\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda), \cdot, \cdot, \cdot)\}$ *and, if the accumulator is dynamic, additionally to* $\{\mathsf{Add}((\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda), \cdot, \cdot, \cdot), \mathsf{Delete}((\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda), \cdot, \cdot, \cdot), \mathsf{WitUpdate}((\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda), \cdot, \cdot, \cdot)\}$.

2.4 Pairing-based Accumulator

We recall the q-SDH-based accumulator from [DHS15], which is based on the accumulator by Nguyen [Ngu05]. The idea here is to encode the accumulated elements in a polynomial. This polynomial is then evaluated for a fixed element and the result is randomized to obtain the accumulator. A witness consists of the evaluation of the same polynomial with the term corresponding to the respective element cancelled out. For verification, a pairing evaluation is used to check whether the polynomial encoded in the witness is a factor of the one encoded in the accumulator. As it is typically more efficient to work with bilinear groups of Type-3 [GPS08], we state the accumulator as depicted in Scheme 1 in this setting. Correctness is clear, except for the $\mathsf{WitUpdate}$ subroutine: To update witness wit_{x_i} of the element x_i after the element x was added to the accumulator $\Lambda_\mathcal{X}$ to create the new accumulator $\Lambda_{\mathcal{X}'} = \Lambda_\mathcal{X}^{(x+s)}$, one computes:

$$\begin{aligned}\Lambda_\mathcal{X} \cdot \mathsf{wit}_{x_i}^{(x-x_i)} &= \Lambda_\mathcal{X}^{(x_i+s)\cdot(x_i+s)^{-1}} \cdot \Lambda_\mathcal{X}^{(x-x_i)\cdot(x_i+s)^{-1}} \\ &= \Lambda_\mathcal{X}^{(x+s)\cdot(x_i+s)^{-1}} = \Lambda_{\mathcal{X}'}^{(x_i+s)^{-1}}\end{aligned}$$

$\underline{\mathsf{Gen}(1^\kappa, q)}$: Let $\mathsf{BG} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g_1, g_2) \leftarrow \mathsf{BGen}(\kappa)$. Choose $s \xleftarrow{R} \mathbb{Z}_p^*$ and return $\mathsf{sk}_\Lambda \leftarrow s$ and $\mathsf{pk}_\Lambda \leftarrow (\mathsf{BG}, (g_1^{s^i})_{i=1}^q, g_2^s)$.

$\underline{\mathsf{Eval}((\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda), \mathcal{X})}$: Parse $\mathcal{X} \subset \mathbb{Z}_p^*$. Choose $r \xleftarrow{R} \mathbb{Z}_p^*$. If $\mathsf{sk}_\Lambda \neq \emptyset$, compute $\Lambda_\mathcal{X} \leftarrow g_1^{r \prod_{x \in \mathcal{X}} (x+s)}$. Otherwise, expand the polynomial $\prod_{x \in \mathcal{X}} (x + X) = \sum_{i=0}^n a_i X^i$, and compute $\Lambda_\mathcal{X} \leftarrow ((\prod_{i=0}^n g_1^{s^i})^{a_i})^r$. Return $\Lambda_\mathcal{X}$ and $\mathsf{aux} \leftarrow (\mathsf{add} \leftarrow 0, r, \mathcal{X})$.

$\underline{\mathsf{WitCreate}((\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda), \Lambda_\mathcal{X}, \mathsf{aux}, x)}$: Parse aux as $(r, \mathcal{X})$. If $x \notin \mathcal{X}$, return $\perp$. If $\mathsf{sk}_\Lambda \neq \emptyset$, compute and return $\mathsf{wit}_x \leftarrow \Lambda_\mathcal{X}^{(x+s)^{-1}}$. Otherwise, run $(\mathsf{wit}_x, \ldots) \leftarrow \mathsf{Eval}((\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda), \mathcal{X} \setminus \{x\}; r)$, and return wit_x.

$\underline{\mathsf{Verify}(\mathsf{pk}_\Lambda, \Lambda_\mathcal{X}, \mathsf{wit}_x, x)}$: Return 1 if $e(\Lambda_\mathcal{X}, g_2) = e(\mathsf{wit}_x, g_2^x \cdot g_2^s)$, otherwise return 0.

$\underline{\mathsf{Add}((\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda), \Lambda_\mathcal{X}, \mathsf{aux}, x)}$: Parse aux as $(r, \mathcal{X})$. If $x \in \mathcal{X}$, return $\perp$. Set $\mathcal{X}' \leftarrow \mathcal{X} \cup \{x\}$. If $\mathsf{sk}_\Lambda \neq \emptyset$, compute and return $\Lambda_{\mathcal{X}'} \leftarrow \Lambda_\mathcal{X}^{x+s}$ and $\mathsf{aux}' \leftarrow (r, \mathcal{X}', \mathsf{add} \leftarrow 1, \Lambda_\mathcal{X}, \Lambda_{\mathcal{X}'})$. Otherwise, return $\mathsf{Eval}((\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda), \mathcal{X}'; r)$ with aux extended with $(\mathsf{add} \leftarrow 1, \Lambda_\mathcal{X}, \Lambda_{\mathcal{X}'})$.

$\underline{\mathsf{Delete}((\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda), \Lambda_\mathcal{X}, \mathsf{aux}, x)}$: Parse aux as $(r, \mathcal{X})$. If $x \notin \mathcal{X}$, return $\perp$. Set $\mathcal{X}' \leftarrow \mathcal{X} \setminus \{x\}$. If $\mathsf{sk}_\Lambda \neq \emptyset$, compute and return $\Lambda_{\mathcal{X}'} \leftarrow \Lambda_\mathcal{X}^{(x+s)^{-1}}$ and $\mathsf{aux}' \leftarrow (r, \mathcal{X}', \mathsf{add} \leftarrow -1, \Lambda_\mathcal{X}, \Lambda_{\mathcal{X}'})$. Otherwise, return $\mathsf{Eval}((\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda), \mathcal{X}'; r)$ with aux extended with $(\mathsf{add} \leftarrow 0, \Lambda_\mathcal{X}, \Lambda_{X'})$.

$\underline{\mathsf{WitUpdate}((\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda), \mathsf{wit}_{x_i}, \mathsf{aux}, x)}$: Parse aux as $(\perp, \perp, \mathsf{add}, \Lambda_\mathcal{X}, \Lambda_{\mathcal{X}'})$. If $\mathsf{add} = 0$, return $\perp$. Return $\Lambda_\mathcal{X} \cdot \mathsf{wit}_{x_i}^{x - x_i}$ if $\mathsf{add} = 1$. If instead $\mathsf{add} = -1$, return $(\Lambda_{\mathcal{X}'}^{-1} \cdot \mathsf{wit}_{x_i})^{(x-x_i)^{-1}}$. In the last two cases in addition return $\mathsf{aux} \leftarrow (\mathsf{add} \leftarrow 0)$.

Scheme 1: q-SDH-based accumulator in the Type-3 setting.

which results in the desired updated witness. Similar, if the element x gets removed instead, one computes the following to get the desired witness:

$$\begin{aligned}(\Lambda_{\mathcal{X}'}^{-1} \cdot \mathsf{wit}_{x_i})^{(x-x_i)^{-1}} &= \Lambda_{\mathcal{X}'}^{-(x_i+s)\cdot(x_i+s)^{-1}\cdot(x-x_i)^{-1}} \cdot \Lambda_{\mathcal{X}'}^{(x+s)\cdot(x_i+s)^{-1}\cdot(x-x_i)^{-1}} \\ &= \Lambda_{\mathcal{X}'}^{(x_i+s)^{-1}\cdot(x-x_i)^{-1}\cdot(x-x_i+s-s)} = \Lambda_{\mathcal{X}'}^{(x_i+s)^{-1}}\end{aligned}$$

The proof of collision freeness follows from the q-SDH assumption. For completeness, we still restate the theorem from [DHS15] adopted to the Type-3 setting.

Theorem 1. *If the q-SDH assumption holds, then Scheme 1 is collision-free.*

Proof. Assume that $\mathcal{A}$ is an adversary against the collision freeness of the accumulator. We show that this adversary can be transformed into an efficient adversary $\mathcal{B}$ against the q-SDH assumption. We perform a proof by reduction in the following way:

– When $\mathcal{B}$ is started on a q-SDH instance $\left(\mathsf{BG}, \left(g_1^{s^i}\right)_{i \in [q]}, g_2^s\right)$, set $\mathsf{pk}_\Lambda \leftarrow \left(\mathsf{BG}, \left(g_1^{s^i}\right)_{i \in [q]}, g_2^s\right)$ and start $\mathcal{A}$ on pk_Λ. The oracles for $\mathcal{A}$ are simulated by forwarding the inputs directly to the corresponding algorithms with $(\emptyset, \mathsf{pk}_\Lambda)$ as argument for the keys.

$\mathcal{O}^{\mathsf{Eval}}(\mathcal{X})$: Return $\mathsf{Eval}((\emptyset, \mathsf{pk}_\Lambda), \mathcal{X})$.
$\mathcal{O}^{\mathsf{WitCreate}}(\Lambda_\mathcal{X}, \mathsf{aux}, x)$: Return $\mathsf{WitCreate}((\emptyset, \mathsf{pk}_\Lambda), \mathcal{X})$.
$\mathcal{O}^{\mathsf{Add}}(\Lambda_\mathcal{X}, \mathsf{aux}, x)$: Return $\mathsf{Add}((\emptyset, \mathsf{pk}_\Lambda), \Lambda_\mathcal{X}, \mathsf{aux}, x)$.
$\mathcal{O}^{\mathsf{Delete}}(\Lambda_\mathcal{X}, \mathsf{aux}, x)$: Return $\mathsf{Delete}((\emptyset, \mathsf{pk}_\Lambda), \Lambda_\mathcal{X}, \mathsf{aux}, x)$.
$\mathcal{O}^{\mathsf{WitUpdate}}(\mathsf{wit}_{x'}, \mathsf{aux}, x)$: Return $\mathsf{WitUpdate}((\emptyset, \mathsf{pk}_\Lambda), \mathsf{wit}_{x'}, \mathsf{aux}, x)$.

– At some point $\mathcal{A}$ outputs a set $\mathcal{X}^\star$, an element $x^\star \notin \mathcal{X}^\star$, a witness $\mathsf{wit}^\star_{x^\star}$ for $x^\star$, and the randomizer $r^\star$, such that for $\Lambda_{\mathcal{X}^\star}, \mathsf{aux} \leftarrow \mathsf{Eval}((\emptyset, \mathsf{pk}_\Lambda), \mathcal{X}^\star)$, the verification relation $e(\Lambda_{\mathcal{X}^\star}, g_2) = e(\mathsf{wit}^\star_{x^\star}, g_2^{x^\star} \cdot g_2^s)$ holds. Now, compute $h(X) = \prod_{x \in \mathcal{X}^\star}(x + X)$ and $\phi(X)$ such that $h(X) = \phi(X)(x^\star + X) + d$, which exists since $x^\star \notin \mathcal{X}^\star$. Then compute $g_1^{r^\star \phi(s)}$ by expanding the polynomial $\phi(X)$ and the $g_1^{s^i}$ stored in pk_Λ. Then, $\mathcal{B}$ outputs

$$\left(\mathsf{wit}^\star_{x^\star} \cdot \left(g_1^{r^\star \phi(s)}\right)^{-1}\right)^{\frac{1}{r^\star d}} = \left(g_1^{\frac{r^\star h(s)}{x^\star + s}} \cdot g_1^{\frac{-r^\star (h(s) - d)}{x^\star + s}}\right)^{\frac{1}{r^\star d}}$$
$$= \left(g_1^{\frac{r^\star d}{x^\star + s}}\right)^{\frac{1}{r^\star d}} = g_1^{\frac{1}{x^\star + s}}$$

and $x^\star$ as solution to the q-SDH problem instance.

Hence, $\mathcal{B}$ succeeds with the same probability as $\mathcal{A}$. □

Remark 1. Note that for support of arbitrary accumulation domains, the accumulator requires a suitable hash function mapping to $\mathbb{Z}_p^*$. For the MPC-based accumulators that we will define later, it is clear that the hash function can be evaluated in public. For simplicity, we omit the hash function in our discussion.

2.5 UC Secure Accumulators

Only recently, Baldimtsi et al. [BCY20] formalized the security of accumulators in the UC framework. Interestingly, they showed, that any correct and collision-free standard accumulator is automatically UC secure. We, however, want to note, that their definitions of accumulators are slightly different then the framework by Derler et al. (which we are using). Hence, we adapt the ideal functionality $\mathcal{F}_{\text{Acc}}$ from [BCY20] to match our setting: First our ideal functionality $\mathcal{F}_{\text{Acc}}$ consists of two more sub-functionalities. This is due to a separation of the algorithms responsible for the evaluation, addition, and deletion. Secondly, our $\mathcal{F}_{\text{Acc}}$ is simplified to our purpose, whereas $\mathcal{F}_{\text{Acc}}$ from Baldimtsi et al. is in their words "an entire menu of functionalities covering all different types of accumulators". Thirdly, we added identity checks to sub-functionalities (where necessary) to be consistent with the given definitions of accumulators.

The resulting ideal functionality is depicted in Functionality 1 in Appendix C. Note that the ideal functionality has up to three parties. First, the party which holds the set $\mathcal{X}$ is the accumulator manager $\mathcal{AM}$, responsible for the algorithms $\mathsf{Gen}, \mathsf{Eval}, \mathsf{WitCreate}, \mathsf{Add}$ and Delete. The second party $\mathcal{H}$ owns a witness and is interested in keeping it updated and for this reason, performs the algorithm

WitUpdate. The last party $\mathcal{V}$ can be seen as an external party. $\mathcal{V}$ is only able to use Verify to check the membership of an element in the accumulated set.

In the following theorem we adapt the proof from [BCY20] to our setting:

Theorem 2. *If* $\Pi_{Acc} = (\mathsf{Gen}, \mathsf{Eval}, \mathsf{WitCreate}, \mathsf{Verify}, \mathsf{Add}, \mathsf{Delete}, \mathsf{WitUpdate})$ *is a correct and collision-free dynamic accumulator with deterministic* Verify*, then* Π_{Acc} *UC emulates* $\mathcal{F}_{Acc}$*.*

Proof. We will proceed by contraposition. Assume to the contrary that Π_{Acc} does not UC emulate $\mathcal{F}_{\text{Acc}}$, i.e., there exists an environment $\mathcal{E}$, for all simulators $\mathcal{S}$ such that $\mathcal{E}$ can distinguish between the distributions of the random variables $EXEC_{\mathcal{F},\mathcal{S},\mathcal{E}}$ and $EXEC_{\Pi,\mathcal{A},\mathcal{E}}$ with non-negligible probability. Since the last statement holds for all simulators, we can choose one. We want a simulator $\mathcal{S}$ that interacts with the ideal functionality $\mathcal{F}_{\text{Acc}}$ in a way such that their distribution can not be distinguished by any environment from the real world, except when it violates either correctness or collision-freeness. Such a simulator would be a contradiction to our assumption and thereby prove the theorem.

Consider a simulator $\mathcal{S}$ that uses the standard corruption model from [Can01]. Further, $\mathcal{S}$ interacts with the environment $\mathcal{E}$ by forwarding any input to the real adversary $\mathcal{A}$ and conversely forwarding any output from $\mathcal{A}$ directly to $\mathcal{E}$. When $\mathcal{S}$ receives the request (GEN,*sid*) from $\mathcal{F}_{\text{Acc}}$, it replies with the actual accumulator algorithms. By construction of $\mathcal{S}$, the only differences to the real world that are visible for the environment $\mathcal{E}$ are the following instances where $\mathcal{F}_{\text{Acc}}$ returns $\perp$: (i) WitCreate: 4., (ii) Verify: 1.b, (iii) Add: 6. and (iv) WitUpdate : 3. The occurrence of one of the above cases would immediately imply a violation of the classical definition. More concretely, if Verify 1.b would return $\perp$, then the collision-freeness would be violated. In the other instances, correctness would not be given any more. □

As a direct consequence of Theorems 1 and 2, the accumulator from Scheme 1 is also secure in the UC model of [BCY20] since it is correct and collision-free:

Corollary 1. *Scheme 1 emulates* $\mathcal{F}_{Acc}$ *in the UC model.*

3 Multi-Party Public-Key Accumulators

With the building blocks in place, we are now able to go into the details of our construction. We first present the formal notion of (threshold) secret-shared accumulators, their ideal functionality, and then present our constructions.

For the syntax of the MPC-based accumulator, which we dub *(threshold) secret-shared accumulator*, we use the bracket notation $\langle s \rangle$ from Section 2.2 to denote a secret shared value. If we want to explicitly highlight the different shares, we write $\langle s \rangle = (s_1, \ldots, s_n)$, where the share s_i belongs to a party P_i. We base the definition on the framework of Derler et al. [DHS15], where our algorithms behave in the same way, but instead of taking an optional secret trapdoor, the algorithms are given shares of the secret as input. Consequently, Gen outputs shares of the secret trapdoor instead of the secret key. The static version of the accumulator is defined as follows:

Definition 5 (Static (Threshold) Secret-Shared Accumulator). *Let us assume that we have a (threshold) secret sharing-scheme. A static (threshold) secret-shared accumulator for $n \in \mathbb{N}$ parties $P_1, \ldots, P_n$ is a tuple of PPT algorithms* $(\mathsf{Gen}, \mathsf{Eval}, \mathsf{WitCreate}, \mathsf{Verify})$ *which are defined as follows:*

$\mathsf{Gen}(1^\kappa, q)$: *This algorithm takes a security parameter κ and a parameter q. If $q \neq \infty$, then q is an upper bound on the number of elements to be accumulated. It returns a key pair $(\mathsf{sk}_\Lambda^i, \mathsf{pk}_\Lambda)$ to each party P_i such that $\mathsf{sk}_\Lambda = \mathsf{Open}(\mathsf{sk}_\Lambda^1, \ldots, \mathsf{sk}_\Lambda^n)$, denoted by $\langle \mathsf{sk}_\Lambda \rangle$. We assume that the accumulator public key pk_Λ implicitly defines the accumulation domain D_Λ.*

$\mathsf{Eval}((\langle \mathsf{sk}_\Lambda \rangle, \mathsf{pk}_\Lambda), \mathcal{X})$: *This algorithm takes a secret-shared private key $\langle \mathsf{sk}_\Lambda \rangle$ a public key pk_Λ and a set $\mathcal{X}$ to be accumulated and returns an accumulator $\Lambda_\mathcal{X}$ together with some auxiliary information* aux *to every party P_i.*

$\mathsf{WitCreate}((\langle \mathsf{sk}_\Lambda \rangle, \mathsf{pk}_\Lambda), \Lambda_\mathcal{X}, \mathsf{aux}, x)$: *This algorithm takes a secret-shared private key $\langle \mathsf{sk}_\Lambda \rangle$ a public key pk_Λ, an accumulator $\Lambda_\mathcal{X}$, auxiliary information* aux *and a value x. It returns $\perp$, if $x \notin \mathcal{X}$, and a witness wit_x for x otherwise to every party P_i.*

$\mathsf{Verify}(\mathsf{pk}_\Lambda, \Lambda_\mathcal{X}, \mathsf{wit}_x, x)$: *This algorithm takes a public key pk_Λ, an accumulator $\Lambda_\mathcal{X}$, a witness wit_x and a value x. It returns 1 if wit_x is a witness for $x \in \mathcal{X}$ and 0 otherwise.*

In analogy to the non-interactive case, dynamic accumulators provide additional algorithms to add elements to the accumulator and remove elements from it, respectively, and update already existing witnesses accordingly. These algorithms are defined as follows:

Definition 6 (Dynamic (Threshold) Secret-Shared Accumulator). *A dynamic (threshold) secret-shared accumulator is a static (threshold) secret-shared accumulator with an additional tuple of PPT algorithms* $(\mathsf{Add}, \mathsf{Delete}, \mathsf{WitUpdate})$ *which are defined as follows:*

$\mathsf{Add}((\langle \mathsf{sk}_\Lambda \rangle, \mathsf{pk}_\Lambda), \Lambda_\mathcal{X}, \mathsf{aux}, x)$: *This algorithm takes a secret-shared private key $\langle \mathsf{sk}_\Lambda \rangle$ a public key pk_Λ, an accumulator $\Lambda_\mathcal{X}$, auxiliary information* aux*, as well as an element x to be added. If $x \in \mathcal{X}$, it returns $\perp$ to every party P_i. Otherwise, it returns the updated accumulator $\Lambda_{\mathcal{X}'}$ with $\mathcal{X}' \leftarrow \mathcal{X} \cup \{x\}$ and updated auxiliary information aux' to every party P_i.*

$\mathsf{Delete}((\langle \mathsf{sk}_\Lambda \rangle, \mathsf{pk}_\Lambda), \Lambda_\mathcal{X}, \mathsf{aux}, x)$: *This algorithm takes a secret-shared private key $\langle \mathsf{sk}_\Lambda \rangle$ a public key pk_Λ, an accumulator $\Lambda_\mathcal{X}$, auxiliary information* aux*, as well as an element x to be added. If $x \notin \mathcal{X}$, it returns $\perp$ to every party P_i. Otherwise, it returns the updated accumulator $\Lambda_{\mathcal{X}'}$ with $\mathcal{X}' \leftarrow \mathcal{X} \setminus \{x\}$ and updated auxiliary information aux' to every party P_i.*

$\mathsf{WitUpdate}((\langle \mathsf{sk}_\Lambda \rangle, \mathsf{pk}_\Lambda), \mathsf{wit}_{x_i}, \mathsf{aux}, x)$: *This algorithm takes a secret-shared private key $\langle \mathsf{sk}_\Lambda \rangle$ a public key pk_Λ, a witness wit_{x_i} to be updated, auxiliary information* aux *and an element x which was added to/deleted from the accumulator, where* aux *indicates addition or deletion. It returns an updated witness wit'_{x_i} on success and $\perp$ otherwise to every party P_i.*

Correctness and collision-freeness naturally translate from the non-interactive accumulators to the (threshold) secret-shared ones. The work of Baldimtsi et al. also introduced the property creation-correctness. Informally speaking, creation-correctness allows the generation of witnesses during addition. In the above definitions, we see that adding an element to the accumulator and creating a witness are two separate algorithms. Therefore, the notion of creation-correctness does not immediately apply to our accumulators.

For our case, the ideal functionality for (threshold) secret-shared accumulators, dubbed $\mathcal{F}_{\text{MPC-Acc}}$ is more interesting. $\mathcal{F}_{\text{MPC-Acc}}$ is very similar to $\mathcal{F}_{\text{Acc}}$ and is depicted in Functionality 2 in Appendix C. The only difference in describing the ideal functionality for accumulators in the MPC setting arises from the fact that we now have not only one accumulator manager but n, denoted by $\mathcal{AM}_1, \ldots, \mathcal{AM}_n$. More concretely, whenever a sub-functionality of $\mathcal{F}_{\text{MPC-Acc}}$ – that makes use of the secret key – gets a request from a manager identity $\mathcal{AM}_i$, it now also gets a participation message from the other managers identities A_j for $j \neq i$. Furthermore, the accumulator managers take the role of the witness holder. The party $\mathcal{V}$, however, stays unchanged.

3.1 Dynamic (Threshold) Secret-Shared Accumulator from the q-SDH Assumption

For the generation of public parameters Gen, we can rely on already established methods to produce ECDSA key pairs and exponentiations with secret exponents, respectively. These methods can directly be applied to the accumulators. Taking the q-SDH accumulator as an example, the first step is to sample the secret scalar $s \in \mathbb{Z}_p$. Intuitively, each party samples its own share s_i and the secret trapdoor s would then be $s = \mathsf{Open}(s_1, \ldots, s_n)$. The next step, the calculation of the basis elements g^{s^j} for $j = 1, \ldots, q$, is optional, but can be performed to provide public parameters, that are useful even to parties without knowledge of s. All of these elements can be computed using $\mathsf{Exp}_{\mathbb{G}}$ and the secret-shared s, respectively its powers. For the accumulator evaluation, Eval, the parties first sample their shares of r. Then, they jointly compute shares of $r \cdot f(s)$ using their shares of r and s. The so-obtained exponent and $\mathsf{Exp}_{\mathbb{G}}$ produce the final result.

For witness creation, $\mathsf{WitCreate}$, it gets more interesting. Of course, one could simply run Eval again with one element removed from the set. In this case, we can do better, though. The difference between the accumulator and a witness is that in the latter, one factor of the polynomial is canceled. Since s is available, it is thus possible to cancel this factor without recomputing the polynomial from the start. Indeed, to compute the witness for an element x, we can compute $(s+x)^{-1}$ and then apply that inverse using $\mathsf{Exp}_{\mathbb{G}}$ to the accumulator to get the witness. Note though, that before the parties perform this step, they need to check if x is actually contained in $\mathcal{X}$. Otherwise, they would produce a membership witness for a non-member. In that case, the verification would check whether $f(s)(s+x)^{-1}(s+x)$ matches $f(s)$, which of course also holds even if $s+x$ is not a factor of $f(s)$. In contrast, when performing Eval with only the publicly available information, this issue does not occur since there the witness will not verify. Add

and Delete can be implemented in a similar manner. When adding an element to the accumulator, the polynomial is extended by one factor. Removal of an element requires that one factor is canceled. Both operations can be performed by first computing the factor using the shares of s and then running $\mathsf{Exp}_{\mathbb{G}}$.

Now, we present the MPC version of the q-SDH accumulator in Scheme 2 following the intuition outlined above. Note, that the algorithm for WitUpdate is unlikely to be faster than its non-MPC version from Scheme 1. Indeed, the non-MPC version requires only exponentiations in $\mathbb{G}_1$ and a multiplication without the knowledge of the secret trapdoor. We provide the version using the trapdoor for completeness but will use the non-MPC version of the algorithm in practical implementations. Note further that we let Gen choose the bilinear group BG, but this group can already be fixed a priori.

$\underline{\mathsf{Gen}(1^\kappa, q)}$: $\mathsf{BG} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g_1, g_2) \leftarrow \mathsf{BGen}(\kappa)$. Compute $\langle \mathsf{sk}_\Lambda \rangle \leftarrow \mathsf{sRand}(\mathbb{Z}_p^*)$. Compute $h \leftarrow \mathsf{Open}(g_2^{\langle \mathsf{sk}_\Lambda \rangle})$. Return $\mathsf{pk}_\Lambda \leftarrow (\mathsf{BG}, h)$.

$\underline{\mathsf{Eval}((\langle \mathsf{sk}_\Lambda \rangle, \mathsf{pk}_\Lambda), \mathcal{X})}$: Parse pk_Λ as (BG, h) and $\mathcal{X}$ as subset of $\mathbb{Z}_p^*$. Choose $\langle r \rangle \leftarrow \mathsf{sRand}(\mathbb{Z}_p^*)$. Compute $\langle q \rangle \leftarrow \prod_{x \in \mathcal{X}} (x + \langle \mathsf{sk}_\Lambda \rangle) \in \mathbb{Z}_p^*$ and $\langle t \rangle \leftarrow \langle q \rangle \cdot \langle r \rangle$. The algorithm returns $\Lambda_\mathcal{X} \leftarrow \mathsf{Open}(g_1^{\langle t \rangle})$ and $\mathsf{aux} \leftarrow (\mathsf{add} \leftarrow 0, \mathcal{X})$.

$\underline{\mathsf{WitCreate}((\langle \mathsf{sk}_\Lambda \rangle, \mathsf{pk}_\Lambda), \Lambda_\mathcal{X}, \mathsf{aux}, x)}$: Returns $\perp$ if $x \notin \mathcal{X}$. Otherwise, $\langle z \rangle \leftarrow \langle (x + \langle \mathsf{sk}_\Lambda \rangle)^{-1} \rangle$. Return $\mathsf{wit}_x \leftarrow \mathsf{Open}(\Lambda_\mathcal{X}^{\langle z \rangle})$.

$\underline{\mathsf{Verify}(\mathsf{pk}_\Lambda, \Lambda_\mathcal{X}, \mathsf{wit}_x, x)}$: Parse pk_Λ as (BG, h). If $e(\Lambda_\mathcal{X}, g_2) = e(\mathsf{wit}_x, g_2^x \cdot h)$ holds, return 1, otherwise return 0.

$\underline{\mathsf{Add}((\langle \mathsf{sk}_\Lambda \rangle, \mathsf{pk}_\Lambda), \Lambda_\mathcal{X}, \mathsf{aux}, x)}$: Returns $\perp$ if $x \in \mathcal{X}$. Otherwise set $\mathcal{X}' \leftarrow \mathcal{X} \cup \{x\}$. Return $\Lambda_{\mathcal{X}'} \leftarrow \Lambda_\mathcal{X}^x \cdot \mathsf{Open}(\Lambda_\mathcal{X}^{\langle \mathsf{sk}_\Lambda \rangle})$ and $\mathsf{aux} \leftarrow (\mathsf{add} \leftarrow 1, \mathcal{X}')$.

$\underline{\mathsf{Delete}((\langle \mathsf{sk}_\Lambda \rangle, \mathsf{pk}_\Lambda), \Lambda_\mathcal{X}, \mathsf{aux}, x)}$: If $x \notin \mathcal{X}$, return $\perp$. Otherwise set $\mathcal{X}' \leftarrow \mathcal{X} \setminus \{x\}$, and compute $\langle y \rangle \leftarrow \langle (x + \langle \mathsf{sk}_\Lambda \rangle)^{-1} \rangle$. Return $\Lambda_{\mathcal{X}'} \leftarrow \mathsf{Open}(\Lambda_\mathcal{X}^{\langle y \rangle})$ and $\mathsf{aux} \leftarrow (\mathsf{add} \leftarrow -1, \mathcal{X}')$.

$\underline{\mathsf{WitUpdate}((\langle \mathsf{sk}_\Lambda \rangle, \mathsf{pk}_\Lambda), \mathsf{wit}_{x_i}, \mathsf{aux}, x)}$: Parse aux as $(\mathsf{add}, \mathcal{X})$. Return $\perp$ if $\mathsf{add} = 0$ or $x_i \notin \mathcal{X}$. In case $\mathsf{add} = 1$, return $\mathsf{wit}_{x_i} \leftarrow \mathsf{wit}_{x_i}^x \cdot \mathsf{Open}(\mathsf{wit}_{x_i}^{\langle \mathsf{sk}_\Lambda \rangle})$ and $\mathsf{aux} \leftarrow (\mathsf{add} \leftarrow 0, \mathcal{X})$. If instead $\mathsf{add} = -1$, it compute $\langle y \rangle \leftarrow \langle (x + \langle \mathsf{sk}_\Lambda \rangle)^{-1} \rangle$. Return $\mathsf{wit}_{x_i} \leftarrow \mathsf{Open}(\mathsf{wit}_{x_i}^{\langle y \rangle})$ and $\mathsf{aux} \leftarrow (\mathsf{add} \leftarrow 0, \mathcal{X})$.

Scheme 2: MPC-q-SDH: Dynamic (threshold) secret-shared accumulator from q-SDH for $n \geq 2$ parties.

Theorem 3. *Scheme 2 UC emulates $\mathcal{F}_{Acc\text{-}MPC}$ in the $\mathcal{F}_{ABB+}$-hybrid model.*

Proof. At this point, we make use of the UC model. Informally speaking, accumulators are UC secure, and SPDZ, Shamir secret sharing, and the derived operations UC emulate $\mathcal{F}_{\text{ABB+}}$. Therefore, according to the universal composition theorem, the use of these MPC protocols in the accumulator Scheme 2 can be done without losing UC security. For a better understanding, we begin by showing the desired accumulator properties for Scheme 2.

The proof of the correctness follows directly from the correctness proof from Scheme 1 for the case where the secret key is known. Collision-freeness is also derived from the non-interactive q-SDH accumulator. (It is true that now each party has a share of the trapdoor, but without the other shares no party can create a valid witness.) Since Verify is obviously deterministic, Scheme 2 fulfills all necessary assumption of Theorem 2. After applying Theorem 2, we get a simulator $\mathcal{S}_{Acc}$ interacting with the ideal functionality $\mathcal{F}_{\text{Acc}}$. Since we now also have to simulate the non-interactive sub-protocols, we have to extend $\mathcal{S}_{Acc}$. We construct $\mathcal{S}_{\text{Acc-MPC}}$ by building upon $\mathcal{S}_{\text{Acc}}$ and in addition internally simulate $\mathcal{F}_{\text{ABB+}}$. As described in Section 2.2, the MPC protocols used in the above algorithms are all secure in the UC model. Since we do not open any secret-shared values besides uniformly random elements and the output or values that can be immediately derived from the output, the algorithms are secure due to the universal composition theorem. □

Remark 2. In Gen of Scheme 2 we explicitly do not compute $h_i \leftarrow g_1^{s^i}$. Hence, using Eval without access to s is not possible. But, on the positive side, the public parameters are significantly smaller and so is the runtime of the Gen algorithm. If, however, these values are needed to support a non-secret-shared Eval, one can modify Gen to also compute the following values: $\langle t_1 \rangle \leftarrow \langle s \rangle$, $\langle t_i \rangle \leftarrow \langle t_{i-1} \rangle \cdot \langle s \rangle$, and $h_i \leftarrow \mathsf{Open}(g_1^{\langle t_i \rangle})$ for $i = 1, \ldots, t$. This opens up the possibilities to trade an efficient Eval computation with an inefficient Gen step, which could be precomputed before the actual accumulator is created. Updates to this accumulator then still profit from the efficiency of the secret shared trapdoor. Additionally, q gives an upper bound on the size of the accumulated sets, and thus needs to be considered in the selection of the curves even though the powers of g_1 are not placed in the public key.

3.2 SPDZ vs. Shamir Secret Sharing

In this section, we want to compare two MPC protocols on which our MPC-q-SDH Accumulator can be based on, namely SPDZ and Shamir secret sharing. Both protocols allow us to keep shares of the secret trapdoor and improve performance compared to the keyless q-SDH Accumulator. However, in relying on these protocols for security, the trust assumptions of the MPC-q-SDH Accumulator also have to include the underlying protocols' trust-assumptions.

SPDZ is a full-threshold dishonest-majority protocol that protects against $n-1$ corrupted parties. Therefore, an honest party will always detect malicious behavior. However, full-threshold schemes are not robust; if one party fails to supply its shares, the computation always fails.

On the contrary, Shamir secret sharing is an honest-majority threshold protocol. It is more robust than SPDZ since it allows $k \leq \frac{n-1}{2}$ corrupted parties while still being capable of providing correct results. This also means, if some parties ($\mathrm{k} \leq \frac{n-1}{2}$) fail to provide their shares, the other parties can still compute the correct results without them. Thus, no accumulator manager on its own is a single point of failure. However, if more than k parties are corrupted, the

adversaries can reconstruct the secret trapdoor and, therefore, compromise the security of our MPC-q-SDH Accumulator.

4 Implementation and Performance Evaluation

We implemented the proposed dynamic (threshold) secret-shared accumulator from q-SDH and evaluated it against small to large sets.[11] Our primary implementations are based on SPDZ with OT-based preprocessing and Shamir secret sharing in the MP-SPDZ [Kel20][12] framework. However, to demonstrate the usability of our accumulator, we additionally build an implementation in the malicious security setting with dishonest-majority based on the FRESCO framework. We discuss the benchmarks for the MP-SPDZ implementation in this section. For a discussion of the FRESCO benchmarks we refer the reader to Appendix E.

Remark 3. We want to note, that in our benchmarks we test the performance of the MPC variant of WitUpdate from Scheme 2, even though in practice the non-MPC variant from scheme Scheme 1 should be used.

MP-SPDZ implements the SPDZ protocol with various extensions [DPSZ12, KOS15,KOS16,CDE+18], as well as semi-honest and malicious variants of Shamir secret sharing [CDM00,CGH+18,LN17]. For pairing and elliptic curve group operations, we rely on relic[13] and integrate $\mathsf{Exp}_{\mathbb{G}}$, Output-$\mathbb{G}$, and the corresponding operations to update the MAC described in [SA19] into MP-SPDZ. We use the pairing friendly BLS12-381 curve [BLS02], which provides around 120 bit of security following recent estimates [BD19]. We want to note, that our implementation can easily be adapted to support other pairing libraries, as well. For completeness, we also implemented the q-SDH accumulator from Scheme 1 and a Merkle-tree accumulator (cf. Appendix B) using SHA-256. This enables us to compare the performance in cases where the secret trapdoors are available in the MPC case and when they are not. In Table 1, we present the numbers for various sizes of accumulated sets.

The evaluation of the MPC protocols was performed on a cluster with a Xeon E5-4669v4 CPU, where each party was assigned only 1 core. The hosts were connected via a 1 Gbit/s LAN network, and an average round-trip time of < 1 ms. For the WAN setting, a network with a round-trip time of 100 ms and a bandwidth of 100 Mbit/s was simulated. We provide benchmarks for both preprocessing and online phases of the MPC protocols, where the cost of the preprocessing phase is determined by the number of shared multiplications, whereas the performance of the online phase is proportional to the multiplicative depth of the circuit and the number of openings.

[11] The source code is available at https://github.com/IAIK/MPC-Accumulator.
[12] https://github.com/data61/MP-SPDZ
[13] https://github.com/relic-toolkit/relic

Table 1. Performance of the accumulator algorithms without access to the secret trapdoors. Time in milliseconds averaged over 100 executions.

Accu.	$\lvert\mathcal{X}\rvert$	Gen	Eval	WitCreate	Add	WitUpdate	Delete	WitUpdate
Scheme 1	2^{10}	649	1 117	1 116	1 116	0.6	1 120	0.7
	2^{14}	9 062	116 031	115 870	115 575	0.6	116 154	0.7
Merkle-Tree	2^{10}	–	1.12	0.05[a]	1.12	0.05[a]	1.12	0.05[a]
	2^{14}	–	15.53	0.83[a]	15.53	0.83[a]	15.53	0.83[a]

[a] Assuming that the full Merkle-tree is known as auxiliary data. If not, the tree has to be rebuilt, which adds the Eval-time.

Table 2. Number of Beaver triples, shared random values, and opening rounds required by MPC-q-SDH.

	Gen	Eval	WitCreate	Add	WitUpdate	Delete	WitUpdate
Beaver triples	0	$\lvert\mathcal{X}\rvert$	1	0	0	1	1
Random values	1	1	1	0	0	1	1
Opening rounds	1	$\lceil\log_2(\lvert\mathcal{X}\rvert+1)\rceil+1$	3	1	1	3	3

[a] Note, semi-honest Shamir secret sharing does not require Beaver triples.

4.1 Evaluation of MPC-q-SDH

In the offline phase of the implemented MPC protocols, the required Beaver triples [Bea91] for shared multiplication and the pre-shared random values are generated. A shared inverse operation requires one multiplication and one shared random value. In Table 2, we list the number of triples required for each operation for the MPC-q-SDH accumulator. Except for Eval they require a constant number of multiplications and inverse operations and, therefore, a constant number of Beaver triples and shared random elements. In Eval, the number of required Beaver triples is determined by $\lvert\mathcal{X}\rvert$. Furthermore, Table 2 lists the number of opening rounds (including openings in multiplications, excluding MAC-checks) of the online phase of the MPC-q-SDH accumulator allowing one to calculate the number of communication rounds for different sharing schemes.

As discussed in Remark 2, Gen is not producing the public parameters h_i. If Eval without MPC is desired, the time and communication of Eval for the respective set sizes should be added to the time and communication of Gen to obtain an estimate of its performance.

Dishonest-Majority based on SPDZ. Table 3 compares the offline performance of the MPC-q-SDH accumulator based on SPDZ in different settings. We give both timings for the accumulation of $\lvert\mathcal{X}\rvert$ elements in Eval and the necessary pre-computation for a single inversion, which is used in several other operations (e.g., WitCreate). Additionally we also give the time for pre-computing a single random element, which is required to generate the authenticated share of the secret-key in Eval. Further note that batching the generation of many triples

Table 3. Offline phase performance of different steps of the MPC-q-SDH accumulator with access to the secret trapdoor based on MP-SPDZ. Time in milliseconds.

			LAN setting				WAN setting			
	Operation	$\lvert\mathcal{X}\rvert$	$n =$ 2	3	4	5	2	3	4	5
	BaseOTs	2^{10}, 2^{14}	0.03	0.08	0.14	0.23	0.14	0.31	0.56	0.84
Semi-Honest	Inverse	2^{10}, 2^{14}	0.78	1.72	3.06	4.03	209.9	227.5	322.8	331.0
Semi-Honest	Gen	2^{10}, 2^{14}	0.44	1.21	1.76	3.01	207.7	223.6	325.9	332.0
Semi-Honest	Eval	2^{10}	189	397	706	959	4 695	8 215	13 680	25 725
Semi-Honest	Eval	2^{14}	4000	8 308	14 380	17 928	55 542	109 720	214 585	356 330
Malicious	Inverse	2^{10}, 2^{14}	4.34	7.93	11.5	15.3	840.5	1 262	1 538	1 914
Malicious	Gen	2^{10}, 2^{14}	2.56	4.23	6.80	9.32	841.3	1 235	1 540	1 856
Malicious	Eval	2^{10}	1 601	2 849	4 345	6 227	25 737	45 254	87 328	141 181
Malicious	Eval	2^{14}	31 099	62 978	89 132	145 574	412 747	682 033	1 364 660	2 236 860

together like for the Eval phase is more efficient in practice than producing a single triple and as these triples are not dependent on the input, all parties can continuously generate triples in the background for later use in the online phase.

In Table 4, we present the online performance of our MPC-q-SDH accumulator based on SPDZ for different set sizes, parties, security settings, and network settings. It can clearly be seen, that – except for the Eval operation – the runtime of each operation is independent of the set size. In other words, after an initial accumulation of a given set, every other operation has constant time. In comparison, the runtime of the non-MPC accumulators without access to the secret trapdoor, as depicted in Table 1, depends on the size of the accumulated set. Our MPC-accumulator outperforms the non-MPC q-SDH accumulators the larger the accumulated set gets. In the LAN setting MPC-q-SDH's Eval is faster than the non-MPC version for all benchmarked players, even in the WAN settings it outperforms the non-MPC version in the two player case. For 2^{14} elements, it is even faster for all benchmarked players in all settings, including the WAN setting. In any case, the witnesses have constant size contrary to the $\log_2(|\mathcal{X}|)$ sized witnesses of the Merkle-tree accumulator.

The numbers for the evaluation of the online phase in the WAN setting are also presented in Table 4. The overhead that can be observed compared to the LAN setting is influenced by the communication cost. Since our implementation implements all multiplications in Eval in a depth-optimized tree-like fashion, the overhead from switching to a WAN setting is not too severe.

On the first look, one can observe an irregularity in our benchmarks. More specifically, notice that for four or more parties, the maliciously secure evaluation of the Eval online phase is consistently faster than the semi-honest evaluation of the same phase. However, this is a direct consequence of a difference in how MP-SPDZ handles the communication in those security models, where communication is handled in a non-synchronized send-to-all approach in the malicious setting and a synchronized broadcast approach in the semi-honest setting. The

Table 4. Online phase performance of the MPC-q-SDH accumulator with access to the secret trapdoor based on SPDZ implemented in MP-SPDZ, for both the LAN and WAN settings with n parties. Time in milliseconds averaged over 50 executions.

		Semi-Honest								Malicious							
		LAN setting				WAN setting				LAN setting				WAN setting			
Operation	$\lvert\mathcal{X}\rvert$	$n=$ 2	3	4	5	2	3	4	5	2	3	4	5	2	3	4	5
Gen	2^{10}	4	4	7	19	53	110	170	219	11	13	25	37	169	278	395	505
	2^{14}	4	4	9	20	56	111	172	220	11	13	28	48	179	280	396	506
Eval	2^{10}	3	13	58	231	635	1 277	1 916	2 558	10	17	50	131	966	1 327	1 669	1 995
	2^{14}	26	47	117	315	949	1 948	3 166	4 571	89	94	174	225	1 297	1 979	2 830	3 872
WitCreate	2^{10}	2	2	32	39	168	320	482	645	5	10	35	75	372	606	823	1 050
	2^{14}	2	2	28	51	168	320	473	638	5	6	28	80	365	606	835	1 052
Add	2^{10}	2	2	8	17	47	107	166	213	5	5	17	31	170	273	388	499
	2^{14}	2	2	5	14	50	108	170	214	5	5	17	42	173	271	383	491
$\mathsf{WitUpdate}_{\mathrm{Add}}$	2^{10}	2	2	5	30	60	108	154	214	5	7	12	34	159	276	390	495
	2^{14}	2	2	3	20	60	107	152	217	5	6	10	54	154	275	390	500
Delete	2^{10}	2	2	21	58	156	319	488	639	5	10	47	78	379	598	818	1 034
	2^{14}	2	2	23	55	158	318	489	642	5	6	38	87	385	603	822	1 033
$\mathsf{WitUpdate}_{\mathrm{Delete}}$	2^{10}	2	2	52	47	165	320	475	643	5	10	26	100	374	604	828	1 044
	2^{14}	2	4	43	57	162	323	475	639	5	10	35	74	365	599	827	1 048

Table 5. Communication cost (in kB per party) of the MPC-q-SDH accumulator with access to the secret trapdoor based on SPDZ implemented in MP-SPDZ.

		Semi-Honest		Malicious	
Operations	$\lvert\mathcal{X}\rvert$	Offline[a]	Online	Offline[a]	Online
Gen	2^{10}, 2^{14}	20	0.10	86	0.24
Eval	2^{10}	12 571	66	79 549	66
	2^{14}	200 823	1 049	1 271 484	1 049
WitCreate, Delete, $\mathsf{WitUpdate}_{\mathrm{Delete}}$	2^{10}, 2^{14}	33	0.15	164	0.37
Add, $\mathsf{WitUpdate}_{\mathrm{Add}}$	2^{10}, 2^{14}	4	0.05	4	0.14

[a] Includes BaseOTs for a new connection

synchronization in the latter case scales worse for more parties and, therefore, introduces some additional delays.

Finally, Table 5 depicts the size of the communication between the parties for both offline and online phases. The communication of Eval has to account for a number of multiplications dependent on $\mathcal{X}$ and therefore scales linearly with its size. As we already observed for the runtime of MPC-q-SDH, also the communication of WitCreate, Add, Delete and WitUpdate is independent of the size of the accumulated set, and additionally less than 200 kB for all algorithms. Combined with the analysis of the runtime, we conclude that the performance of the operations that might be performed multiple times per accumulator is very efficient in both runtime and communication. When compared to the performance of the non-MPC accumulators in Table 1, we see that the performance of

Table 6. Offline phase performance of different steps of the MPC-q-SDH accumulator with access to the secret trapdoor in the semi-honest (SH) and malicious threshold setting implemented in MP-SPDZ. Time in milliseconds.

	Operation	$\lvert\mathcal{X}\rvert$	LAN setting $n=$ 3	4	5	WAN setting 3	4	5
SH	Inverse	2^{10}, 2^{14}	6	9	14	473	585	998
Malicious	Inverse	2^{10}, 2^{14}	7	9	17	1 036	1 231	2 136
	Gen	2^{10}, 2^{14}	6	11	17	1 008	1 256	2 233
	Eval	2^{10}	20	29	48	1 232	2 089	2 629
		2^{14}	218	245	510	3 431	8 130	8 519

operations that benefit from access to the secret trapdoor are multiple orders of magnitude faster in the MPC accumulators and, in the LAN setting, even come close to the performance of the standard Merkle-tree accumulator, for both the semi-honest and malicious variant.

Honest-Majority Threshold Sharing based on Shamir Secret Sharing. In this section, we discuss the benchmarks of our implementation based on Shamir secret sharing. MP-SPDZ implements semi-honest Shamir secret sharing based on [CDM00] and a maliciously secure variant following [LN17][14]. In Table 6, we present the offline phase runtime, in Table 7 we show the runtime of the online phase, and in Table 8 we depict the size of the communication between the parties for the 3-party case.

The most expensive part of the SPDZ offline phase is creating the Beaver triples required for the Eval operation. As Table 6 shows, this step is several orders of magnitudes cheaper in the Shamir-based implementation. This is especially true in the semi-honest setting, in which no Beaver triples are required in the Shamir-based implementation. The offline runtime of the other operations is similar to the SPDZ-based implementations.

The Shamir-based implementation's online runtime is slightly cheaper than the runtime of the SPDZ-based implementation, except for the Eval operation. However, the difference in runtime of the Eval operation is also not significant, especially when considering the trade for the much cheaper offline phase.

Similar behavior can be seen for the communication cost, as depicted in Table 8. Offline communication is several orders of magnitude smaller in the Shamir-based implementation than in SPDZ, while online communication is similar to the SPDZ based version. Only the Eval operation requires about twice as much online communication in the Shamir-based implementation. To summarize, our honest-majority threshold implementation based on Shamir secret sharing

[14] A newer version of MP-SPDZ now implements maliciously secure Shamir secret sharing following [CGH+18].

Table 7. Online phase performance of the MPC-q-SDH accumulator with access to the secret trapdoor in the threshold setting implemented in MP-SPDZ, for both the LAN and WAN settings with n parties. Time in milliseconds averaged over 50 executions.

		Semi-Honest						Malicious					
		LAN setting			WAN setting			LAN setting			WAN setting		
Operation	$\lvert\mathcal{X}\rvert$	$n=$ 3	4	5	3	4	5	3	4	5	3	4	5
Gen	2^{10}	5	5	7	109	111	118	7	7	14	112	119	228
	2^{14}	5	5	7	110	111	120	7	7	14	113	120	230
Eval	2^{10}	5	6	9	1 278	1 314	2 474	9	9	16	1 285	1 422	2 578
	2^{14}	33	40	77	1 788	2 776	3 831	80	84	161	2 018	3 938	4 636
WitCreate	2^{10}	2	2	3	317	319	440	3	3	5	324	336	648
	2^{14}	2	2	3	318	321	443	3	3	5	323	332	642
Add	2^{10}	2	2	3	109	108	114	3	3	5	109	111	215
	2^{14}	2	2	3	107	108	113	3	3	5	110	112	220
$\mathsf{WitUpdate}_{\mathrm{Add}}$	2^{10}	2	2	3	107	107	112	3	3	5	107	112	217
	2^{14}	2	2	3	107	108	114	3	3	5	108	114	220
Delete	2^{10}	2	2	3	320	321	438	3	3	5	320	332	642
	2^{14}	2	2	3	317	321	439	3	3	5	321	331	642
$\mathsf{WitUpdate}_{\mathrm{Delete}}$	2^{10}	2	2	3	320	321	441	3	3	5	321	333	647
	2^{14}	2	2	3	316	320	441	3	3	5	320	332	645

provides much better offline phase performance, with similar online performance compared to our dishonest majority full-threshold implementation.

4.2 Further Improvement

The maliciously secure MPC protocols we use in this work delay the MAC check to the output phase after executing the Open subroutine. This means, it is possible for intermediate results to be wrong due to tampering of an attacker; however, since honest parties only reveal randomized values during the openings in a multiplication, no information about secret values can be gained by attackers.

Similar to threshold signature schemes [GG18,DKLS19,DOK+20], the protocols can be optimized by skipping the MAC checks at the end of WitCreate, Add, Delete, and WitUpdate and use the Verify step of the accumulator to check for correctness instead. The only feasible attack on this optimization is to produce invalid accumulators/witnesses without leaking information on the secret trapdoor; however, false output values can be detected during verification. Therefore, we can execute the semi-honest online phase and call Verify at the end, while still protecting against malicious parties. This trades the extra round of communication in the MAC check for an evaluation of a bilinear pairing (≈ 10 ms on our benchmark platform) which results in a further speedup, especially in the WAN-setting.

Table 8. Communication cost (in kB per party) of the MPC-q-SDH accumulator in the 3-party threshold setting implemented in MP-SPDZ.

Operation	$\lvert\mathcal{X}\rvert$	Semi-Honest Offline	Semi-Honest Online	Malicious Offline	Malicious Online
Gen	2^{10}, 2^{14}	0.26	0.20	0.65	0.20
Eval	2^{10}	0.26	66	459	131
	2^{14}	0.26	1 049	7 340	2 097
WitCreate, Delete, $\mathsf{WitUpdate}_{\mathrm{Delete}}$	2^{10}, 2^{14}	0.26	0.23	1.1	0.3
Add, $\mathsf{WitUpdate}_{\mathrm{Add}}$	2^{10}, 2^{14}	0	0.11	0	0.11

5 Applications

5.1 Credential Revocation in Distributed Credential Systems

As first application of MPC-based accumulators, we focus on distributed credential systems [GGM14], and in particular, on the implementation in Sovrin [KL16]. In general, anonymous credentials provide a mechanism for making identity assertions while maintaining privacy, yet, in classical, non-distributed systems require a trusted credential issuer. This central issuer, however, is both a single point of failure and a target for compromise and can make it challenging to deploy such a system. In a distributed credential system, on the other hand, this trusted credential issuer is eliminated, e.g., by using distributed ledgers.

We shortly recall how Sovrin implements revocation. When issuing a credential, every user gets a unique revocation identifier i_R. All valid revocation IDs are accumulated using a q-SDH accumulator which is published. Additionally, the users obtains a witness certifying membership of its i_R in the accumulator. Whenever a user shows their credential, they have to prove that they know this witness for their i_R with respect to the published accumulator. When a new user joins, the accumulator has to be updated. Consequently, all the witnesses have to be updated as well, as otherwise they would no longer be able to provide a valid proof. Similar, in the case that a user is revoked and thus removed from the accumulator, all other users have to update their witnesses accordingly. Also, the verifiers always have to check for updated accumulators.

Now, recall that the q-SDH accumulator supports all required operations without needing access to the trapdoor. Hence, all operations can be performed and, especially, the users can update their witnesses on their own if the corresponding i_Rs are published on the ledger. While functionality-wise all operations are supported, performance-wise a large number of users becomes an issue. With potentially millions to billions of users, adding and deleting members from the accumulator becomes increasingly expensive (cf. Table 1). Hence, at a certain size, having access to the trapdoor would be beneficial. But, on the other side, generating membership witnesses for non-members would then become possible.

The latter is also an issue during the setup of the system. Trusting one third party to generate the public parameters of the accumulator might be undesired in a distributed system as in this case. The special structure of the Sovrin ecosystem with their semi-trusted foundation members, however, naturally fits to our multi-party accumulator. First, the foundation members can setup the public parameters in a distributed manner. Secondly, as all of them have shares of the trapdoor, they can also run the updates of the accumulator using the MPC-q-SDH-accumulator. Additionally, using a threshold secret sharing scheme can add robustness against foundation members failing to provide their shares for computations. The change to this accumulator is completely transparent to the clients and verifiers and no changes are required there. Furthermore, the Verify step of the MPC-q-SDH-accumulator is equal to the Verify operation of the non-MPC q-SDH-accumulator. Therefore, the same efficient zero-knowledge proofs [ACN13] can be used to prove knowledge of a witness without revealing it. These proofs are significantly more efficient then proving witnesses of a Merkle-Tree-accumulator, even when SNARK-friendly hash functions (e.g., Poseidon [GKK+19]) are used.

5.2 Privacy-Preserving Certificate-Transparency Logs

We finally look at the application of accumulators in the CT ecosystem. Certificate Authorities request the inclusion of certificates in the log whenever they sign a new certificate. Once the certificate was included in the log, auditors can check the consistency of this log. Additionally, TLS clients also verify whether all certificates that they obtain were actually logged, thereby ensuring that log servers do not hand out promises of certificate inclusion without following through. Technically, the CT log is realized as a Merkle-tree accumulator containing all certificates. As certificates need to be added continuously, it is made dynamic by simply recalculating the root hash and all the proofs. Functionality wise, dynamic accumulators would perfectly fit this use-case. However, their real-world performance without secret trapdoors is not good enough – recalculating hash trees is just more efficient. Knowledge of the secret trapdoors would however be catastrophic for this application, as the guarantees of the whole system break down: log servers could produce witnesses for any certificate they get queried on, even if it was never submitted to the log servers for inclusion.

In the CT ecosystem, the clients need to contact the log servers for the inclusion proof, and therefore verifying certificates has negative privacy implications, as this query reveals the browsing behavior of the client to the log server. Based on previous work by Lueks and Goldberg [LG15], Kales et al. [KOR19] proposed to rethink retrieval of the inclusion proofs by employing multi-server private information retrieval (PIR) to query the proofs. To further improve performance, the accumulator is split into sub-accumulators based on, e.g., time periods. All sub-accumulators are then accumulated in a top-level accumulator. Consequently, the witnesses with respect to the sub-accumulator stay constant and can be embedded in the server's certificate and only the membership-proofs of the sub-accumulators need to be updated when new certificates are added to the log. Only these top-level proofs have to be queried using PIR, thus greatly

improving the overall performance, as smaller databases are more efficient to query.

However, one drawback of this solution is the increase in certificate size if one were to include this static membership witness for the sub-accumulator in the certificate itself. Kales et al. [KOR19] propose to build sub-accumulators per hour, which would result in sub-accumulators that hold about 2^{16} certificates. A Merkle-tree membership proof for these sub-accumulators is 512 bytes in size when using SHA-256. In contrast, a membership proof for the q-SDH accumulator is only 48 bytes in size (with the curve used in our implementation). A typical DER-encoded X509 certificate using RSA-2048 as used in TLS is about 1-2 KB in size, meaning inclusion of the Merkle-tree sub-accumulator membership proof would increase the certificate size by $25 - 50\%$, whereas the q-SDH sub-accumulator membership proof only increases the size by $2.5 - 5\%$.

We can now leverage the fact that their solution already requires two non-colluding servers for the multi-server PIR. These servers hold copies of the Merkle-tree accumulator and answer private membership queries for the top-level accumulator. Switching the used accumulators to our MPC-q-SDH accumulator would give the benefit of small, constant size membership proofs, while still being performant enough to accumulate and produce witnesses for all elements of a sub-accumulator in one hour.

6 Conclusion

In this work, we introduced dynamic (threshold) secret-shared accumulators which remove the need of a trusted third party for public-key accumulators. By replacing the trusted party with a distributed setup algorithm, we achieved even more: since shares of the secret trapdoor are now available, the otherwise expensive algorithms can also be implemented as MPC protocol making use of the trapdoor. Thereby we obtained – especially in the bilinear groups setting – an efficient accumulator even for large sizes of accumulated sets.

Since our constructions are generic in a sense, improvements in the underlying MPC protocols and their implementations will directly translate to our accumulators.

Acknowledgments. This work was supported by EU's Horizon 2020 project under grant agreement n°825225 (Safe-DEED) and n°871473 (KRAKEN), and EU's Horizon 2020 ECSEL Joint Undertaking grant agreement n°783119 (SECREDAS), and by the "DDAI" COMET Module within the COMET – Competence Centers for Excellent Technologies Programme, funded by the Austrian Federal Ministry for Transport, Innovation and Technology (bmvit), the Austrian Federal Ministry for Digital and Economic Affairs (bmdw), the Austrian Research Promotion Agency (FFG), the province of Styria (SFG) and partners from industry and academia. The COMET Programme is managed by FFG.

References

ABLZ17. Behzad Abdolmaleki, Karim Baghery, Helger Lipmaa, and Michal Zajac. A subversion-resistant SNARK. In *ASIACRYPT (3)*, volume 10626 of *LNCS*, pages 3–33. Springer, 2017.

ACN13. Tolga Acar, Sherman S. M. Chow, and Lan Nguyen. Accumulators and u-prove revocation. In *Financial Cryptography*, volume 7859 of *LNCS*, pages 189–196. Springer, 2013.

ARS20. Behzad Abdolmaleki, Sebastian Ramacher, and Daniel Slamanig. Lift-and-shift: Obtaining simulation extractable subversion and updatable snarks generically. In *CCS*, pages 1987–2005. ACM, 2020.

ATSM09. Man Ho Au, Patrick P. Tsang, Willy Susilo, and Yi Mu. Dynamic universal accumulators for DDH groups and their application to attribute-based anonymous credential systems. In *CT-RSA*, volume 5473 of *LNCS*, pages 295–308. Springer, 2009.

BB89. Judit Bar-Ilan and Donald Beaver. Non-cryptographic fault-tolerant computing in constant number of rounds of interaction. In *PODC*, pages 201–209. ACM, 1989.

BB08. Dan Boneh and Xavier Boyen. Short signatures without random oracles and the SDH assumption in bilinear groups. *J. Cryptology*, 21(2):149–177, 2008.

BBF19. Dan Boneh, Benedikt Bünz, and Ben Fisch. Batching techniques for accumulators with applications to iops and stateless blockchains. In *CRYPTO (1)*, volume 11692 of *LNCS*, pages 561–586. Springer, 2019.

BBHR19. Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable zero knowledge with no trusted setup. In *CRYPTO (3)*, volume 11694 of *LNCS*, pages 701–732. Springer, 2019.

BCY20. Foteini Baldimtsi, Ran Canetti, and Sophia Yakoubov. Universally composable accumulators. In *CT-RSA*, volume 12006 of *LNCS*, pages 638–666. Springer, 2020.

BD19. Razvan Barbulescu and Sylvain Duquesne. Updating key size estimations for pairings. *J. Cryptology*, 32(4):1298–1336, 2019.

BdM93. Josh Cohen Benaloh and Michael de Mare. One-way accumulators: A decentralized alternative to digital sinatures (extended abstract). In *EUROCRYPT*, volume 765 of *LNCS*, pages 274–285. Springer, 1993.

Bea91. Donald Beaver. Efficient multiparty protocols using circuit randomization. In *CRYPTO*, volume 576 of *LNCS*, pages 420–432. Springer, 1991.

BFS16. Mihir Bellare, Georg Fuchsbauer, and Alessandra Scafuro. Nizks with an untrusted CRS: security in the face of parameter subversion. In *ASIACRYPT (2)*, volume 10032 of *LNCS*, pages 777–804, 2016.

BL11. Ernie Brickell and Jiangtao Li. Enhanced privacy ID from bilinear pairing for hardware authentication and attestation. *IJIPSI*, 1(1):3–33, 2011.

BL12. Ernie Brickell and Jiangtao Li. Enhanced privacy ID: A direct anonymous attestation scheme with enhanced revocation capabilities. *IEEE Trans. Dependable Sec. Comput.*, 9(3):345–360, 2012.

BLS02. Paulo S. L. M. Barreto, Ben Lynn, and Michael Scott. Constructing elliptic curves with prescribed embedding degrees. In *SCN*, volume 2576 of *LNCS*, pages 257–267. Springer, 2002.

BN05. Paulo S. L. M. Barreto and Michael Naehrig. Pairing-friendly elliptic curves of prime order. In *SAC*, volume 3897 of *LNCS*, pages 319–331. Springer, 2005.

BP97. Niko Baric and Birgit Pfitzmann. Collision-free accumulators and fail-stop signature schemes without trees. In *EUROCRYPT*, volume 1233 of *LNCS*, pages 480–494. Springer, 1997.

BS01. Emmanuel Bresson and Jacques Stern. Efficient revocation in group signatures. In *PKC*, volume 1992 of *LNCS*, pages 190–206. Springer, 2001.

Can01. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145. IEEE, 2001.

CCD+20. Megan Chen, Ran Cohen, Jack Doerner, Yashvanth Kondi, Eysa Lee, Schuyler Rosefield, and Abhi Shelat. Multiparty generation of an RSA modulus. In *CRYPTO (3)*, volume 12172 of *LNCS*, pages 64–93. Springer, 2020.

CDE+18. Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. SPD$\mathbb{Z}_{2^k}$: Efficient MPC mod 2^{k} for dishonest majority. In *CRYPTO (2)*, volume 10992 of *LNCS*, pages 769–798. Springer, 2018.

CDM00. Ronald Cramer, Ivan Damgård, and Ueli M. Maurer. General secure multi-party computation from any linear secret-sharing scheme. In *EUROCRYPT*, volume 1807 of *LNCS*, pages 316–334. Springer, 2000.

CGGN17. Matteo Campanelli, Rosario Gennaro, Steven Goldfeder, and Luca Nizzardo. Zero-knowledge contingent payments revisited: Attacks and payments for services. In *ACM CCS*, pages 229–243. ACM, 2017.

CGH+18. Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. Fast large-scale honest-majority MPC for malicious adversaries. In *CRYPTO (3)*, volume 10993 of *Lecture Notes in Computer Science*, pages 34–64. Springer, 2018.

CHI+20. Megan Chen, Carmit Hazay, Yuval Ishai, Yuriy Kashnikov, Daniele Micciancio, Tarik Riviere, Abhi Shelat, Muthuramakrishnan Venkitasubramaniam, and Ruihan Wang. Diogenes: Lightweight scalable RSA modulus generation with a dishonest majority. *IACR Cryptol. ePrint Arch.*, 2020:374, 2020.

CL02. Jan Camenisch and Anna Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In *CRYPTO*, volume 2442 of *LNCS*, pages 61–76. Springer, 2002.

DG20. Samuel Dobson and Steven D. Galbraith. Trustless groups of unknown order with hyperelliptic curves. *IACR ePrint*, 2020:196, 2020.

DHS15. David Derler, Christian Hanser, and Daniel Slamanig. Revisiting cryptographic accumulators, additional properties and relations to other primitives. In *CT-RSA*, volume 9048 of *LNCS*, pages 127–144. Springer, 2015.

DK01. Ivan Damgård and Maciej Koprowski. Practical threshold RSA signatures without a trusted dealer. In *EUROCRYPT*, volume 2045 of *LNCS*, pages 152–165. Springer, 2001.

DKL+13. Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In *ESORICS*, volume 8134 of *LNCS*, pages 1–18. Springer, 2013.

DKLS19. Jack Doerner, Yashvanth Kondi, Eysa Lee, and Abhi Shelat. Threshold ECDSA from ECDSA assumptions: The multiparty case. In *IEEE S&P*, pages 1051–1066. IEEE, 2019.

DN03. Ivan Damgård and Jesper Buus Nielsen. Universally composable efficient multiparty computation from threshold homomorphic encryption. In *CRYPTO*, volume 2729 of *LNCS*, pages 247–264. Springer, 2003.

DOK+20. Anders P. K. Dalskov, Claudio Orlandi, Marcel Keller, Kris Shrishak, and Haya Shulman. Securing DNSSEC keys via threshold ECDSA from generic MPC. In *ESORICS (2)*, volume 12309 of *LNCS*, pages 654–673. Springer, 2020.

DPSZ12. Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO*, volume 7417 of *LNCS*, pages 643–662. Springer, 2012.

DRS18. David Derler, Sebastian Ramacher, and Daniel Slamanig. Post-quantum zero-knowledge proofs for accumulators with applications to ring signatures from symmetric-key primitives. In *PQCrypto*, volume 10786 of *LNCS*, pages 419–440. Springer, 2018.

FHM11. Chun-I Fan, Ruei-Hau Hsu, and Mark Manulis. Group signature with constant revocation costs for signers and verifiers. In *CANS*, volume 7092 of *LNCS*, pages 214–233. Springer, 2011.

FLOP18. Tore Kasper Frederiksen, Yehuda Lindell, Valery Osheter, and Benny Pinkas. Fast distributed RSA key generation for semi-honest and malicious adversaries. In *CRYPTO (2)*, volume 10992 of *LNCS*, pages 331–361. Springer, 2018.

Fuc18. Georg Fuchsbauer. Subversion-zero-knowledge snarks. In *PKC (1)*, volume 10769 of *LNCS*, pages 315–347. Springer, 2018.

GG18. Rosario Gennaro and Steven Goldfeder. Fast multiparty threshold ECDSA with fast trustless setup. In *ACM CCS*, pages 1179–1194. ACM, 2018.

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

Gil99. Niv Gilboa. Two party RSA key generation. In *CRYPTO*, volume 1666 of *LNCS*, pages 116–129. Springer, 1999.

GKK+19. Lorenzo Grassi, Daniel Kales, Dmitry Khovratovich, Arnab Roy, Christian Rechberger, and Markus Schofnegger. Starkad and poseidon: New hash functions for zero knowledge proof systems. *IACR ePrint*, 2019:458, 2019.

GKM+18. Jens Groth, Markulf Kohlweiss, Mary Maller, Sarah Meiklejohn, and Ian Miers. Updatable and universal common reference strings with applications to zk-snarks. In *CRYPTO (3)*, volume 10993 of *LNCS*, pages 698–728. Springer, 2018.

GPS08. Steven D. Galbraith, Kenneth G. Paterson, and Nigel P. Smart. Pairings for cryptographers. *Discret. Appl. Math.*, 156(16):3113–3121, 2008.

HM00. Safuat Hamdy and Bodo Möller. Security of cryptosystems based on class groups of imaginary quadratic orders. In *ASIACRYPT*, volume 1976 of *LNCS*, pages 234–247. Springer, 2000.

Kel20. Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. In *CCS*, pages 1575–1590. ACM, 2020.

KL16. Dmitry Khovratovich and Jason Law. Sovrin: digitial signatures in the blockchain area, 2016.

KOR19. Daniel Kales, Olamide Omolola, and Sebastian Ramacher. Revisiting user privacy for certificate transparency. In *EuroS&P*, pages 432–447. IEEE, 2019.

KOS15. Marcel Keller, Emmanuela Orsini, and Peter Scholl. Actively secure OT extension with optimal overhead. In *CRYPTO (1)*, volume 9215 of *LNCS*, pages 724–741. Springer, 2015.

KOS16. Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: faster malicious arithmetic secure computation with oblivious transfer. In *ACM CCS*, pages 830–842. ACM, 2016.

Lau14. Ben Laurie. Certificate transparency. *ACM Queue*, 12(8):10–19, 2014.

LG15. Wouter Lueks and Ian Goldberg. Sublinear scaling for multi-client private information retrieval. In *Financial Cryptography*, volume 8975 of *LNCS*, pages 168–186. Springer, 2015.

Lip12. Helger Lipmaa. Secure accumulators from euclidean rings without trusted setup. In *ACNS*, volume 7341 of *LNCS*, pages 224–240. Springer, 2012.

LLNW16. Benoît Libert, San Ling, Khoa Nguyen, and Huaxiong Wang. Zero-knowledge arguments for lattice-based accumulators: Logarithmic-size ring signatures and group signatures without trapdoors. In *EUROCRYPT (2)*, volume 9666 of *LNCS*, pages 1–31. Springer, 2016.

LN17. Yehuda Lindell and Ariel Nof. A framework for constructing fast MPC over arithmetic circuits with malicious adversaries and an honest-majority. In *CCS*, pages 259–276. ACM, 2017.

LN18. Yehuda Lindell and Ariel Nof. Fast secure multiparty ECDSA with practical distributed key generation and applications to cryptocurrency custody. In *ACM CCS*, pages 1837–1854. ACM, 2018.

Mer89. Ralph C. Merkle. A certified digital signature. In *CRYPTO*, volume 435 of *LNCS*, pages 218–238. Springer, 1989.

MGGR13. Ian Miers, Christina Garman, Matthew Green, and Aviel D. Rubin. Zerocoin: Anonymous distributed e-cash from bitcoin. In *IEEE S&P*, pages 397–411. IEEE, 2013.

NFHF09. Toru Nakanishi, Hiroki Fujii, Yuta Hira, and Nobuo Funabiki. Revocable group signature schemes with constant costs for signing and verifying. In *PKC*, volume 5443 of *LNCS*, pages 463–480. Springer, 2009.

Ngu05. Lan Nguyen. Accumulators from bilinear pairings and applications. In *CT-RSA*, volume 3376 of *LNCS*, pages 275–292. Springer, 2005.

NKHF05. Toru Nakanishi, Fumiaki Kubooka, Naoto Hamada, and Nobuo Funabiki. Group signature schemes with membership revocation for large groups. In *ACISP*, volume 3574 of *LNCS*, pages 443–454. Springer, 2005.

NS04. Toru Nakanishi and Yuji Sugiyama. A group signature scheme with efficient membership revocation for reasonable groups. In *ACISP*, volume 3108 of *LNCS*, pages 336–347. Springer, 2004.

OSV20. Emmanuela Orsini, Nigel P. Smart, and Frederik Vercauteren. Overdrive2k: Efficient secure MPC over $\mathbb{Z}_{2^k}$ from somewhat homomorphic encryption. In *CT-RSA*, volume 12006 of *LNCS*, pages 254–283. Springer, 2020.

PTT08. Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. Authenticated hash tables. In *ACM CCS*, pages 437–448. ACM, 2008.

SA19. Nigel P. Smart and Younes Talibi Alaoui. Distributing any elliptic curve based protocol. In *IMACC*, volume 11929 of *LNCS*, pages 342–366. Springer, 2019.

San99. Tomas Sander. Efficient accumulators without trapdoor extended abstracts. In *ICICS*, volume 1726 of *LNCS*, pages 252–262. Springer, 1999.

Sha79. Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

SSU16. Daniel Slamanig, Raphael Spreitzer, and Thomas Unterluggauer. Linking-based revocation for group signatures: A pragmatic approach for efficient revocation checks. In *Mycrypt*, volume 10311 of *LNCS*, pages 364–388. Springer, 2016.

Ver16. Eric R. Verheul. Practical backward unlinkable revocation in fido, german e-id, idemix and u-prove. *IACR ePrint*, 2016:217, 2016.

WGC19. Sameer Wagh, Divya Gupta, and Nishanth Chandran. Securenn: 3-party secure computation for neural network training. *PoPETs*, 2019(3):26–49, 2019.

A Notation and Assumptions

Notation. Notation-wise, let $[n] := \{1, \ldots, n\}$ for $n \in \mathbb{N}$. For an algorithm $\mathcal{A}$, we write $\mathcal{A}(\cdots; r)$ to make the random coins explicit. We say that an algorithm is efficient if it runs in probabilistic polynomial time (PPT).

Assumptions. Let $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ be cyclic groups of prime order p. A *pairing* $e\colon \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is a map that is *bilinear* (i.e., for all $g_1, g_2' \in \mathbb{G}_1$ and $g_2, g_2' \in \mathbb{G}_2$, we have $e(g_1 \cdot g_1', g_2) = e(g_1, g_2) \cdot e(g_1', g_2)$ and $e(g_1, g_2 \cdot g_2') = e(g_1, g_2) \cdot e(g_1, g_2')$, *non-degenerate* (i.e., for generators $g_1 \in \mathbb{G}_1, g_2 \in \mathbb{G}_2$, we have that $e(g_1, g_2) \in G_T$ is a generator), and *efficiently computable*. Let BGen be a PPT algorithm that, on input of a security parameter κ, outputs $\mathsf{BG} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g_1, g_2) \leftarrow \mathsf{BGen}(\kappa)$ for generators g_1 and g_2 of $\mathbb{G}_1$ and $\mathbb{G}_2$, respectively, and $\Theta(\kappa)$-bit prime p. If $\mathbb{G}_1 = \mathbb{G}_2$ the pairing is of Type-1 and if $\mathbb{G}_1 \neq \mathbb{G}_2$ and no non-trivial efficiently computable homomorphism $\mathbb{G}_2 \to \mathbb{G}_1$ exists, then it is of Type-3.

We recall the q-SDH assumption for Type-3 bilinear groups [BB08].

Definition 7 (q-SDH assumption). *For $q > 0$, we define the advantage of an adversary $\mathcal{A}$ as*

$$\mathbf{Adv}^{\mathsf{q\text{-}SDH}}_{\mathsf{BGen},\mathcal{A}}(\kappa) = \Pr\left[x \xleftarrow{R} \mathbb{Z}_p, (c, y) \leftarrow A\left(\mathsf{BG}, \left(g_1^{x^i}\right)_{i\in[q]}, g_2^x\right) : y = g_1^{(x+c)^{-1}}\right].$$

The q-SDH assumption holds if $\mathbf{Adv}^{\mathsf{q\text{-}SDH}}_{\mathsf{BGen},A}$ is a negligible function in the security parameter κ for all PPT adversaries $\mathcal{A}$.

In the Type-1 setting, the assumption can be simplified to only providing the powers of g_1 to the adversary, since $g_1 = g_2$.

B Merkle-tree Accumulator

In Scheme 3, we cast the Merkle-tree accumulator in the framework of [DHS15] as done in [DRS18, KOR19]. In practical instantiations, the requirement that Eval only works on sets of a size that is a power of 2 can be dropped. It is always possible to repeat the last element until the tree is of the correct size. Correctness can easily be verified. We restate the well-known fact that this accumulator is collision-free.

Lemma 1. *If $\{H_k\}_{k\in\mathsf{K}^\kappa}$ is a family of collision-resistant hash functions, the static accumulator in Scheme 3 is collision-free.*

$\underline{\mathsf{Gen}(1^\kappa, t)}$: Fix a family of hash functions $\{H_k\}_{k\in\mathsf{K}^\kappa}$ with $H_k\colon \{0,1\}^* \to \{0,1\}^\kappa\ \forall\ k \in \mathsf{K}^\kappa$. Choose $k \xleftarrow{R} \mathsf{K}^\kappa$ and return $(\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda) \leftarrow (\emptyset, H_k)$.

$\underline{\mathsf{Eval}((\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda), \mathcal{X})}$: Parse pk_Λ as H_k and $\mathcal{X}$ as $(x_0, \ldots, x_{n-1})$. If $\nexists\ k \in \mathbb{N}$ so that $n = 2^k$ return $\perp$. Otherwise, let $\ell_{u,v}$ refer to the u-th leaf (the leftmost leaf is indexed by 0) in the v-th layer (the root is indexed by 0) of a perfect binary tree. Return $\Lambda_\mathcal{X} \leftarrow \ell_{0,0}$ and $\mathsf{aux} \leftarrow ((\ell_{u,v})_{u\in[n/2^{k-v}]})_{v\in[k]}$, where

$$\ell_{u,v} \leftarrow \begin{cases} H_k(\ell_{2u,v+1}||\ell_{2u+1,v+1}) & \text{if } v < k, \text{ and} \\ H_k(x_i) & \text{if } v = k. \end{cases}$$

$\underline{\mathsf{WitCreate}((\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda), \Lambda_\mathcal{X}, \mathsf{aux}, x_i)}$: Parse aux as $((\ell_{u,v})_{u\in[n/2^{k-v}]})_{v\in[k]}$ and return wit_{x_i} where

$$\mathsf{wit}_{x_i} \leftarrow (\ell_{\lfloor i/2^v \rfloor + \eta, k-v})_{0\le v\le k}, \eta = \begin{cases} 1 & \text{if } \lfloor i/2^v \rfloor \pmod 2 = 0 \\ -1 & \text{otherwise.} \end{cases}$$

$\underline{\mathsf{Verify}(\mathsf{pk}_\Lambda, \Lambda_\mathcal{X}, \mathsf{wit}_{x_i}, x_i)}$: Parse pk_Λ as H_k, $\Lambda_\mathcal{X}$ as $\ell_{0,0}$, set $\ell_{i,k} \leftarrow H_k(x_i)$. Recursively check for all $0 < v < k$ whether the following holds and return 1 if so. Otherwise return 0.

$$\ell_{\lfloor i/2^{v+1} \rfloor, k-(v+1)} = \begin{cases} H_k(\ell_{\lfloor i/2^v \rfloor, k-v}||\ell_{\lfloor i/2^v \rfloor + 1, k-v}) & \text{if } 2|\lfloor i/2^v \rfloor \\ H_k(\ell_{\lfloor i/2^v \rfloor - 1, k-v}||\ell_{\lfloor i/2^v \rfloor, k-v}) & \text{otherwise.} \end{cases}$$

Scheme 3: Merkle-tree accumulator.

C Ideal Functionalities

In this section, we list the ideal functionalities used in the proofs in this paper. In Functionality 1 we present the ideal functionality $\mathcal{F}_{\text{Acc}}$, in Functionality 2 we present $\mathcal{F}_{\text{MPC-Acc}}$.

GEN: On input (gen, sid) from $\mathcal{AM}$ the functionality does the following:

1. If this is not the first gen command, or if sid does not encode the identity of $\mathcal{AM}$, ignore this command. Otherwise, continue.
2. $t \leftarrow 0$.
3. Initialize an empty list A (keeps track of all accumulator states).
4. Initialize map S, and set $S[0] \leftarrow \emptyset$ (maps operation counters to current accumulated sets).
5. Send (gen, sid) to $\mathcal{S}$.
6. Get $(algorithms, sid, (\mathsf{Gen}, \mathsf{Eval}, \mathsf{WitCreate}, \mathsf{Verify}, \mathsf{Add}, \mathsf{Delete}, \mathsf{WitUpdate}))$ from $\mathcal{S}$. Their expected input output behavior is described in Definition 3. All of them should be polynomial-time and Verify should be deterministic.
7. Run $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}(1^\lambda)$.
8. Store sk, pk; add $\Lambda_\emptyset \leftarrow \emptyset$ to A.
9. Send $(algorithms, sid, (\mathsf{Gen}, \mathsf{Eval}, \mathsf{WitCreate}, \mathsf{Verify}, \mathsf{Add}, \mathsf{Delete}, \mathsf{WitUpdate}))$ to $\mathcal{AM}$.

EVAL: On input $(eval, sid, \mathcal{X})$ from $\mathcal{AM}$ the functionality does the following:

1. If sid does not encode the identity of $\mathcal{AM}$, or if $\mathcal{X} \not\subset \mathsf{D}_\Lambda$, ignore this command. Otherwise, continue.
2. $t \leftarrow t + 1$, and $S[t] \leftarrow \mathcal{X}$.
3. Run $(\Lambda_\mathcal{X}, \mathsf{aux}) \leftarrow \mathsf{Eval}((\mathsf{sk}, \mathsf{pk}), \mathcal{X})$.
4. Store aux; add $\Lambda_\mathcal{X}$ to A.
5. Send $(eval, sid, \Lambda_\mathcal{X}, \mathcal{X})$ to $\mathcal{AM}$.

WITCREATE: On input $(witcreate, sid, x)$ from $\mathcal{AM}$ the functionality does the following:

1. If sid does not encode the identity of $\mathcal{AM}$, or if $x \notin \mathsf{D}_\Lambda$, ignore this command. Otherwise, continue.
2. Run $w \leftarrow \mathsf{WitCreate}((\mathsf{sk}, \mathsf{pk}), \Lambda_\mathcal{X}, \mathsf{aux}, x)$.
3. If $x \notin S[t]$, send $\perp$ to $\mathcal{AM}$ and halt. Otherwise, continue.
4. If $\mathsf{Verify}(\mathsf{pk}, \Lambda_\mathcal{X}, w, x) = 1$ continue. Otherwise, send $\perp$ to $\mathcal{AM}$ and halt.
5. Send $(witness, sid, x, w)$ to $\mathcal{AM}$.

VERIFY: On input $(verify, sid, \Lambda, \mathsf{Verify}', x, w)$ from party $\mathcal{V}$ the functionality does the following:

1. If $\mathsf{Verify}' = \mathsf{Verify} \wedge \Lambda \in A$:
   (a) $t \leftarrow$ largest t such that $S[t]$ corresponds to Λ.
   (b) If $\mathcal{AM}$ not corrupted $\wedge\ x \notin S[t] \wedge \mathsf{Verify}(\mathsf{pk}, \Lambda, x, w) = 1$, send $\perp$ to $\mathcal{P}$. Otherwise, continue.
   (c) $b \leftarrow \mathsf{Verify}(\mathsf{pk}, \Lambda, w, x)$
   Otherwise, set $b = \mathsf{Verify}'(\mathsf{pk}, \Lambda, w, x)$.
2. Send $(verified, sid, \Lambda, \mathsf{Verify}', x, w, b)$ to $\mathcal{V}$.

ADD: On input (add, sid, x) from $\mathcal{AM}$ the functionality does the following:

1. If sid does not encode the identity of $\mathcal{AM}$, or if $x \notin D_\Lambda$, ignore this command. Otherwise, continue.
2. If $x \in S[t]$ send $\perp$ to $\mathcal{AM}$ and halt. Otherwise, continue.
3. $t \leftarrow t + 1$, and $S[t] \leftarrow S[t-1]$.
4. Run $(\Lambda_{\mathcal{X}'}, \mathsf{aux}') \leftarrow \mathsf{Add}((\mathsf{sk}, \mathsf{pk}), \Lambda_\mathcal{X}, \mathsf{aux}, x)$.
5. Run $w \leftarrow \mathsf{WitCreate}((\mathsf{sk}, \mathsf{pk}), \Lambda_{\mathcal{X}'}, \mathsf{aux}', x)$.
6. If $\mathsf{Verify}(\mathsf{pk}, \Lambda_{\mathcal{X}'}, w, x) = 0$, send $\perp$ to $\mathcal{AM}$ and halt. Otherwise, continue.
7. Store aux'; add x to $S[t]$ and $\Lambda_{\mathcal{X}'}$ to A.
8. Send $(added, sid, \Lambda_{\mathcal{X}'}, x)$ to $\mathcal{AM}$.

DELETE: On input $(delete, sid, x)$ from $\mathcal{AM}$ the functionality does the following:

1. If sid does not encode the identity of $\mathcal{AM}$, or if $x \notin D_\Lambda$, ignore this command. Otherwise, continue.
2. If $x \notin S[t]$ send $\perp$ to $\mathcal{AM}$ and halt. Otherwise, continue.
3. $t \leftarrow t + 1$, and $S[t] \leftarrow S[t-1]$.
4. Run $(\Lambda_{\mathcal{X}'}, \mathsf{aux}') \leftarrow \mathsf{Delete}((\mathsf{sk}, \mathsf{pk}), \Lambda_X, \mathsf{aux}, x)$.
5. Store aux'; remove x from $S[t]$ and add $\Lambda_{\mathcal{X}'}$ to A.
6. Send $(deleted, sid, \Lambda_{\mathcal{X}'}, x)$ to $\mathcal{AM}$.

WITUPDATE: On input $(witupdate, sid, \Lambda_{\mathcal{X}_{\text{old}}}, \Lambda_{\mathcal{X}_{\text{new}}}, x, w_{\text{old}})$ from a party $\mathcal{H}$, the functionality does the following:

1. If $\Lambda_{\mathcal{X}_{\text{old}}} \notin A \vee \Lambda_{\mathcal{X}_{\text{new}}} \notin A$, send $\perp$ to $\mathcal{H}$ and halt. Otherwise continue.
2. Run $w_{\text{new}} \leftarrow \mathsf{WitUpdate}((\mathsf{sk}, \mathsf{pk}), w_{\text{old}}, \mathsf{aux}, x)$.
3. If $\mathsf{Verify}(\mathsf{pk}, \Lambda_{\mathcal{X}_{\text{old}}}, w_{\text{old}}, x) = 1 \wedge x \in S[t] \wedge \mathsf{Verify}(\mathsf{pk}, \Lambda_{\mathcal{X}_{\text{new}}}, w_{\text{new}}, x) = 0$, send $\perp$ to $\mathcal{V}$ and halt. Otherwise, continue.
4. Send $(updatedwit, sid, \Lambda_{\mathcal{X}_{\text{old}}}, \Lambda_{\mathcal{X}_{\text{new}}}, x, w_{\text{old}}, w_{\text{new}})$ to $\mathcal{H}$.

Functionality 1: Ideal Functionality $\mathcal{F}_{\text{Acc}}$ for dynamic accumulators

GEN: On input (gen, sid_i) from all parties $\mathcal{AM}_i$, the functionality does the following:

1. If this is not the first gen command, or if for any $i \in [n]$, sid_i does not encode the identity of $\mathcal{AM}_i$, ignore this command. Otherwise, continue.
2. $t \leftarrow 0$.
3. Initialize an empty list A (keeps track of all accumulator states).
4. Initialize map S, and set $S[0] \leftarrow \emptyset$ (maps operation counters to current accumulated sets).
5. Send $(gen, sid_i)_{i\in[n]}$ to $\mathcal{S}$.
6. Get $(algorithms, (sid_1, \ldots, sid_n), (\mathsf{Gen}, \mathsf{Eval}, \mathsf{WitCreate}, \mathsf{Verify}, \mathsf{Add}, \mathsf{Delete}, \mathsf{WitUpdate}))$ from $\mathcal{S}$. Their expected input output behavior is described in Definition 6. All of them should be polynomial-time and Verify should be deterministic.
7. Run $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}(1^\lambda)$. Store sk, pk; add $\Lambda_\emptyset \leftarrow \emptyset$ to A.
8. Send $(algorithms, sid_i, (\mathsf{Gen}, \mathsf{Eval}, \mathsf{WitCreate}, \mathsf{Verify}, \mathsf{Add}, \mathsf{Delete}, \mathsf{WitUpdate}))$ to $\mathcal{AM}_i$, for all $i = 1, \ldots, n$.

EVAL: On input $(eval, sid_k, \mathcal{X})$ from $\mathcal{AM}_k$ and $(eval, sid_j, ?)$ from all other parties $\mathcal{AM}_j$, for $j \neq k$, the functionality does the following:

1. If for any $i \in [n]$, sid_i does not encode the identity of $\mathcal{AM}_i$, or if $\mathcal{X} \not\subset \mathsf{D}_\Lambda$, ignore this command. Otherwise, continue.
2. $t \leftarrow t + 1$, and $S[t] \leftarrow \mathcal{X}$.
3. Run $(\Lambda_\mathcal{X}, \mathsf{aux}) \leftarrow \mathsf{Eval}((\mathsf{sk}, \mathsf{pk}), \mathcal{X})$. Store aux; add $\Lambda_\mathcal{X}$ to A.
4. Send $(eval, sid_i, \Lambda_\mathcal{X}, \mathcal{X})$ to $\mathcal{AM}_i$, for all $i = 1, \ldots, n$.

WITCREATE: On input $(witcreate, sid_k, x)$ from $\mathcal{AM}_k$ and $(witcreate, sid_j, ?)$ from all parties $\mathcal{AM}_j$, for $j \neq k$, the functionality does the following:

1. If for any $i \in [n]$, sid_i does not encode the identity of $\mathcal{AM}_i$, or if $x \notin \mathsf{D}_\Lambda$, ignore this command. Otherwise, continue.
2. Run $w \leftarrow \mathsf{WitCreate}((\mathsf{sk}, \mathsf{pk}), \Lambda_\mathcal{X}, \mathsf{aux}, x)$.
3. If $x \notin S[t]$, send $\perp$ to all $\mathcal{AM}_i$ and halt. Otherwise, continue.
4. If $\mathsf{Verify}(\mathsf{pk}, \Lambda_\mathcal{X}, w, x) = 1$ continue. Otherwise, send $\perp$ to all $\mathcal{AM}_\rangle$ and halt.
5. Send $(witness, sid_i, x, w)$ to $\mathcal{AM}_i$, for all $i = 1, \ldots, n$.

VERIFY: On input $(verify, sid, \Lambda, \mathsf{Verify}', x, w)$ from party $\mathcal{V}$ the functionality does the following:

1. If $\mathsf{Verify}' = \mathsf{Verify} \wedge \Lambda \in A$:
   (a) $t \leftarrow$ largest t such that $S[t]$ corresponds to Λ.
   (b) If at least one $\mathcal{AM}_i$ is not corrupted $\wedge\ x \notin S[t] \wedge \mathsf{Verify}(\mathsf{pk}, \Lambda, x, w) = 1$, send $\perp$ to $\mathcal{V}$. Otherwise, continue.
   (c) $b \leftarrow \mathsf{Verify}(\mathsf{pk}, \Lambda, w, x)$
   Otherwise, set $b = \mathsf{Verify}'(\mathsf{pk}, \Lambda, w, x)$.
2. Send $(verified, sid, \Lambda, \mathsf{Verify}', x, w, b)$ to $\mathcal{V}$.

ADD: On input (add, sid_k, x) from $\mathcal{AM}_k$ and $(add, sid_j, ?)$ from all parties $\mathcal{AM}_j$, for $j \neq k$, the functionality does the following:

1. If for any $i \in [n]$, sid_i does not encode the identity of $\mathcal{AM}_i$, or if $x \notin D_\Lambda$, ignore this command. Otherwise, continue.
2. If $x \in S[t]$ send $\perp$ to all $\mathcal{AM}_i$ and halt. Otherwise, continue.
3. $t \leftarrow t + 1$, and $S[t] \leftarrow S[t-1]$.
4. Run $(\Lambda_{\mathcal{X}'}, \mathsf{aux}') \leftarrow \mathsf{Add}((\mathsf{sk}, \mathsf{pk}), \Lambda_X, \mathsf{aux}, x)$. Run $w \leftarrow \mathsf{WitCreate}((\mathsf{sk}, \mathsf{pk}), \Lambda_{\mathcal{X}'}, \mathsf{aux}', x)$.
5. If $\mathsf{Verify}(\mathsf{pk}, \Lambda_{\mathcal{X}'}, w, x) = 0$, send $\perp$ to all $\mathcal{AM}_i$ and halt. Otherwise, continue.
6. Store aux'; add x to $S[t]$ and $\Lambda_{\mathcal{X}'}$ to A.
7. Send $(added, sid_i, \Lambda_{\mathcal{X}'}, x)$ to $\mathcal{AM}_i$, for all $i = 1, \ldots, n$.

DELETE: On input $(delete, sid_k, x)$ from $\mathcal{AM}_k$ and $(delete, sid_j, ?)$ from all parties $\mathcal{AM}_j$, for $j \neq k$, the functionality does the following:

1. If for any $i \in [n]$, sid_i does not encode the identity of $\mathcal{AM}_i$, or if $x \notin D_\Lambda$, ignore this command. Otherwise, continue.
2. If $x \notin S[t]$ send $\perp$ to all $\mathcal{AM}_i$ and halt. Otherwise, continue.
3. $t \leftarrow t + 1$, and $S[t] \leftarrow S[t-1]$.
4. Run $(\Lambda_{\mathcal{X}'}, \mathsf{aux}') \leftarrow \mathsf{Delete}((\mathsf{sk}, \mathsf{pk}), \Lambda_X, \mathsf{aux}, x)$.
5. Store aux'; remove x from $S[t]$ and add $\Lambda_{\mathcal{X}'}$ to A.
6. Send $(deleted, sid_i, \Lambda_{\mathcal{X}'}, x)$ to $\mathcal{AM}_i$, for all $i = 1, \ldots, n$.

WITUPDATE: On input $(witupdate, sid_k, \Lambda_{\mathcal{X}_{\text{old}}}, \Lambda_{\mathcal{X}_{\text{new}}}, x, w_{\text{old}})$ from $\mathcal{AM}_k$ and $(witupdate, sid_j, ?, ?, ?, ?)$ from all parties $\mathcal{AM}_j$, for $j \neq k$, the functionality does the following:

1. If for any $i \in [n]$, sid_i does not encode the identity of $\mathcal{AM}_i$, or if $x \notin D_\Lambda$, ignore this command. Otherwise, continue.
2. If $\Lambda_{\mathcal{X}_{\text{old}}} \notin A \vee \Lambda_{\mathcal{X}_{\text{new}}} \notin A$, send $\perp$ to all $\mathcal{AM}_i$ and halt. Otherwise continue.
3. Run $w_{\text{new}} \leftarrow \mathsf{WitUpdate}((\mathsf{sk}, \mathsf{pk}), w_{\text{old}}, \mathsf{aux}, x)$.
4. If $\mathsf{Verify}(\mathsf{pk}, \Lambda_{\mathcal{X}_{\text{old}}}, w_{\text{old}}, x) = 1 \wedge x \in S[t] \wedge \mathsf{Verify}(\mathsf{pk}, \Lambda_{\mathcal{X}_{\text{new}}}, w_{\text{new}}, x) = 0$, send $\perp$ to $\mathcal{V}$ and halt. Otherwise, continue.
5. Send $(updatedwit, sid_i, \Lambda_{\mathcal{X}_{\text{old}}}, \Lambda_{\mathcal{X}_{\text{new}}}, x, w_{\text{old}}, w_{\text{new}})$ to $\mathcal{AM}_i$, for all $i = 1, \ldots, n$.

Functionality 2: Ideal Functionality $\mathcal{F}_{\text{Acc-MPC}}$

D Strong-RSA Accumulator

We recall the strong RSA assumption [BP97].

Definition 8 (Strong RSA assumption). *Given two appropriately chosen primes p and q such that $N = pq$ has bit-length κ, then it holds for all PPT adversaries $\mathcal{A}$ that*

$$\Pr\left[u \xleftarrow{R} \mathbb{Z}_N^*, (v, w) \leftarrow \mathcal{A}(u, N) \colon v^w \equiv u \pmod{N}\right]$$

is negligible in the security parameter κ.

Major lines of work investigated accumulators in the hidden order groups, i.e., RSA-based, and the known order groups, i.e., discrete logarithm-based, setting. The first collision-free RSA-based accumulator is due to Baric and Pfitzmann [BP97]. The accumulator in this construction consists of a generator raised to the product of all elements of the set. Then witnesses essentially consist of the same value skipping the respective elements in the product. Thereby, the witness can easily be verified by raising the power of the witness to the element and checking if the result matches the accumulator. We recall the RSA-based accumulator in Scheme 4.

$\underline{\mathsf{Gen}(1^\kappa, t)}$: Choose an RSA modulus $N = p \cdot q$ with two large safe primes p, q, and let g be a random quadratic residue $\bmod N$. Set $\mathsf{sk}_\Lambda \leftarrow \emptyset$ and $\mathsf{pk}_\Lambda \leftarrow (N, g)$.

$\underline{\mathsf{Eval}((\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda), \mathcal{X})}$: Parse pk_Λ as (N, g) and let $\mathcal{X} \subset \mathbb{P}$. Return $\Lambda_\mathcal{X} \leftarrow g^{\prod_{x \in \mathcal{X}} x} \bmod N$ and $\mathsf{aux} \leftarrow (\mathsf{add} \leftarrow 0, \mathcal{X})$.

$\underline{\mathsf{WitCreate}((\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda), \Lambda_\mathcal{X}, \mathsf{aux}, x)}$: If $x \notin \mathcal{X}$, return $\perp$. If $\mathsf{sk}_\Lambda \neq \emptyset$, return $\mathsf{wit}_x \leftarrow \Lambda_\mathcal{X}^{x^{-1}} \bmod N$, otherwise return $\mathsf{wit}_x \leftarrow g^{\prod_{x' \in \mathcal{X} \setminus \{x\}} x'} \bmod N$.

$\underline{\mathsf{Verify}(\mathsf{pk}_\Lambda, \Lambda_\mathcal{X}, \mathsf{wit}_x, x)}$: Parse pk_Λ as (N, g). If $\mathsf{wit}_x^x = \Lambda_\mathcal{X} \bmod N$ holds, return 1, otherwise return 0.

$\underline{\mathsf{Add}((\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda), \Lambda_\mathcal{X}, \mathsf{aux}, x)}$: Parse pk_Λ as (N, g) and aux as $\mathcal{X}$. If $x \in \mathcal{X}$, return $\perp$. Set $\mathcal{X}' \leftarrow \mathcal{X} \cup \{x\}$, $\mathsf{aux}' \leftarrow (\mathcal{X}', \mathsf{add} \leftarrow 1)$, and $\Lambda_{\mathcal{X}'} \leftarrow \Lambda_\mathcal{X}^x \bmod N$. Return $\Lambda_{\mathcal{X}'}$ and aux'.

$\underline{\mathsf{Delete}((\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda), \Lambda_\mathcal{X}, \mathsf{aux}, x)}$: Parse pk_Λ as (N, g) and aux as $\mathcal{X}$. If $x \notin \mathcal{X}$, return $\perp$. If $\mathsf{sk}_\Lambda \neq \emptyset$, set $\mathcal{X}' \leftarrow \mathcal{X} \setminus \{x\}$, $\mathsf{aux}' \leftarrow (\mathcal{X}', \mathsf{add} \leftarrow -1)$, and $\Lambda_{\mathcal{X}'} \leftarrow \Lambda_\mathcal{X}^{x^{-1}} \bmod N$. Otherwise, compute $(\Lambda_{\mathcal{X}'}, \mathsf{aux}') \leftarrow \mathsf{Eval}((\emptyset, \mathsf{pk}_\Lambda), \mathcal{X} \setminus \{x\})$ with $\mathsf{add} \leftarrow -1$ in aux'. Return $\Lambda_{\mathcal{X}'}$ and aux'.

$\underline{\mathsf{WitUpdate}((\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda), \mathsf{wit}_{x_i}, \mathsf{aux}, x)}$: Parse pk_Λ as (N, g) and aux as $(\mathcal{X}, \mathsf{add})$. If $\mathsf{add} = 0$, return $\perp$. Return $\mathsf{wit}_{x_i}^x \bmod N$ if $\mathsf{add} = 1$. If instead $\mathsf{add} = -1$ and $\mathsf{sk}_\Lambda \neq \emptyset$, then return $\mathsf{wit}_{x_i}^{x^{-1}} \bmod N$. Otherwise, compute $a, b \in \mathbb{Z}$ such that $ax_i + bx = 1$ and return $\mathsf{wit}_{x_i}^b \cdot \Lambda_\mathcal{X}^a \bmod N$. In the last two cases in addition return $\mathsf{aux} \leftarrow (\mathsf{add} \leftarrow 0)$.

Scheme 4: Strong RSA-based accumulator.

Note that, whenever the factorization of N is available, the Chinese remainder theorem can be used to speed up the computations. For WitCreate and WitUpdate

we can use the factorization to compute inverses $\mod (p-1)\cdot(q-1)$. Deletion of values from the accumulator is not possible if the factorization is unknown.

Correctness can easily be verified and collision freeness follows from the strong RSA assumption:

Theorem 4 ([BP97]). *If the strong RSA assumption holds, Scheme 4 is collision-free.*

Again, from Theorems 2 and 4, it follows that Scheme 4 is also secure in the UC model:

Corollary 2. *Scheme 4 emulates $\mathcal{F}_{Acc}$ in the UC model.*

D.1 Dynamic (Threshold) Secret-Shared Accumulator From the Strong RSA Assumption

For our dynamic (threshold) secret-shared accumulator from the strong RSA assumption, observe that the two main operations that have to be performed in the context of an MPC protocol are the sampling of the secret prime factors as well as the computation of the inverses of the public elements in the exponent. Both operations are also performed during RSA key generation with the sole difference that in this case, the public exponent is inverted. Therefore, we can make use of any protocol for distributed RSA key generation. In particular, our construction makes use of the state-of-the-art protocol by Frederiksen et al. [FLOP18] for the two-party case.

In the following, we will recap the structure of this UC secure two-party protocol and highlight the most essential parts of the protocol. The key generation in the malicious setting consists of the following four phases:

Candidate Generation: Both parties P_1 and P_2 choose random shares $p_1 \in \mathbb{N}$ respectively $p_2 \in \mathbb{N}$ and commit to it. Based on maliciously secure OT, they do a secure trial division of $p = p_1 + p_2$ with public threshold $B_1 \in \mathbb{N}$.

Construct Modulus: Given shared candidate primes $p = p_1 + p_2, q = q_1 + q_2$ the parties compute $N = pq$ by a custom version of the Gilboa protocol [Gil99]. The candidate modulus N is send to both parties.

Verify Modulus: This phase consists of three steps:

1. Second trial division with threshold $B_2 > B_1$.
2. Secure biprimality test.
3. Proof of honesty that checks the commitments and whether $\gcd(e, \phi(N)) = 1$, where e is the public exponent.

Construct Keys: Computes the shares of $d = d_1 + d_2$ such that $e(d_1 + d_2) \equiv 1 \mod \phi(N)$ (uses additional output from the proof of honesty).

We will denote this protocol by Π_{RSA} and by $\mathcal{F}_{\text{RSA}}$ its ideal functionality. At a later point in the RSA-based accumulator, we need to invert an element $x \in \mathbb{P}$ in $\mathbb{Z}_N^*$. Since neither P_1 nor P_2 knows the order $\phi(N)$ of this ring, we employ parts of the MPC protocol. For this task, we will perform the 3rd step of **Verify Modulus** with $e = x$, and then immediately run **Construct Keys**. The output

$\underline{\mathsf{Gen}(1^\kappa, t)}$: Generate an RSA modulus $N = (p_1+p_2)\cdot(q_1+q_2)$ via the protocol Π_{RSA}, where the party P_i receives (p_i, q_i) for $i = 1, 2$. Further, let g be a random quadratic residue $\mod N$. Set $\mathsf{pk}_\Lambda \leftarrow (N, g)$.

$\underline{\mathsf{Eval}((\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda), \mathcal{X})}$: Parse pk_Λ as (N, g) and $\mathcal{X} \subset \mathbb{P}$. Return $\Lambda_\mathcal{X} \leftarrow g^{\prod_{x\in\mathcal{X}} x} \mod N$ and $\mathsf{aux} \leftarrow \mathcal{X}$.

$\underline{\mathsf{WitCreate}(((p_1, q_1), (p_2, q_2), \mathsf{pk}_\Lambda), \Lambda_\mathcal{X}, \mathsf{aux}, x)}$: If $x \notin \mathcal{X}$, return $\perp$. Compute $\langle y\rangle \leftarrow \mathsf{Invert}_{\phi(N)}(x)$ and return $\mathsf{wit}_x \leftarrow \mathsf{Open}(\Lambda_\mathcal{X}^{\langle y\rangle} \mod N)$.

$\underline{\mathsf{Verify}(\mathsf{pk}_\Lambda, \Lambda_\mathcal{X}, \mathsf{wit}_x, x)}$: Parse pk_Λ as (N, g). If $\mathsf{wit}_x^x = \Lambda_\mathcal{X} \mod N$ holds, return 1, otherwise return 0.

$\underline{\mathsf{Add}(\mathsf{pk}_\Lambda, \Lambda_\mathcal{X}, \mathsf{aux}, x)}$: Parse pk_Λ as (N, g) and aux as $\mathcal{X}$. If $x \in \mathcal{X}$, return $\perp$. Set $\mathcal{X}' \leftarrow \mathcal{X} \cup \{x\}$, $\mathsf{aux}' \leftarrow (\mathcal{X}', \mathrm{add} \leftarrow 1)$, and $\Lambda_{\mathcal{X}'} \leftarrow \Lambda_\mathcal{X}^x \mod N$. Return $\Lambda_{\mathcal{X}'}$ and aux'.

$\underline{\mathsf{Delete}(((p_1, q_1), (p_2, q_2), \mathsf{pk}_\Lambda), \Lambda_\mathcal{X}, \mathsf{aux}, x)}$: Parse pk_Λ as (N, g) and aux as $\mathcal{X}$. If $x \notin \mathcal{X}$, return $\perp$. Set $\mathcal{X}' \leftarrow \mathcal{X} \setminus \{x\}$, $\mathsf{aux}' \leftarrow (\mathcal{X}', \mathrm{add} \leftarrow -1)$, and $\langle y\rangle \leftarrow \mathsf{Invert}_{\phi(N)}(x)$. Return $\Lambda_{\mathcal{X}'} \leftarrow \mathsf{Open}(\Lambda_\mathcal{X}^{\langle y\rangle} \mod N)$ and aux'.

$\underline{\mathsf{WitUpdate}(((p_1, q_1), (p_2, q_2), \mathsf{pk}_\Lambda), \mathsf{wit}_{x_i}, \mathsf{aux}, x)}$: Parse pk_Λ as (N, g) and aux as $(\mathcal{X}, \mathrm{add})$. Return $\perp$ if $\mathrm{add} = 0 \vee x_i \notin \mathcal{X}$. Return $\mathsf{wit}_{x_i}^x \mod N$ if $\mathrm{add} = 1$. If instead $\mathrm{add} = -1$ and compute $\langle y\rangle \leftarrow \mathsf{Invert}_{\phi(N)}(x)$, then return $\mathsf{Open}(\mathsf{wit}_{x_i}^{\langle y\rangle} \mod N)$. In the last two cases in addition return $\mathsf{aux} \leftarrow (\mathrm{add} \leftarrow 0)$.

Scheme 5: MPC-RSA: Dynamic (threshold) secret-shared accumulator based on RSA for two parties.

of this sub-protocol, which we will denote by $\mathsf{Invert}_{\phi(N)}(x)$, is a secret-shared value $\langle y\rangle = y_1+y_2$, where y_i is the share of party i, s.t. $x(y_1+y_2) \equiv 1 \mod \phi(N)$.

However, we have to note that the used distributed RSA key generation protocols may leak a small amount of secret information. We refer to the full version of [FLOP18] for a detailed discussion of this leakage. Since our protocol may invert multiple elements, the parties might need to add bounds on the maximum number of operations to prevent leakage of the secret key.

The last open point during Gen is the sampling of the quadratic non-residue mod N. An option here is to simply sample g at random from $\mathbb{Z}_N^*$ and checking whether the Jacobi symbol satisfies $(\frac{g}{N}) = -1$. Despite its definition as the product of the Legendre symbols of the prime factors of N, the Jacobi symbol can be computed efficiently without knowledge of the prime factors using an algorithm analogous to the Euclidean algorithm. Hence, we perform this step outside of the MPC protocol once N was generated.

In the security analysis of Scheme 5, we can reuse the arguments of Theorem 3. Therefore, we will omit the proof of the following theorem. We, however, note, that the theorem only holds provided that the parties keep track of the number of potentially leaked bits and abort if this number gets too large.

Theorem 5. *Scheme 5 UC emulates $\mathcal{F}_{Acc\text{-}MPC}$ in the $\mathcal{F}_{ABB+}, \mathcal{F}_{RSA}$-hybrid model.*

Table 9. Offline phase performance of different steps of the MPC-q-SDH accumulator with access to the secret trapdoor implemented in FRESCO. Time in seconds.

Operation	$\lvert\mathcal{X}\rvert$	LAN setting $n=$ 2	3	4	5	WAN setting 2	3	4	5
BaseOTs	2^{10}	21.5	60.3	104.6	148.9	76.5	215.0	364.2	504.3
	2^{14}	21.5	60.3	104.6	148.9	76.5	215.0	364.2	504.3
Eval	2^{10}	54.4	154.0	265.0	409.1	95.7	283.1	465.2	683.3
	2^{14}	825.8	2 259.5	4 048.8	6 150.9	1 449.8	3 587.1	6 578.4	10 056.3
Inverse	2^{10}	1.3	15.3	16.8	19.7	3.2	68.2	70.4	72.8
	2^{10}	1.3	15.3	16.8	19.7	3.2	68.2	70.4	72.8

E FRESCO Benchmarks

In this section, we discuss the benchmarks of our q-SDH implementation in the maliciously secure dishonest-majority setting in the FRESCO framework. FRESCO is a Java framework facilitating fast prototyping of MPC-based applications and protocols. FRESCO implements the SPDZ protocol and various extensions [DPSZ12,KOS15,KOS16,CDE+18]. For pairing and elliptic curve group operations, we rely on the ECCelerate library[15] and integrate $\mathsf{Exp}_{\mathbb{G}}$, Output-$\mathbb{G}$, and the corresponding $\mathrm{MACCheck}_{\mathrm{ECC}}$ algorithms of [SA19] into FRESCO. As a pairing-friendly curve, a 400-bit Barreto-Naehrig curve [BN05] is used, which provides around 100 bit of security following recent estimates [BD19].

The offline phase performance of our FRESCO implementation can be seen in Table 9. A comparison to our MP-SPDZ implementation (Table 3) shows that the offline phase in FRESCO is not yet as optimized as the one of MP-SPDZ (as an example, the BaseOTs used in FRESCO are not using elliptic curve arithmetic) and that the performance of the latter is more indicative of an optimized implementation. Further note that batching the generation of many triples together like for the Eval phase is more efficient in practice than producing a single triple and as these triples are not dependent on the input, all parties can continuously generate triples in the background to fill a triple-buffer for use in the online phase.

In Table 10, we present the online phase performance of our q-SDH implementation in the FRESCO framework. Since the FRESCO implementation does not support a depth-optimized tree-like multiplication, the Eval operation scales worse with the number of parties. Compared to our MP-SPDZ implementation (Table 4) it is, therefore, much slower, especially in the WAN setting.

Finally, we present the communication cost of our q-SDH implementation in the FRESCO framework in Table 11. Again, a comparison to our MP-SPDZ implementation (Table 5) shows, that the FRESCO framework requires more communication to achieve the same result. While some of this overhead can be

[15] https://jce.iaik.tugraz.at/sic/Products/Core_Crypto_Toolkits/ECCelerate

Table 10. Online phase performance of the MPC-q-SDH accumulator with access to the secret trapdoor implemented in FRESCO, for both the LAN and WAN settings with n parties. Time in milliseconds averaged over 50 executions.

Operation	$\lvert\mathcal{X}\rvert$	LAN setting $n =$ 2	3	4	5	WAN setting 2	3	4	5
Gen	2^{10}	78	106	301	772	604	1 124	1 399	1 684
	2^{14}	75	110	246	766	608	1 118	1 401	1 673
Eval	2^{10}	133	354	4 062	14 193	52 316	56 802	58 728	60 043
	2^{14}	1 389	4 362	61 222	196 563	828 522	892 293	918 445	935 274
WitCreate	2^{10}	40	84	279	906	1 109	1 858	2 156	2 444
	2^{14}	41	98	267	992	1 120	1 853	2 152	2 442
Add	2^{10}	43	78	231	646	574	1 088	1 373	1 650
	2^{14}	41	74	225	706	571	1 087	1 363	1 642
$\mathsf{WitUpdate}_{\mathrm{Add}}$	2^{10}	40	72	259	745	569	1 094	1 372	1 649
	2^{14}	40	85	213	732	564	1 088	1 369	1 644
Delete	2^{10}	47	80	299	958	1 075	1 849	2 150	2 454
	2^{14}	42	92	297	857	1 071	1 847	2 152	2 446
$\mathsf{WitUpdate}_{\mathrm{Delete}}$	2^{10}	40	82	258	977	1 071	1 852	2 151	2 447
	2^{14}	38	38	294	880	1 077	1 852	2 149	2 441

attributed to the different choice of elliptic curve, the rest is inherent to the implementation of the framework.

Table 11. Communication cost (in MiB per party) of the MPC-q-SDH accumulator with access to the secret trapdoor implemented in FRESCO.

Operations	$\lvert\mathcal{X}\rvert$	Offline phase[a]	Online Phase[b]
Gen	2^{10}	0.297	0.103
	2^{14}	0.297	0.103
Eval	2^{10}	220.971	0.176
	2^{14}	3 530.148	3.164
WitCreate, Delete, $\mathsf{WitUpdate}_{\mathrm{Delete}}$	2^{10}	0.634	0.041
	2^{14}	0.634	0.041
Add, $\mathsf{WitUpdate}_{\mathrm{Add}}$	2^{10}	0.297	0.039
	2^{14}	0.297	0.039

[a] Includes BaseOTs for a new connection
[b] Includes the setup of a fresh MAC for each share of the secret trapdoor