Multi-party Revocation in Sovrin: Performance through Distributed Trust

Lukas Helminger[1,2], Daniel Kales[1], Sebastian Ramacher[3] ,
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[1]

---

[1] https://sovrin.org/.

are of central interest as underlined by a recent push in the European Union for a cross-border SSI system.[2] 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 [3,13,31] among many others.

One promising approach regarding efficiency is based on denylists (or allowlists) via cryptographic accumulators which were introduced by Benaloh and de Mare [11]. 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 [37] and Hyperledger Indy[3] such an accumulator-based approach with allowlists following the ideas of [31] 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 [44] have seen many applications in both the cryptographic literature but also in practice. For example, they have been used to implement Certificate Transparency (CT) [38] where all issued certificates are publicly logged, i.e., accumulated. Accumulators also find application in credentials [13], ring, and group signatures [26,39], anonymous cash [45], among many 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. [35]). 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. [31] 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 [47] 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 [16,17,29], or distributed ECDSA key generation [41].

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 [21,24] and honest majority threshold protocols based on Shamir secret sharing [48].

1.2   Our Contribution

Starting from the very recent treatment of accumulators in the UC model [15] by Baldimtsi et al. [4], 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 [25,46], which can be build from dishonest-majority full-threshold protocols (e.g., SPDZ [21,24]) and from honest-majority threshold MPC protocols (e.g., Shamir secret sharing [48]). 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 [36] and FRESCO.[4] 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

---

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

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 [37]. 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 [35].

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 [14]. 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 [9] to reduce the trust required in the CRS generator. Groth et al. [33] recently introduced the notion of an updatable CRS where first generic compilers [1] 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 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 [10].

Approaches that try to fix the issue directly in the formalization of accumulators and corresponding constructions have also been studied. For example, Lipmaa [42] 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. [12] 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 [34] and hyperelliptic curves of genus 2 or 3 [27].

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 [21,24] has seen a lot of interest, improvements and extensions. This interest also led to multiple MPC frameworks, e.g., MP-SPDZ [36],

FRESCO and SCALE-MAMBA,[5] 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).[6] For such a virtual HSM, one essentially wants to provide distributed key generation [29] together with threshold signatures [22] allowing to replace a physical HSM. Similar techniques are also interesting for securing wallets for the use in cryptocurrencies, where especially protocols for ECDSA [32,41] 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 [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 [50] for example. Recent works [49] 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 the full version.

2.1   UC Security and ABB

In this paper, we mainly work in the UC model first introduced by Canetti [15]. 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.

The importance of the UC model for secure multiparty computation stems from the arithmetic black box (ABB) [23]. 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 [21,24] and Shamir secret sharing [48], 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

---

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

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 [18] or [40].

We will denote the ideal functionality of the online protocol of SPDZ and Shamir secret sharing by $\mathcal{F}_{\mathrm{Abb}}$. 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 $\mathcal{F}_{\mathrm{Abb}}$ 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 [5]. 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 [49] introduced protocols – in particular based on SPDZ – for group operations of elliptic curve groups supporting secret exponents and secret group elements. 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 [49] 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. [25]. We recall definitions of static and dynamic accumulators in the full version.

2.4   Pairing-Based Accumulator

We recall the q-SDH-based accumulator from [25], which is based on the accumulator by Nguyen [46]. The idea here is to encode the accumulated elements in a

$\underline{\mathsf{Gen}(1^\kappa, q)}$: 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.

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 [30], 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:

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

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

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

The proof of collision freeness follows from the q-SDH assumption. For completeness, we still restate the theorem from [25] adopted to the Type-3 setting. For the proof, we refer to the full version.

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

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. [4] 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 [4] 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 can be found in the full version. 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 Gen, Eval, WitCreate, 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 [4] 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 $\mathcal{F}_{Acc}$.*

For the proof we refer to the full version. As a direct consequence of Theorems 1 and 2, the accumulator from Scheme 1 is also secure in the UC model of [4] 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 Sect. 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. [25], 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 1 (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* (Gen, Eval, WitCreate, Verify) *which are defined as follows:*

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.*

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.*

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 $\bot$, if $x \notin \mathcal{X}$, and a witness wit_x for x otherwise to every party P_i.*

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.

Definition 2 (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* (Add, Delete, WitUpdate) *which are defined as follows:*

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 $\bot$ 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.*

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 $\bot$ 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 $\bot$ otherwise to every party P_i.*

Correctness and collision-freeness naturally translate from the non-interactive accumulators to the (threshold) secret-shared ones.

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 can be found in the full version. 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.

$\underline{\mathsf{Gen}(1^\kappa, q)}$: 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 $\Lambda_{\mathcal{X}'} \leftarrow \Lambda_\mathcal{X} \cdot \mathsf{Open}(\Lambda_\mathcal{X}^{\langle\mathsf{sk}_\Lambda\rangle})$ 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 $\mathcal{F}_{ABB+}$. 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 $\mathcal{F}_{\mathrm{Acc}}$. Since we now also have to simulate the non-interactive sub-protocols, we have to extend $\mathcal{S}_{Acc}$. We construct $\mathcal{S}_{\mathrm{Acc\text{-}MPC}}$ by building upon $\mathcal{S}_{\mathrm{Acc}}$ and in addition internally simulate $\mathcal{F}_{\mathrm{ABB+}}$. As described in Sect. 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 the public key is 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 necessary values enabling trade-offs between an efficient Eval and an inefficient Gen. 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 ($k \leq \frac{n-1}{2}$) 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.

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

Accu.	$|\mathcal{X}|$	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$^{\mathrm{a}}$	1.12	0.05$^{\mathrm{a}}$	1.12	0.05$^{\mathrm{a}}$
	2^{14}	–	15.53	0.83$^{\mathrm{a}}$	15.53	0.83$^{\mathrm{a}}$	15.53	0.83$^{\mathrm{a}}$

$^{\mathrm{a}}$Assuming that the full Merkle-tree is known as auxiliary data.

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.[7] Our primary implementations are based on SPDZ with OT-based preprocessing and Shamir secret sharing in the MP-SPDZ [36][8] 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 the full version.

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 1 should be used.

MP-SPDZ implements the SPDZ protocol with various extensions, as well as semi-honest and malicious variants of Shamir secret sharing [18,19,40]. For pairing and elliptic curve group operations, we rely on relic[9] and integrate $\mathsf{Exp}_{\mathbb{G}}$, Output-$\mathbb{G}$, and the corresponding operations to update the MAC described in [49] into MP-SPDZ. We use the pairing friendly BLS12-381 curve [7], which provides around 120 bit of security following recent estimates [6]. For completeness, we also implemented the q-SDH accumulator from Scheme 1 and a Merkle-tree accumulator 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.

---

[7] The source code is available at https://github.com/IAIK/MPC-Accumulator.

[8] https://github.com/data61/MP-SPDZ.

[9] https://github.com/relic-toolkit/relic.

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[a]	0	$\|\mathcal{X}\|$	1	0	0	1	1
Random values	1	1	1	0	0	1	1
Opening rounds	1	$\lceil \log_2(\|\mathcal{X}\| + 1) \rceil + 1$	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 [8] 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 $|\mathcal{X}|$. 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 $|\mathcal{X}|$ 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 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

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.

	Operation	$|\mathcal{X}|$	LAN setting				WAN setting			
			$n = 2$	3	4	5	2	3	4	5
Semi-honest	BaseOTs	2^{10}, 2^{14}	0.03	0.08	0.14	0.23	0.14	0.31	0.56	0.84
	Inverse	2^{10}, 2^{14}	0.78	1.72	3.06	4.03	209.9	227.5	322.8	331.0
	Gen	2^{10}, 2^{14}	0.44	1.21	1.76	3.01	207.7	223.6	325.9	332.0
	Eval	2^{10}	189	397	706	959	4 695	8 215	13 680	25 725
		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
	Gen	2^{10}, 2^{14}	2.56	4.23	6.80	9.32	841.3	1 235	1 540	1 856
	Eval	2^{10}	1 601	2 849	4 345	6 227	25 737	45 254	87 328	141 181
		2^{14}	31 099	62 978	89 132	145 574	412 747	682 033	1 364 660	2 236 860

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 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 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

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.

Operation	$|\mathcal{X}|$	Semi-honest								Malicious							
		LAN setting				WAN setting				LAN setting				WAN setting			
		$n=2$	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
WitUpdate$_{\text{Add}}$	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
WitUpdate$_{\text{Delete}}$	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.

Operations	$|\mathcal{X}|$	Semi-honest		Malicious	
		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, WitUpdate$_{\text{Delete}}$	2^{10}, 2^{14}	33	0.15	164	0.37
Add, WitUpdate$_{\text{Add}}$	2^{10}, 2^{14}	4	0.05	4	0.14

[a]Includes BaseOTs for a new connection

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 [19] and a maliciously secure variant following [40][10]. 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.

---

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

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	$|\mathcal{X}|$	LAN setting			WAN setting		
			$n =$ 3	4	5	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

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

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.

Operation	$|\mathcal{X}|$	Semi-honest						Malicious					
		LAN setting			WAN setting			LAN setting			WAN setting		
		$n =$ 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
WitUpdate$_{Add}$	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
WitUpdate$_{Delete}$	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

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 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 [20, 28, 32], 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 ($\approx 10\,\text{ms}$ on our benchmark platform) which results in a further speedup, especially in the WAN-setting.

5   Applications

5.1   Credential Revocation in Distributed Credential Systems

As first application of MPC-based accumulators, we focus on distributed credential systems [31], and in particular, on the implementation in Sovrin [37]. In general, anonymous credentials provide a mechanism for making identity assertions while maintaining privacy, yet, in classical, non-distributed systems require

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	$|\mathcal{X}|$	Semi-honest		Malicious	
		Offline	Online	Offline	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, WitUpdate$_{\text{Delete}}$	2^{10}, 2^{14}	0.26	0.23	1.1	0.3
Add, WitUpdate$_{\text{Add}}$	2^{10}, 2^{14}	0	0.11	0	0.11

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 [2] 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 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 [43], Kales et al. [35] 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. [35] 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 noncolluding 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.

References

1. Abdolmaleki, B., Ramacher, S., Slamanig, D.: Lift-and-shift: obtaining simulation extractable subversion and updatable snarks generically. In: CCS, pp. 1987–2005. ACM (2020)

2. Acar, T., Chow, S.S.M., Nguyen, L.: Accumulators and U-prove revocation. In: Sadeghi, A.-R. (ed.) FC 2013. LNCS, vol. 7859, pp. 189–196. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39884-1_15

3. Baldimtsi, F., et al.: Accumulators with applications to anonymity-preserving revocation. In: EuroS&P, pp. 301–315. IEEE (2017)

4. Badimtsi, F., Canetti, R., Yakoubov, S.: Universally composable accumulators. In: Jarecki, S. (ed.) CT-RSA 2020. LNCS, vol. 12006, pp. 638–666. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-40186-3_27

5. Bar-Ilan, J., Beaver, D.: Non-cryptographic fault-tolerant computing in constant number of rounds of interaction. In: PODC, pp. 201–209. ACM (1989)

6. Barbulescu, R., Duquesne, S.: Updating key size estimations for pairings. J. Cryptol. **32**(4), 1298–1336 (2019)

7. Barreto, P.S.L.M., Lynn, B., Scott, M.: Constructing elliptic curves with prescribed embedding degrees. In: Cimato, S., Persiano, G., Galdi, C. (eds.) SCN 2002. LNCS, vol. 2576, pp. 257–267. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36413-7_19

8. Beaver, D.: Efficient multiparty protocols using circuit randomization. In: Feigenbaum, J. (ed.) CRYPTO 1991. LNCS, vol. 576, pp. 420–432. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-46766-1_34

9. Bellare, M., Fuchsbauer, G., Scafuro, A.: NIZKs with an untrusted CRS: security in the face of parameter subversion. In: Cheon, J.H., Takagi, T. (eds.) ASIACRYPT 2016. LNCS, vol. 10032, pp. 777–804. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53890-6_26

10. Ben-Sasson, E., Bentov, I., Horesh, Y., Riabzev, M.: Scalable zero knowledge with no trusted setup. In: Boldyreva, A., Micciancio, D. (eds.) CRYPTO 2019. LNCS, vol. 11694, pp. 701–732. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-26954-8_23

11. Benaloh, J., de Mare, M.: One-way accumulators: a decentralized alternative to digital signatures. In: Helleseth, T. (ed.) EUROCRYPT 1993. LNCS, vol. 765, pp. 274–285. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-48285-7_24

12. Boneh, D., Bünz, B., Fisch, B.: Batching techniques for accumulators with applications to IOPs and stateless blockchains. In: Boldyreva, A., Micciancio, D. (eds.) CRYPTO 2019. LNCS, vol. 11692, pp. 561–586. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-26948-7_20

13. Camenisch, J., Lysyanskaya, A.: Dynamic accumulators and application to efficient revocation of anonymous credentials. In: Yung, M. (ed.) CRYPTO 2002. LNCS, vol. 2442, pp. 61–76. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45708-9_5

14. Campanelli, M., Gennaro, R., Goldfeder, S., Nizzardo, L.: Zero-knowledge contingent payments revisited: attacks and payments for services. In: ACM CCS, pp. 229–243. ACM (2017)

15. Canetti, R.: Universally composable security: a new paradigm for cryptographic protocols. In: FOCS, pp. 136–145. IEEE (2001)

16. Chen, M., et al.: Multiparty generation of an RSA modulus. In: Micciancio, D., Ristenpart, T. (eds.) CRYPTO 2020. LNCS, vol. 12172, pp. 64–93. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-56877-1_3

17. Chen, M., et al.: Diogenes: lightweight scalable RSA modulus generation with a dishonest majority. IACR Cryptol. ePrint Arch. **2020**, 374 (2020)

18. Chida, K., et al.: Fast large-scale honest-majority MPC for malicious adversaries. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018. LNCS, vol. 10993, pp. 34–64. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96878-0_2

19. Cramer, R., Damgård, I., Maurer, U.: General secure multi-party computation from any linear secret-sharing scheme. In: Preneel, B. (ed.) EUROCRYPT 2000. LNCS, vol. 1807, pp. 316–334. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-45539-6_22

20. Dalskov, A., Orlandi, C., Keller, M., Shrishak, K., Shulman, H.: Securing DNSSEC keys via threshold ECDSA from generic MPC. In: Chen, L., Li, N., Liang, K., Schneider, S. (eds.) ESORICS 2020. LNCS, vol. 12309, pp. 654–673. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-59013-0_32

21. Damgård, I., Keller, M., Larraia, E., Pastro, V., Scholl, P., Smart, N.P.: Practical covertly secure MPC for dishonest majority – or: breaking the SPDZ limits. In: Crampton, J., Jajodia, S., Mayes, K. (eds.) ESORICS 2013. LNCS, vol. 8134, pp. 1–18. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40203-6_1

22. Damgård, I., Koprowski, M.: Practical threshold RSA signatures without a trusted dealer. In: Pfitzmann, B. (ed.) EUROCRYPT 2001. LNCS, vol. 2045, pp. 152–165. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44987-6_10

23. Damgård, I., Nielsen, J.B.: Universally composable efficient multiparty computation from threshold homomorphic encryption. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, pp. 247–264. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45146-4_15

24. Damgård, I., Pastro, V., Smart, N., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 643–662. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32009-5_38

25. Derler, D., Hanser, C., Slamanig, D.: Revisiting cryptographic accumulators, additional properties and relations to other primitives. In: Nyberg, K. (ed.) CT-RSA 2015. LNCS, vol. 9048, pp. 127–144. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-16715-2_7

26. Derler, D., Ramacher, S., Slamanig, D.: Post-quantum zero-knowledge proofs for accumulators with applications to ring signatures from symmetric-key primitives. In: Lange, T., Steinwandt, R. (eds.) PQCrypto 2018. LNCS, vol. 10786, pp. 419–440. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-79063-3_20

27. Dobson, S., Galbraith, S.D.: Trustless groups of unknown order with hyperelliptic curves. IACR ePrint **2020**, 196 (2020)

28. Doerner, J., Kondi, Y., Lee, E., Shelat, A.: Threshold ECDSA from ECDSA assumptions: the multiparty case. In: IEEE S&P, pp. 1051–1066. IEEE (2019)

29. Frederiksen, T.K., Lindell, Y., Osheter, V., Pinkas, B.: Fast distributed RSA key generation for semi-honest and malicious adversaries. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018. LNCS, vol. 10992, pp. 331–361. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96881-0_12

30. Galbraith, S.D., Paterson, K.G., Smart, N.P.: Pairings for cryptographers. Discret. Appl. Math. **156**(16), 3113–3121 (2008)

31. Garman, C., Green, M., Miers, I.: Decentralized anonymous credentials. In: NDSS. The Internet Society (2014)

32. Gennaro, R., Goldfeder, S.: Fast multiparty threshold ECDSA with fast trustless setup. In: ACM CCS, pp. 1179–1194. ACM (2018)

33. Groth, J., Kohlweiss, M., Maller, M., Meiklejohn, S., Miers, I.: Updatable and universal common reference strings with applications to zk-SNARKs. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018. LNCS, vol. 10993, pp. 698–728. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96878-0_24

34. Hamdy, S., Möller, B.: Security of cryptosystems based on class groups of imaginary quadratic orders. In: Okamoto, T. (ed.) ASIACRYPT 2000. LNCS, vol. 1976, pp. 234–247. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-44448-3_18

35. Kales, D., Omolola, O., Ramacher, S.: Revisiting user privacy for certificate transparency. In: EuroS&P, pp. 432–447. IEEE (2019)

36. Keller, M.: MP-SPDZ: a versatile framework for multi-party computation. In: CCS, pp. 1575–1590. ACM (2020)

37. Khovratovich, D., Law, J.: Sovrin: digitial signatures in the blockchain area (2016). https://sovrin.org/wp-content/uploads/AnonCred-RWC.pdf

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

39. Libert, B., Ling, S., Nguyen, K., Wang, H.: Zero-knowledge arguments for lattice-based accumulators: logarithmic-size ring signatures and group signatures without trapdoors. In: Fischlin, M., Coron, J.-S. (eds.) EUROCRYPT 2016. LNCS, vol. 9666, pp. 1–31. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49896-5_1

40. Lindell, Y., Nof, A.: A framework for constructing fast MPC over arithmetic circuits with malicious adversaries and an honest-majority. In: CCS, pp. 259–276. ACM (2017)

41. Lindell, Y., Nof, A.: Fast secure multiparty ECDSA with practical distributed key generation and applications to cryptocurrency custody. In: ACM CCS, pp. 1837–1854. ACM (2018)

42. Lipmaa, H.: Secure accumulators from Euclidean rings without trusted setup. In: Bao, F., Samarati, P., Zhou, J. (eds.) ACNS 2012. LNCS, vol. 7341, pp. 224–240. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31284-7_14

43. Lueks, W., Goldberg, I.: Sublinear scaling for multi-client private information retrieval. In: Böhme, R., Okamoto, T. (eds.) FC 2015. LNCS, vol. 8975, pp. 168–186. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-47854-7_10

44. Merkle, R.C.: A certified digital signature. In: Brassard, G. (ed.) CRYPTO 1989. LNCS, vol. 435, pp. 218–238. Springer, New York (1990). https://doi.org/10.1007/0-387-34805-0_21

45. Miers, I., Garman, C., Green, M., Rubin, A.D.: Zerocoin: anonymous distributed e-cash from bitcoin. In: IEEE S&P, pp. 397–411. IEEE (2013)

46. Nguyen, L.: Accumulators from bilinear pairings and applications. In: Menezes, A. (ed.) CT-RSA 2005. LNCS, vol. 3376, pp. 275–292. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-30574-3_19

47. Sander, T.: Efficient accumulators without trapdoor extended abstract. In: Varad-harajan, V., Mu, Y. (eds.) ICICS 1999. LNCS, vol. 1726, pp. 252–262. Springer, Heidelberg (1999). https://doi.org/10.1007/978-3-540-47942-0_21
48. Shamir, A.: How to share a secret. Commun. ACM **22**(11), 612–613 (1979)
49. Smart, N.P., Talibi Alaoui, Y.: Distributing any elliptic curve based protocol. In: Albrecht, M. (ed.) IMACC 2019. LNCS, vol. 11929, pp. 342–366. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-35199-1_17
50. Wagh, S., Gupta, D., Chandran, N.: SecureNN: 3-party secure computation for neural network training. PoPETs **2019**(3), 26–49 (2019)