Powers of Tau in Asynchrony

Sourav Das[*], Zhuolun Xiang[†], and Ling Ren[*]
[*]University of Illinois at Urbana-Champaign, [†]Aptos
souravd2@illinois.edu, xiangzhuolun@gmail.com, renling@illinois.edu

***Abstract*—The q-Strong Diffie-Hellman (q-SDH) parameters are foundational to efficient constructions of many cryptographic primitives such as zero-knowledge succinct non-interactive argument of knowledge, polynomial/vector commitments, verifiable secret sharing, and randomness beacon. The only existing method to generate these parameters securely is highly sequential, requires strong network synchrony assumptions, and has very high communication and computation cost. For example, to generate parameters for any given q, each party incurs a communication cost of $\Omega(nq)$ and requires $\Omega(n)$ rounds. Here n is the number of parties in the secure multiparty computation protocol. Since q is typically large, i.e., on the order of billions, the cost is highly prohibitive.**

In this paper, we present TAURON[*], a distributed protocol to generate q-SDH parameters in an asynchronous network. In a network of n parties, TAURON tolerates up to one-third of malicious parties. Each party incurs a communication cost of $O(q + n^2 \log q)$ and the protocol finishes in $O(\log q + \log n)$ expected rounds. We provide a rigorous security analysis of our protocol. We implement TAURON and evaluate it with up to 128 geographically distributed parties. Our evaluation illustrates that TAURON is highly scalable and results in a 2-6× better runtime and 4-13× better per-party bandwidth usage.

I. INTRODUCTION

The q-Strong Diffie Helmann assumption, or q-SDH assumption for short, refers to the cryptographic intractability problem of computing a group element of the form $(\tau+i)^{-1}\mathfrak{g}$ for any $i \neq -\tau \in \mathbb{F}$, given $\{\mathfrak{g}, \tau\mathfrak{g}, \tau^2\mathfrak{g}, \ldots, \tau^q\mathfrak{g}\}$. Here, $\mathfrak{g}$ is a generator of a group $\mathbb{G}$ (typically an elliptic curve group), and τ is a random field element from the scalar field $\mathbb{F}$ of $\mathbb{G}$. The vector $\{\mathfrak{g}, \tau\mathfrak{g}, \tau^2\mathfrak{g}, \ldots, \tau^q\mathfrak{g}\}$ is referred to as the q-SDH parameters, also known as the *powers of* τ.

The q-SDH assumption is used in many applications. Boneh and Boyen [11] introduced the q-SDH assumption to design a short signature scheme that does not rely on a random oracle. Kate, Zaverucha, and Goldberg used the q-SDH assumption to design a constant size polynomial commitment scheme with constant size opening proofs [32]. This polynomial commitment scheme and many of its follow-up schemes have been used extensively in designing a variety of Succinct Non-interactive Argument of Knowledge (SNARK) protocol [35], [44], [14], [26]. The q-SDH parameters have also been used in designing efficient verifiable secret sharing [5], cryptographic accumulators [31], vector commitments [43], distributed randomness beacon [9], etc.

For many applications, the degree q is typically very large. For example, in SNARKs, q is proportional to the size of the SNARK circuit, measured in the number of multiplication gates. Thus, the overall size of the circuit ranges from a few million to hundreds of millions [35]. In vector commitment schemes, q is the size of the committed vector, which can be very large [43].

For the q-SDH assumption to hold, it is critical to keep τ hidden from an adversary $\mathcal{A}$. Otherwise, $\mathcal{A}$ can trivially break the q-SDH assumption and the security of the applications using it. One mechanism to generate such q-SDH parameters is to rely on a trusted third party. Specifically, a trusted party locally samples a random τ, computes and publishes $\{\mathfrak{g}, \tau\mathfrak{g}, \tau^2\mathfrak{g}, \ldots, \tau^q\mathfrak{g}\}$, and then deletes τ. However, this approach introduces a central trusted party and a single point of failure, which is undesirable.

This paper studies the problem of secure and robust generation of q-SDH parameters in a distributed manner.

Existing works. Existing protocols [8], [33], [12], [39] for generating q-SDH parameters follow the blueprint of [8]. These protocols assume a synchronous network and a broadcast channel, proceed in a round-robin manner. Briefly, in these protocols, parties take turns to update existing q-SDH parameters with a randomly chosen value and broadcast the updated parameters to all other parties. Once every party updates the q-SDH parameters with its private randomness, the final output is obtained. Intuitively, as long as one honest party updates the parameter, the trapdoor τ remains hidden from the adversary. As we elaborate next, this approach has major limitations in robustness and efficiency.

Synchrony assumption. The reliance on network synchrony significantly reduces the above protocol's fault tolerance under asynchrony. Briefly, if a party experiences temporary network asynchrony, then other parties will *time out*, skip that party and move on. The fault tolerance is thus reduced. In the network remains asynchronous for a long time, all honest parties can be skipped, and the trapdoor τ will be fully controlled and known to the adversary, as only adversarial parties will contribute randomness to the final output.

Inefficiency. Existing protocols are inefficient and inherently sequential. They require running $O(n)$ sequential Byzantine broadcasts, once by each party, where the party needs to send $O(q)$ group elements through a costly broadcast channel. Each party must wait for all previous parties to update the q-SDH parameters. Moreover, each party also needs to verify updates by all previous parties before applying its own update.

Insecure deployments. To mitigate the inefficiencies, actual deployments of these protocols often cut corners on robustness

[*]It is a merge of TAU (τ) and SAURON who forged the *Rings of Power*.

Table I: Comparison of protocols for generating secure q-SDH parameters. Here n is the total number of parties and q is the highest degree of the q-SDH parameters. All of these protocols (including ours) generates updatable parameters in the sense of [35]. We provide a detailed breakdown of our cost in Table III. We measure the computation cost as the number of elliptic curve group multiplications. For existing works, we plug in the state-of-the-art synchronous honest-majority broadcast protocol [38], [37] to measure their communicaiton cost. The broadcast protocol has communication cost $O(nq+n^2 \log n)$ and latency $O(n)$ for broadcasting $O(q)$ field elements. Thus, to favor existing protocols, we report their communication cost and round complexity as $\Omega(nq)$ and $\Omega(n)$, respectively.

	Network Model	Fault Tolerance	Communication Cost (Per Party)	Computation Cost (Per Party)	Total Round Complexity	Setup Assumption
[8], [33], [12], [39]	sync.	$n/2$	$\Omega(nq)$	$O(nq)^‡$	$\Omega(n)$	CRS & PKI†
TAURON (this work)	async.	$n/3$	$O(q+n^2 \log q)$	$O(q \log n + n^2 \log q)^‡$	$O(\log n + \log q)$	CRS & PKI

† Existing protocols require PKI to implement broadcast channel.

‡ Both state-of-the-art synchronous protocols [39] and TAURON require parties to perform $O(n)$ and $O(n \log q)$ bilinear pairing, respectively.

or security. For example, one deployed version relies on a single party to act as a broadcast channel [3]. Moreover, in the deployed version, parties skip verifying other parties' updates during the protocol and only verify the entire protocol transcript at the end. A consequence is that a single malicious party can now make the protocol produce invalid parameters by performing an invalid update. If that happens, the only recourse is to restart the entire protocol.

Furthermore, even after cutting these corners on security and robustness, existing protocols perform poorly. For example, according to [25], to generate q-SDH parameters for $q = 2^{28}$, each party needs to perform 24 hours of computation. Hence, with n parties, the protocol would run for n days.

One might believe that a slow and costly q-SDH parameter generation protocol is acceptable because it needs to be run only once and can then be reused across all applications by all organizations. This is not always the case. One reason is that parameters generated by a set of parties may not be trusted by another set of parties. For example, two startups Aztec [2] and Semaphore [3] repeated the powers-of-tau ceremony for the same elliptic curve BN254. Moreover, q-SDH parameters are elliptic curve groups specific and hence must be generated from scratch if a system decides to adopt a new curve. For example, Etherum [23] plans to run another powers-of-tau ceremony for the BLS12381 elliptic curve, although they have already run a powers-of-tau for the BN254 curve before [2].

Our contributions. In this paper, we present TAURON, the first protocol for distributed generation of q-SDH parameters in asynchronous networks. TAURON can tolerate up to t malicious parties out of $n \geq 3t+1$ parties. We provide a detailed summary of the properties of TAURON in Table I. The protocol finishes in $O(\log q + \log n)$ rounds. The per-party communication cost is $O(q + n^2 \log q)$ group elements, which improves upon the communication cost of prior best synchronous protocols by a factor $O(n)$. We also improve the per-party computation cost to $O(q \log n)$ group multiplications and $O(n \log q)$ pairings. Unlike existing protocols, TAURON is also *responsive*, i.e., it makes progress at a rate of actual network speed.

Evaluation. We implement TAURON in `python` with `rust` for cryptographic operations. Our implementation is *single-threaded*, supports the `bls12381` elliptic curve, and is publicly available†. We evaluate TAURON with a network of up to 128 geographically distributed parties. We also compare TAURON with the state-of-the-art synchronous protocol. Our evaluation illustrates that TAURON gives 2-6× faster runtime and 4-13× better per-party bandwidth usage. For example, with $n = 128$ and $q = 2^{14}$, TAURON takes 514.51 seconds to generate the q-SDH parameters, whereas existing protocol takes at least 1805.20 seconds (3.5× faster). Similarly, in the same experiment, each party in TAURON needs to send 118.17 Megabytes of data whereas, compared to 1536 Megabytes of data (13× better) in the existing protocol.

Paper organization. We introduce notations, system model, and present an overview of our protocol in §II. In §III, we discuss the required preliminaries. We provide a detailed description of TAURON in §IV and §V. We analyze the correctness and security of TAURON in §VI. We present implementation and evaluation details in §VII. We discuss related works in §VIII and conclude with a discussion in §IX.

Table II: Notations used in the paper

Notation	Description
n	Total number of parties
t	Maximum number of malicious parties
κ	Security parameter
$\mathbb{G}$	Group of order p with hard Discrete Logarithm
$\mathbb{F}$	Scalar field of group $\mathbb{G}$
$\mathfrak{g}, \mathfrak{h}$	Random and independent generators of $\mathbb{G}$
q	Maximum degree of the q-SDH parameters
τ	q-SDH parameter trapdoor
$[\![z]\!]$	$(n, t+1)$ Shamir secret sharing of z
$[\![z]\!]^{2t}$	$(n, 2t+1)$ Shamir secret sharing of z
$[\![z]\!]_i, [\![z]\!]_i^{2t}$	Shares received by party i
$[\![z]\!]\mathfrak{g}$	The set $\{[\![z]\!]_1\mathfrak{g}, [\![z]\!]_2\mathfrak{g}, \ldots, [\![z]\!]_n\mathfrak{g}\}$
$[a, b]$	The set $\{a, a+1, a+2, \ldots, b\}$
$[a]$	The set $\{1, 2, \ldots, a\}$

II. System Model and Overview

A. Notations and System Model

We use κ to denote the security parameter. For example, when we use a collision-resistant hash function, κ denotes the size of the output of the hash function. We use $|S|$ to denote the

†https://github.com/sourav1547/qsdh-py

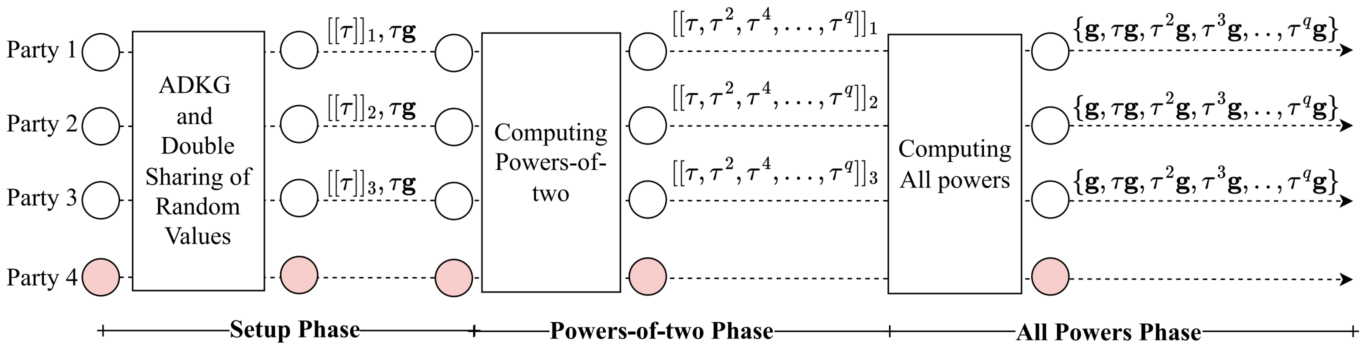
Party 1
Party 2
Party 3
Party 4
ADKG and Double Sharing of Random Values
$[[\tau]]_1, \tau\mathbf{g}$
$[[\tau]]_2, \tau\mathbf{g}$
$[[\tau]]_3, \tau\mathbf{g}$
Computing Powers-of-two
$[[\tau, \tau^2, \tau^4, \ldots, \tau^q]]_1$
$[[\tau, \tau^2, \tau^4, \ldots, \tau^q]]_2$
$[[\tau, \tau^2, \tau^4, \ldots, \tau^q]]_3$
Computing All powers
$\{\mathbf{g}, \tau\mathbf{g}, \tau^2\mathbf{g}, \tau^3\mathbf{g}, .., \tau^q\mathbf{g}\}$
$\{\mathbf{g}, \tau\mathbf{g}, \tau^2\mathbf{g}, \tau^3\mathbf{g}, .., \tau^q\mathbf{g}\}$
$\{\mathbf{g}, \tau\mathbf{g}, \tau^2\mathbf{g}, \tau^3\mathbf{g}, .., \tau^q\mathbf{g}\}$
Setup Phase
Powers-of-two Phase
All Powers Phase

Figure 1: Overview of our protocol in a network of 4 parties where party 4 is malicious.

Functionality $\mathcal{F}_{\mathsf{qSDH}}$

- Let $\mathbb{G}$ be a elliptic curve group with scalar field $\mathbb{F}$. Let $\mathfrak{g}$ be a uniformly random generator of $\mathbb{G}$.
- Sample a uniformly random element $\tau \in \mathbb{F}$. Compute $\{\mathfrak{g}, \tau\mathfrak{g}, \tau^2\mathfrak{g}, \ldots, \tau^q\mathfrak{g}\}$ and send it to everyone.

Figure 2: The functionality for generating q-SDH parameters

size of a set S. Let $\mathbb{G}$ be a group of prime order p with scalar field $\mathbb{F}$, in which q-SDH problem is believed to be hard. We will be using the additive notation for elliptic curve operations. For any $x \in \mathbb{F}$ and any group element $\mathfrak{g} \in \mathbb{G}$, we use $x\mathfrak{g}$ to denote the group operations repeated x times. For any integer a, we use $[a]$ to denote the ordered set $\{1, 2, \ldots, a\}$. Also, for two integers a and b where $a < b$, we use $[a, b]$ to denote the ordered set $\{a, a+1, \ldots, b\}$.

For any $x \in \mathbb{F}$, we use $[\![x]\!]$ to denote the $(n, t+1)$ secret sharing of x, i.e., x is secret shared using a polynomial of degree t. Also, we use $[\![x]\!]_i$ to denote the share held by party i, and $[\![x]\!]_0$ to denote x. For a vector $\boldsymbol{x}$, we use $[\![\boldsymbol{x}]\!]$ to denote the element-wise secret sharing of $\boldsymbol{x}$. Similarly, we use $[\![\boldsymbol{x}]\!]_i$ to denote the element-wise share of $\boldsymbol{x}$ held by party i.

Threat model and network assumption. We consider a network of n parties where every pair of parties are connected via a pairwise authenticated channel. We consider the presence of a probabilistic polynomial time malicious adversary $\mathcal{A}$ that can corrupt up to t out of the $n \geq 3t+1$ parties in the network. We assume the network is asynchronous, i.e., $\mathcal{A}$ can arbitrarily delay any message but must eventually deliver all messages sent between honest parties.

State-of-the-art solutions to two building blocks of our protocol, specifically, asynchronous distributed key generation and random double sharing, assume the existence of a public key infrastructure (PKI) for efficiency [21]. Both building blocks can be instantiated without PKI at higher costs [34], so the PKI assumption can be removed at the cost of lowering the efficiency of the protocol if the application calls for it.

B. Problem Definition

The q-Strong Diffie-Helmann (q-SDH for short) refers to the cryptographic intractability problem defined below.

Definition 1 (q-SDH Hardness)**.** Let κ be the security parameter. Let $\mathbb{G}$ and $\mathbb{F}$ be a group and field of size exponential in κ, respectively. Let $\mathfrak{g} \leftarrow \mathsf{Gen}(1^\kappa)$ be a uniform random generator of group $\mathbb{G}$ and $\tau \in \mathbb{F}$ be a trapdoor. For any given q, which is polynomially bounded in κ, given the vector $\{\mathfrak{g}, \tau\mathfrak{g}, \tau^2\mathfrak{g}, \ldots, \tau^q\mathfrak{g}\}$, q-SDH is assumed to be hard with respect to this vector if the following probability is negligible for all PPT adversary $\mathcal{A}$

$$\Pr[(c, (\tau+c)^{-1}\mathfrak{g}), c \in \mathbb{F} \setminus \{-\tau\} : \mathcal{A}(\mathfrak{g}, \tau\mathfrak{g}, \tau^2\mathfrak{g}, \ldots, \tau^q\mathfrak{g}); \mathfrak{g} \leftarrow \mathsf{Gen}(1^\kappa), \tau \xleftarrow{\$} \mathbb{F}] \quad (1)$$

The vector $\{\mathfrak{g}, \tau\mathfrak{g}, \tau^2\mathfrak{g}, \ldots, \tau^q\mathfrak{g}\}$ is referred to as the q-SDH parameters.

The goal of this paper is to design a distributed protocol to implement $\mathcal{F}_{\mathsf{qSDH}}$, i.e., to generate $\{\mathfrak{g}, \tau\mathfrak{g}, \tau^2\mathfrak{g}, \ldots, \tau^q\mathfrak{g}\}$ for $\tau \in \mathbb{F}$ at all honest parties given a uniformly random generator $\mathfrak{g} \in \mathbb{G}$ as the common random string (CRS). The protocol is called t-secure if the following *Correctness* and *Secrecy* properties hold in the presence of an adversary $\mathcal{A}$ that corrupts up to t parties.

- **Correctness.** If all honest parties start the protocol, every honest party will eventually terminate and output an identical vector $\{\mathfrak{g}, \tau\mathfrak{g}, \tau^2\mathfrak{g}, \ldots, \tau^q\mathfrak{g}\}$ for some $\tau \in \mathbb{F}$.
- **Security.** The protocol generates hard q-SDH parameters as per the definition 1.

We will prove security assuming the hardness of q-SDH for parameters generated by the ideal functionality $\mathcal{F}_{\mathsf{qSDH}}$. Specifically, we will prove that if $\mathcal{A}$ can break the q-SDH assumption with respect to the parameters generated by our protocol, we can design a reduction adversary, $\mathcal{A}_{\mathsf{qSDH}}$, that can use $\mathcal{A}$ to break the q-SDH assumption for parameters generated by $\mathcal{F}_{\mathsf{qSDH}}$.

C. Overview of Our Protocol

As described in §I, in existing protocols [8], [33], [12], [39], parties take turns to update the existing q-SDH parameters with their private randomness and then broadcast the result. We do not see a way to adapt these protocols to asynchrony as it is impossible to implement a broadcast channel in asynchrony. Intuitively, it is impossible to distinguish a malicious broadcaster from an honest broadcaster with a slow network.

One approach to address this issue is to use a generic asynchronous secure multiparty computation (MPC) protocol for a circuit $\mathcal{C}$ that outputs q-SDH parameters. However, this approach is very costly, primarily for the following reasons. First, the circuit $\mathcal{C}$ consists of $O(q)$ multiplication gates; Second, the circuit $\mathcal{C}$ needs to resolve the discrepancy between the scalar and the base field of the underlying elliptic curve. We elaborate on this in §VIII.

We give a new approach to distributed q-SDH parameter generation in this paper. Our first main idea is to securely compute $\tau\mathfrak{g}$ for a uniformly random $\tau \in \mathbb{F}$. Moreover, we want to have τ secret shared among the parties using a $(n, t+1)$ Shamir secret sharing. We will then find a way to use the public value $\tau\mathfrak{g}$ and the secret shares of τ to efficiently compute the remaining powers of τ. We will first describe a naïve method that is incomplete and inefficient but demonstrates a core idea in our final protocol.

Naïve approach. Let $[\![\tau]\!]_i$ be the secret share of party i. The protocol proceeds in rounds, where in the k-th round, parties generate $\tau^{k+1}\mathfrak{g}$ using $\tau^k\mathfrak{g}$. Hence, at the end of round k, parties generate the parameters $\{\mathfrak{g}, \tau\mathfrak{g}, \tau^2\mathfrak{g}, \dots, \tau^{k+1}\mathfrak{g}\}$. At the start of k-th round, each party i locally computes $[\![\tau]\!]_i\tau^k\mathfrak{g}$ and then multicasts $[\![\tau]\!]_i\tau^k\mathfrak{g}$ to all parties. Also, party i upon receiving $[\![\tau]\!]_j\tau^k\mathfrak{g}$ from $t+1$ distinct parties, locally computes $\tau^{k+1}\mathfrak{g}$ as

$$\tau^{k+1}\mathfrak{g} = \sum_{i \in T} \lambda_i [\![\tau]\!]_i \tau^k \mathfrak{g} \qquad (2)$$

where λ_i is the appropriate Lagrange coefficient.

This approach, however, has two major issues. First, the protocol is not robust, as a malicious party can send different and inconsistent messages to different parties, violating Correctness. Second, the protocol is very inefficient. It requires $O(q)$ rounds of interaction and $O(nq)$ per-party communication. Since q can be quite large in practice, say millions, this naïve approach is impractical.

Ensuring Correctness. Addressing the Correctness issue is relatively standard. Specifically, we need a mechanism for honest parties to validate the messages they receive from other parties. To achieve this, at the start of the protocol, we require that parties additionally hold $[\![\tau]\!]\mathfrak{g}$, i.e., the vector $[[\![\tau]\!]_1\mathfrak{g}, [\![\tau]\!]_2\mathfrak{g}, \dots, [\![\tau]\!]_n\mathfrak{g}]$. Then, each party i, when sending $[\![\tau]\!]_i\tau^k\mathfrak{g}$, attaches a non-interactive zero knowledge (NIZK) proof π_i proving that $[\![\tau]\!]_i\tau^k\mathfrak{g}$ is correctly computed from $\tau^k\mathfrak{g}$ and $[\![\tau]\!]_i\mathfrak{g}$.

For this plan to work, the next natural question is how we establish the initial condition that parties agree on group elements $\tau\mathfrak{g}$ and $[\![\tau]\!]\mathfrak{g}$, and also hold their individual secret share $[\![\tau]\!]_i$. Our second observation is that the above initial condition exactly matches the output of an Asynchronous Distributed Key Generation (ADKG) protocol. An ADKG protocol generates a uniformly random secret key $\tau \in \mathbb{F}$ where each party i receives its share $[\![\tau]\!]_i$ of the secret key, the public key $\tau\mathfrak{g}$ and the threshold public keys $[\![\tau]\!]\mathfrak{g}$.

Reducing round complexity. One approach to reducing the round complexity is once again generic secure multiparty computation (MPC). Let $\mathcal{C}$ be the arithmetic circuit that outputs all powers of τ in the exponent. Using repeated squaring, the depth of $\mathcal{C}$ will be $O(\log q)$. However, as we mentioned before, generic MPC is very costly.

We give an efficient method to reduce the round complexity to $O(\log q)$ by combining the idea of the naïve approach and the MPC approach. At a high level, parties first use a customized MPC protocol to compute $\tau^{2^k}\mathfrak{g}$ and secret shares of $[\![\tau^{2^k}]\!]$ for each $k \in [\log q]$ (referred to as *powers-of-two*). For each $k \in [\log q]$, parties additionally output $[\![\tau^{2^k}]\!]\mathfrak{g}$. Parties then use these values to efficiently compute the remaining powers using only $O(\log q)$ rounds of interaction.

Computing powers-of-two. Let $\mathcal{F}_{\mathsf{sq}}$ be a secure MPC functionality for squaring defined as follows. $\mathcal{F}_{\mathsf{sq}}$ takes as input the secret sharing of a, i.e., each party i inputs $[\![a]\!]_i$, and a publicly available list of group elements $[\![a]\!]\mathfrak{g}$. $\mathcal{F}_{\mathsf{sq}}$ then outputs to party i secret sharing of a^2, i.e., $[\![a^2]\!]_i$, and additionally outputs $[\![a^2]\!]\mathfrak{g}$ to all parties. We can then invoke $\mathcal{F}_{\mathsf{sq}}$ sequentially $\log q$ times to compute the powers-of-two values.

Computing remaining powers. We generate the remaining powers of τ using the following ideas. We observe that given $\tau^\alpha\mathfrak{g}$, secret shares of τ^β (i.e., $[\![\tau^\beta]\!]$), and public values $[\![\tau^\beta]\!]\mathfrak{g}$, parties can compute $\tau^{\alpha+\beta}\mathfrak{g}$, using a generalization of our naïve approach. Now consider any $a \in [q]$, we can write a as a sum of a subset of elements in $\{1, 2, 2^2, 2^3, \dots, 2^{\log(q)-1}\}$ according to its binary representation. Let S_a be the subset. Then, $\tau^a\mathfrak{g} = \tau^{\sum_{k \in S_a} k}\mathfrak{g}$. It is easy to see that, using the idea of multiplication in the exponent, parties can compute the subset sum using $|S_a| \le O(\log q)$ multiplications. Furthermore, parties can compute $\tau^a\mathfrak{g}$ for every $a \in [q]$ in parallel.

Further optimizations. The method above incurs a per-party communication cost of $O(nq \log q)$ and a per-party computation cost of $O(nq \log q)$ group multiplications. In §V, we discuss how we reduce the communication cost to $O(q + n^2 \log q)$ and computation cost to $O(q \log n)$ group multiplications and $O(n \log q)$ bilinear pairings using *memoization* and *batching*.

III. Preliminaries

A. Threshold Secret Sharing

A (n, k) threshold secret sharing scheme allows a secret $s \in \mathbb{F}$ to be shared among n parties such that any k of them can come together to recover the original secret, but any subset of $k-1$ shares does not reveal any information about the secret [41], [10]. We use the common Shamir secret sharing [41] scheme, where the secret is embedded in a random degree $k-1$ polynomial in the field $\mathbb{F}$. Specifically, to share a secret $s \in \mathbb{F}$, a polynomial $p(\cdot)$ of degree $k-1$ is chosen such that $s = p(0)$. The remaining coefficients of $p(\cdot)$, $p_1, p_2, \cdots, p_{k-1}$ are chosen uniformly randomly from $\mathbb{F}$. The resulting polynomial $p(x)$ is defined as:

$$p(x) = s + p_1x + p_2x^2 + \cdots + p_{k-1}x^{k-1}$$

Each party is then given a single evaluation of $p(\cdot)$. In particular, the i^{th} party is given $p(i)$, i.e., the polynomial evaluated at i. Observe that given k points on the polynomial $p(\cdot)$, one can efficiently reconstruct the polynomial using Lagrange Interpolation. Also, s is information-theoretically hidden from an adversary that knows any subset of $k-1$ or fewer evaluation points on the polynomial other than $p(0)$.

B. Asynchronous Distributed Key Generation

Our protocol uses asynchronous distributed key generation (ADKG) functionality $\mathcal{F}_{\mathsf{ADKG}}$, defined in Figure 10. Concretely, we run ADKG protocol from [21]. At the end of the ADKG protocol, parties output a $(n, t+1)$ Shamir secret sharing of a random value τ (i.e., $[\![\tau]\!]$), the ADKG public key $\tau\mathfrak{g}$, and threshold public keys of every party, i.e., $[\![\tau]\!]\mathfrak{g}$. The ADKG protocol of [21] assumes the hardness of Discrete Logarithm, has per-party communication cost of $O(n^2)$, and terminates in expected $O(\log n)$ rounds.

C. Asynchronous Double Sharing of Random Values

Our realization of $\mathcal{F}_{\mathsf{sq}}$ uses double sharing of uniformly random field elements [18]. Specifically, we will use secret shares of a random field element z with both degree t and degree $2t$ polynomials, denoted as $[\![z]\!]$ and $[\![z]\!]^{2t}$, respectively. Here z is uniformly random and independent of τ.

Looking ahead, each invocation of $\mathcal{F}_{\mathsf{sq}}$ will use double sharing of one random field element. Since we invoke $\mathcal{F}_{\mathsf{sq}}$ $\log q$ times, we need $\log q$ double sharing of independent random field elements. Our realization of $\mathcal{F}_{\mathsf{sq}}$ additionally require publicly available $[\![z]\!]\mathfrak{g}$ and $[\![z]\!]^{2t}\mathfrak{g}$ to publicly reconstruct $[\![\tau^{2^k}]\!]\mathfrak{g}$ for each $k \in [\log q]$.

To generate double sharing of $\log q$ random elements along with their corresponding public keys, we use the functionality $\mathcal{F}_{\mathsf{Dou}}$ defined in Figure 8. We use the random double sharing protocol of [21, §6.1] but make minor modifications to facilitate a simulation-based security proof (described in Appendix B). Our modifications maintain their $O(n^2)$ per-party communication cost and $O(\log n)$ round complexity.

D. Equality of Discrete Logarithm

Our protocol requires parties to produce zero-knowledge proofs about the equality of discrete logarithms for a tuple of publicly known values. In particular, given a group $\mathbb{G}$ with scalar field $\mathbb{F}$ of prime order p, two elements $\mathfrak{g}, \mathfrak{h} \in \mathbb{G}$ with unknown discrete logarithm relations and a tuple $(\mathfrak{g}, \mathfrak{a}, \mathfrak{h}, \mathfrak{b}) \in \mathbb{G}^4$, a prover $\mathcal{P}$ wants to prove to a probabilistic polynomial time verifier $\mathcal{V}$, in zero-knowledge, the knowledge of a witness $\alpha \in \mathbb{F}$ such that $\mathfrak{a} = \alpha\mathfrak{g}$ and $\mathfrak{b} = \alpha\mathfrak{h}$. We present the protocol in Appendix A.

We will use the Chaum-Pedersen "Σ-protocol" [13], which assumes the hardness of the Discrete Logarithm in $\mathbb{G}$. This protocol guarantees completeness, knowledge soundness, and zero-knowledge. The knowledge soundness implies that if $\mathcal{P}$ convinces the $\mathcal{V}$ with non-negligible probability, there exists an efficient (polynomial time) extractor that can extract α from $\mathcal{P}$ non-negligible probability.

The Chaum-Pedersen protocol can be made non-interactive in the Random Oracle model using the Fiat-Shamir heuristic [24]. We use the non-interactive variant of the protocol. For

Algorithm 1 TAURON protocol for party i

```
INPUT: $\mathfrak{g}, sk_i, \{pk_j\}$ for each $j \in [n]$
OUTPUT: $\{\mathfrak{g}, \tau\mathfrak{g}, \tau^2\mathfrak{g}, \ldots, \tau^q\mathfrak{g}\}$

SETUP PHASE:
    // Run the ADKG and the Double sharing protocol
11: Let $\tau\mathfrak{g}, [\![\tau]\!]\mathfrak{g}, [\![\tau]\!]_i \leftarrow \Pi_{\mathsf{ADKG}}()$
12: Let $\{z_k\mathfrak{g}, [\![z_k]\!]\mathfrak{g}, [\![z_k]\!]^{2t}\mathfrak{g}, [\![z_k]\!]_i\}_{\forall k \in [\log q]} \leftarrow \Pi_{\mathsf{Dou}}(\log q)$

POWERS-OF-TWO PHASE:
21: for each $k \in [\log q]$ do
22:     // Use double sharing of $z_k$ to run $\Pi_{\mathsf{sq}}$
23:     Let $[\![\tau^{2^k}]\!]; [\![\tau^{2^k}]\!]\mathfrak{g} = \Pi_{\mathsf{sq}}([\![\tau^{2^{k-1}}]\!]; [\![\tau^{2^{k-1}}]\!]\mathfrak{g})$
24:     Compute $\tau^{2^k}\mathfrak{g}$ by interpolating $[\![\tau^{2^k}]\!]\mathfrak{g}$ and output $\tau^{2^k}\mathfrak{g}$

ALL POWERS PHASE:
31: Compute all remaining powers as
        $\{\tau^a\mathfrak{g}\}_{\forall a \in [q]} := \Pi_{\mathsf{all}}(\{[\![\tau^{2^k}]\!]; \tau^{2^k}\mathfrak{g}, [\![\tau^{2^k}]\!]\mathfrak{g}\}_{\forall k \in [\log q]})$
```

any given tuple $(\mathfrak{g}, \mathfrak{a}, \mathfrak{h}, \mathfrak{b}) \in \mathbb{G}^4$ where $\mathfrak{a} = \alpha\mathfrak{g}$ and $\mathfrak{b} = \alpha\mathfrak{h}$, $\mathcal{P}$ uses $\mathsf{dleq.Prove}(\alpha, \mathfrak{g}, \mathfrak{a}, \mathfrak{h}, \mathfrak{b})$ to generate the non-interactive zero-knowledge proof π. The proof π is $O(\kappa)$ bits long. Given a proof π and $(\mathfrak{g}, \mathfrak{a}, \mathfrak{h}, \mathfrak{b})$, $\mathcal{V}$ uses the $\mathsf{dleq.Verify}(\pi, \mathfrak{g}, \mathfrak{a}, \mathfrak{h}, \mathfrak{b})$ to verify the proof.

IV. GENERATING POWERS-OF-TWO

$\mathcal{F}_{\mathsf{sq}}$ is the secure multiparty computation (MPC) functionality for squaring that takes as input $[\![a]\!]$, i.e., $(n, t+1)$ secret shares of a field element $a \in \mathbb{F}$ and outputs $[\![a^2]\!]_i$ to party i. $\mathcal{F}_{\mathsf{sq}}$ additionally takes the publicly available $[\![a]\!]\mathfrak{g}$ as input and outputs threshold public keys of a^2, i.e., $[\![a^2]\!]\mathfrak{g}$. Formally, we write the functionality $\mathcal{F}_{\mathsf{sq}}$ as:

$$[\![a^2]\!]; [\![a^2]\!]\mathfrak{g} = \mathcal{F}_{\mathsf{sq}}\left([\![a]\!]; [\![a]\!]\mathfrak{g}\right) \tag{3}$$

Our protocol uses $\mathcal{F}_{\mathsf{sq}}$ to compute $\tau^{2^k}\mathfrak{g}$ for each $k \in [\log q]$. In this process, parties also receive secret shares of τ^{2^k}, i.e., $[\![\tau^{2^k}]\!]$, and $[\![\tau^{2^k}]\!]\mathfrak{g}$, which they later use to compute $\tau^\alpha\mathfrak{g}$ for any arbitrary $\alpha \in [q]$.

A. Design of $\mathcal{F}_{\mathsf{sq}}$

We next describe our protocol Π_{sq} for realizing $\mathcal{F}_{\mathsf{sq}}$. Its pseudocode is given in Algorithm 2. Π_{sq} could be designed using various techniques, such as multiplication triples [6], degree reduction [7], and random double sharing [19]. We adopt the random double sharing approach.

Π_{sq} assumes that parties hold double shares of a uniform random $z \in \mathbb{F}$ using $(n, t+1)$ and $(n, 2t+1)$ Shamir secret sharing, denoted as $[\![z]\!]_i$ and $[\![z]\!]_i^{2t}$, respectively. Also, Π_{sq} assumes that $[\![z]\!]\mathfrak{g}$ and $[\![z]\!]^{2t}\mathfrak{g}$ are public.

Each party i, locally multiplies its shares of a to get $[\![a]\!]_i[\![a]\!]_i$. Parties then publicly reconstruct the $a^2 + z$. In particular, each party i locally computes the non-interactive zero-knowledge (NIZK) proof π_i of equality of discrete logarithm (dleq) between $\{\mathfrak{g}, [\![a]\!]_i\mathfrak{g}, [\![a]\!]_i\mathfrak{g}, [\![a]\!]_i[\![a]\!]_i\mathfrak{g}\}$, i.e.,

$$\pi_i = \mathsf{dleq.Prove}([\![a]\!]_i, \mathfrak{g}, [\![a]\!]_i\mathfrak{g}, [\![a]\!]_i\mathfrak{g}, [\![a]\!]_i[\![a]\!]_i\mathfrak{g})$$

Algorithm 2 Π_{sq} protocol at party i

```
INPUT: $[\![a]\!]_i$ and $[\![a]\!]\mathfrak{g}$
SETUP: $[\![z]\!]_i, [\![z]\!]_i^{2t}, [\![z]\!]\mathfrak{g}, [\![z]\!]^{2t}\mathfrak{g}$
OUTPUT: $[\![a^2]\!]_i$ and $[\![a^2]\!]\mathfrak{g}$

1: Let $a_i = [\![a]\!]_i[\![a]\!]_i + [\![z]\!]_i^{2t}$
2: Let $\pi_i := \mathsf{dleq.Prove}([\![a]\!]_i, \mathfrak{g}, [\![a]\!]_i\mathfrak{g}, [\![a]\!]_i\mathfrak{g}, [\![a]\!]_i[\![a]\!]_i\mathfrak{g})$
3: Send $\langle \mathtt{SQ}, [\![a]\!]_i[\![a]\!]_i\mathfrak{g}, a_i, \pi_i\rangle$ to all
4: Let $K = \{\}$
5: upon receiving $\langle \mathtt{SQ}, \hat{a}_j\mathfrak{g}, \tilde{a}_j, \pi_j\rangle$ from party $j$ do
6:     Check $\pi_j$ is a valid proof
7:     Check $\hat{a}_j\mathfrak{g} + [\![z]\!]_j^{2t}\mathfrak{g} = \tilde{a}\mathfrak{g}$
8:     if both checks are successful then
9:         $K := K \cup \{j, \tilde{a}_j\}$
10:    if $|K| \geq 2t+1$ then
11:        Compute $a^2 + z$ by interpolating the values of $K$
12:        Let $[\![a^2]\!]_i := (a^2 + z) - [\![z]\!]_i$
13:        Let $[\![a^2]\!]_j\mathfrak{g} := (a^2 + z)\mathfrak{g} - [\![z]\!]_j\mathfrak{g}$
14:        Compute $[\![a^2]\!]\mathfrak{g}$ by interpolating $[\![a^2]\!]_j\mathfrak{g}$ for all $j \in K$.
15:        output $[\![a^2]\!]_i$ and $[\![a^2]\!]\mathfrak{g}$
```

Party i then multicasts the message $\langle \mathtt{SQ}, [\![a]\!]_i[\![a]\!]_i + [\![z]\!]_i^{2t}, [\![a]\!]_i[\![a]\!]_i\mathfrak{g}, \pi_i\rangle$ to every party. Upon receiving the message $\langle \mathtt{SQ}, \tilde{a}_j, \mathfrak{g}_j, \pi_j\rangle$, party i checks that $\tilde{a}_j$ is computed correctly, i.e.,

$$\mathsf{dleq.Verify}(\mathfrak{g}, [\![a]\!]_j\mathfrak{g}, [\![a]\!]_j\mathfrak{g}, \mathfrak{g}_j, \pi_j) = 1; \text{ and}$$
$$\tilde{a}_j\mathfrak{g} = \mathfrak{g}_j + [\![z]\!]_j^{2t}\mathfrak{g}$$

Upon receiving $2t+1$ valid SQ messages, parties reconstruct $a^2 + z$ by interpolating $\tilde{a}_j$, i.e.,

$$a^2 + z = \sum_j \lambda_j \tilde{a}_j \tag{4}$$

where λ_j is the appropriate Lagrange coefficients.

Upon reconstructing $a^2 + z$, party i computes its share of a^2 as $[\![a^2]\!]_i = a^2 + z - [\![z]\!]_i$. Furthermore, for each $j \in [n]$, party i computes $[\![a^2]\!]_j\mathfrak{g}$ as:

$$[\![a^2]\!]_j\mathfrak{g} := (a^2 + z)\mathfrak{g} - [\![z]\!]_j\mathfrak{g} \tag{5}$$

In §VI-A, we prove that Π_{sq} securely realizes $\mathcal{F}_{\mathsf{sq}}$ with a total communication cost of $O(n^2)$ per invocation.

B. *Using $\mathcal{F}_{\mathsf{sq}}$ for Generating Powers-of-two.*

We now describe how parties use Π_{sq} to compute $\tau^{2^k}\mathfrak{g}$ for every $k \in [\log q]$, i.e., the *powers-of-two*. While computing the powers-of-two, parties also output auxiliary values that they later use to compute the remaining powers.

Parties start by running an ADKG protocol to secret share a uniformly random secret τ using a $(n, t+1)$ Shamir secret sharing. In our implementation, we use the ADKG protocol from [21] with a reconstruction threshold of t. As a result, parties also output the *threshold* public keys $[\![\tau]\!]\mathfrak{g}$.

While running the ADKG protocol, parties concurrently run Π_{Dou}, the protocol to generate random double sharing of $\log q$ uniform random secrets $\{z_1, z_2, \ldots, z_{\log q}\}$.

Once the ADKG and random double sharing protocol terminate, parties compute powers-of-two as follows. The protocol

Algorithm 3 Π_{all} protocol at party i

```
INPUT: $\{[\![\tau^{2^k}]\!]_i; \tau^{2^k}\mathfrak{g}, [\![\tau^{2^k}]\!]\mathfrak{g}\}$ for each $k \in [\log q]$
OUTPUT: $\{\tau^a\mathfrak{g}\}$ for each $a \in [q]$

1: Create a binary tree with $u_0^0$ as its root and $\mathsf{val}(u_0^0) = \tau^0\mathfrak{g}$.
2: for each depth $d = 1, ..., \log q$ do
3:     Create $2^d$ nodes labeled $u_0^d, u_1^d, ..., u_{2^d-1}^d$
4:     Let $\{\mathsf{val}(u_{2j}^d)\} := \{\mathsf{val}(u_{2j}^{d-1})\},\ \forall j \in [0, 2^{d-1}-1]$.
5:     Let $\alpha := \tau^{2^{d-1}}$
6:     Let $\{\mathsf{val}(u_{2j+1}^d)\} := \Pi_{\mathsf{BatMul}}([\![\alpha]\!]_i, [\![\alpha]\!]\mathfrak{g}, \{\mathsf{val}(u_{2j+1}^{d-1})\})$
       $\forall j \in [0, 2^{d-1}-1]$.
7: output $\mathsf{val}(u_0^{\log q}), \mathsf{val}(u_1^{\log q}), \ldots, \mathsf{val}(u_{q-1}^{\log q})$.
```

proceeds in rounds starting with round 1. In round 1, parties invoke Π_{sq} to receive secret shares of τ^2 along with $[\![\tau^2]\!]\mathfrak{g}$. Recall that at the start of round 1, as part of the ADKG output, each party i has already received $[\![\tau]\!]_i$ and $[\![\tau]\!]\mathfrak{g}$. Similarly, in round 2, parties use secret shares of τ^2 and $[\![\tau^2]\!]\mathfrak{g}$ to compute secret shares of τ^4 and $[\![\tau^4]\!]\mathfrak{g}$, and so on. In other words, parties will compute powers-of-two by repeated invocations of the Π_{sq} protocol $\log q$ times in sequence, i.e.,

$$[\![\tau^2]\!]; [\![\tau^2]\!]\mathfrak{g} = \Pi_{\mathsf{sq}}\left([\![\tau]\!]; [\![\tau]\!]\mathfrak{g}\right)$$
$$[\![\tau^4]\!]; [\![\tau^4]\!]\mathfrak{g} = \Pi_{\mathsf{sq}}\left([\![\tau^2]\!]; [\![\tau^2]\!]\mathfrak{g}\right)$$
$$\vdots$$
$$[\![\tau^{2^k}]\!]; [\![\tau^{2^k}]\!]\mathfrak{g} = \Pi_{\mathsf{sq}}\left([\![\tau^{2^{k-1}}]\!]; [\![\tau^{2^{k-1}}]\!]\mathfrak{g}\right)$$

V. Generating All Powers

Using the protocol described in §IV, parties obtain secret shares of powers-of-two of τ, i.e., $[\![\tau, \tau^2, \tau^4, \ldots, \tau^q]\!]$, as well as $\tau^{2^k}\mathfrak{g}$ and $[\![\tau^{2^k}]\!]\mathfrak{g}$ for each $k \in [\log q]$. In this section, we will describe how parties compute $\tau^a\mathfrak{g}$ for all remaining $a \in [q]$. Formally, the interface of this functionality Π_{all} is:

$$\{\tau^a\mathfrak{g}\}_{\forall a \in [q]} = \Pi_{\mathsf{all}}\left(\left\{[\![\tau^{2^k}]\!]; \tau^{2^k}\mathfrak{g}, [\![\tau^{2^k}]\!]\mathfrak{g}\right\}_{\forall k \in [\log q]}\right)$$

A. *Main idea*

As we briefly describe in §II-C, any integer $a \in [q]$ can be written as:

$$a = \sum_{k=1}^{\log q} b_k 2^{k-1} \tag{6}$$

where b_k is the k-th bit in the binary representation of a. Thus, we can write $\tau^a\mathfrak{g}$ as,

$$\tau^a\mathfrak{g} = \left(\prod_{k \in [\log q]} \tau^{b_k 2^k}\right)\mathfrak{g} \tag{7}$$

Next, we use the idea, referred to as the *multiplication in the exponent*, that given secret shares of τ^α, $[\![\tau^\alpha]\!]\mathfrak{g}$, and $\tau^\beta\mathfrak{g}$, parties can compute $\tau^{\alpha+\beta}\mathfrak{g}$ using $O(n)$ per party communication cost and only one round of interaction. In particular, each

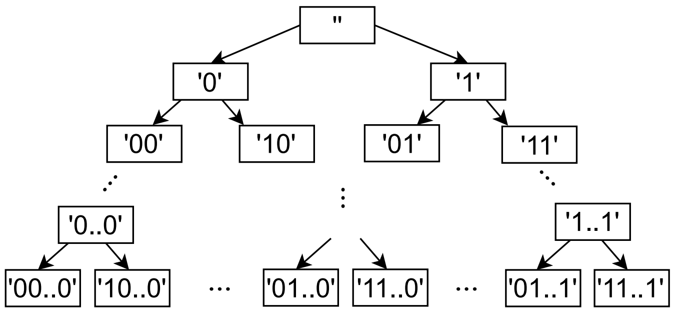

Figure 3: The memoization protocol to compute all powers using a total $O(q)$ multiplications in the exponent.

party i, locally computes $[\![\tau^{\alpha}]\!]_i\beta\mathfrak{g}$ and the NIZK proof π_i of its correctness. Party i then sends the tuple $\langle [\![\tau^{\alpha}]\!]_i\tau^{\beta}\mathfrak{g}, \pi_i\rangle$ to all parties. Also, upon receiving $\langle \mathfrak{g}_j, \pi_j\rangle$ from party j, party i validates $\mathfrak{g}_j$ for correctness using $[\![\tau^{\alpha}]\!]_i\mathfrak{g}$, $\tau^{\beta}\mathfrak{g}$ and π_j. Finally, upon receiving $t+1$ such valid tuples T, it computes $\tau^{\alpha+\beta}\mathfrak{g}$ as

$$\tau^{\alpha+\beta}\mathfrak{g} = \tau^{\alpha}\tau^{\beta}\mathfrak{g} == \sum_{i\in T}\lambda_i\mathfrak{g}_i \tag{8}$$

Thus, for any given $a \in [q]$, parties can easily compute $\tau^a\mathfrak{g}$ by repeating this technique for $\log q$ iterations, as per the bit representation of a and equation 7. This requires per-party communication cost of $O(n\log q)$ and $O(\log q)$ rounds of interactions. Hence, the per-party communication cost for computing $\tau^a\mathfrak{g}$ for all $a \in [q]$ is $O(nq\log q)$ and each party performs $O(nq\log q)$ group multiplications.

We will next reduce the per-party communication cost to $O(nq)$ and computation cost to $O(nq)$ group multiplications using *memoization*. In §V-C, we will further reduce the communication cost to $O(q + n\log q)$ and computation cost to $O(q\log n)$ group multiplications and $O(n\log q)$ pairings using appropriate batching.

B. Memoization

We observe that many of the $O(\log q)$ multiplications in the exponent are redundant. Consider two integers a, b that differ only in their most significant bit, i.e., $a = 0\|s$ and $b = 1\|s$ for some binary string s. Then, once we compute $\tau^a\mathfrak{g}$ (which equals $\tau^s\mathfrak{g}$), we can directly compute $\tau^b\mathfrak{g}$ as $\tau^{2^{|s|}}\tau^a\mathfrak{g}$, using only one more multiplication in the exponent.

Based on this observation, we create a binary tree of height $\log q$ where each node is associated with a binary string as illustrated in Figure 3. We sequentially place the binary representations of each $a = 0, 1, ..., q$ at the leaves of the binary tree. Any path in the tree from the root r to a leaf node a consists of internal nodes $r = a_0$, a_1, a_2, ..., and $a_{\log q} = a$. Stated differently, the left child and the right child of a node with the binary string s are binary strings $0\|s$ and $1\|s$, respectively. To each node in the tree with bit string s, we associate a value, $\mathsf{val}(s) = \tau^s\mathfrak{g}$. The root of the tree is initialized with the empty string and the value $\tau^0\mathfrak{g}$.

To compute $\tau^a\mathfrak{g}$ for $a \in [q]$, we only need to traverse the tree from the root to the leaves (using parallel breadth-first), and compute $\tau^x\mathfrak{g}$ for each internal node with the binary string x. As mentioned, if a node x is the left child of a node s, then $\tau^x\mathfrak{g} = \tau^s\mathfrak{g}$; if x is the right child, then $\tau^x\mathfrak{g} = \tau^{2^{|s|}}\tau^s\mathfrak{g}$. Finally, the $\tau^a\mathfrak{g}$ for $a \in [q]$ can be computed at all the leaves. Parties now compute one multiplication in the exponent per internal node of the tree, leading to a total of $O(q)$ multiplication in the exponent. The per-party communication and computation (in group multiplications) using this approach are both $O(nq)$.

Algorithm 4 Π_{BatMul} protocol for party i

```
INPUT: $\mathfrak{g}, [\![\alpha]\!]_i, [\![\alpha]\!]\mathfrak{g}, \{\beta_1\mathfrak{g}, \beta_2\mathfrak{g}, \ldots, \beta_m\mathfrak{g}\}$
OUTPUT: $\{\alpha\beta_1\mathfrak{g}, \alpha\beta_2\mathfrak{g}, \ldots, \alpha\beta_m\mathfrak{g}\}$.

11: Let $\ell = \lceil m/(n-t)\rceil$ be the number of batches.
12: Let $\beta_1^{(1)}, \ldots, \beta_{n-t}^{(1)}, \ldots, \beta_1^{(\ell)}, \ldots, \beta_{n-t}^{(\ell)}$ denote $\beta_1, \beta_2, \ldots, \beta_m$
13: for each batch $k \in [\ell]$ do
14:     Let $\beta^{(k)}(x) = \beta_1^{(k)} + \beta_2^{(k)}x + \cdots + \beta_{n-t}^{(k)}x^{n-t-1}$
15:     Compute $[\![\alpha]\!]_i\beta^{(k)}(j)\mathfrak{g}, \forall j \in [n]$ using NTT
    // Derive shared randomness using random oracle
16: Let $\gamma_1, \gamma_2, \ldots, \gamma_\ell = \mathsf{RO}(\{\beta_1\mathfrak{g}, \beta_2\mathfrak{g}, \ldots, \beta_m\mathfrak{g}\})$
17: for each $j \in [n]$ do
18:     Let $\mathfrak{a}_j := \sum_{k\in[\ell]}\gamma_k\beta^{(k)}(j)$
19:     Let $\mathfrak{b}_{i,j} = [\![\alpha]\!]_i\mathfrak{a}_j$
20:     Let $\pi_{i,j} = \mathsf{dleq.Prove}([\![\alpha]\!]_i, \mathfrak{g}, [\![\alpha]\!]_i\mathfrak{g}, \mathfrak{a}_j, \mathfrak{b}_{i,j})$    ▷ Batching
21: Send $\langle \texttt{SHARE}, \{[\![\alpha]\!]_i\beta^{(k)}(j)\mathfrak{g}, \}_{\forall k\in[\ell]}, \pi_{i,j}\rangle$ to party $j$
22: Let $T_k = \{\}$ for each $k \in [\ell]$
23: upon receiving $\langle \texttt{SHARE}, \{\tilde{\mathfrak{g}}_j^{(k)}\}_{\forall k\in[\ell]}, \pi_{j,i}\rangle$ from party $j$ do
24:     Let $\tilde{\mathfrak{b}}_{j,i} := \sum_{k\in[\ell]}\gamma_k\tilde{\mathfrak{g}}_j^{(k)}$
25:     if $\mathsf{dleq.Verify}(\mathfrak{g}, [\![\alpha]\!]_j\mathfrak{g}, \mathfrak{a}_i, \tilde{\mathfrak{b}}_{j,i}, \pi_{j,i})$ then
26:         $T_k := T_k \cup \{(j, \tilde{\mathfrak{g}}_j^{(k)})\}$ for each $k \in [\ell]$
27:     if $|T_k| \geq t+1$ then
28:         Compute $\alpha\beta^{(k)}(i)\mathfrak{g}$ using Lagrange interpolation
29: Send $\langle \texttt{EVAL}, \{\alpha\beta^{(k)}(i)\mathfrak{g}\}_{\forall k\in[\ell]}\rangle$ to all parties
30: Let $S_k = \{\}$ for each $k \in [\ell]$
31: upon receiving $\langle \texttt{EVAL}, \{\hat{\mathfrak{g}}_j^{(k)}\}_{\forall k\in[\ell]}\rangle$ from party $j$ do
32:     Locally sample $\chi_k \in \mathbb{F}$ for each $k \in [\ell]$
33:     if $e(\sum_{k\in[\ell]}\chi_k\beta^{(k)}(j)\mathfrak{g}, \alpha\mathfrak{g}) = e(\sum_{k\in[\ell]}\chi_k\hat{\mathfrak{g}}_j^{(k)}, \mathfrak{g})$ then
34:         $S_k := S_k \cup \{(j, \hat{\mathfrak{g}}_j^{(k)}\}$, for each $k \in [\ell]$
35:         if $|S_k| \geq n-t$, $\forall k \in [\ell]$ then
36:             Compute $\{\alpha\beta_a^{(k)}\mathfrak{g}\}_{\forall a\in[n-t]}$ using NTT for $k \in [\ell]$
37:             output $\{\alpha\beta_a\mathfrak{g}\}_{\forall a\in[m]}$
```

C. Batched Multiplication in the Exponent

Given $\{\beta_1\mathfrak{g}, \beta_2\mathfrak{g}, \ldots, \beta_m\mathfrak{g}\}$, $[\![\alpha]\!]$, and publicly available $[\![\alpha]\!]\mathfrak{g}$, we need to compute $\{\alpha\beta_1\mathfrak{g}, \alpha\beta_2\mathfrak{g}, \ldots, \alpha\beta_m\mathfrak{g}\}$. More formally,

$$\{\alpha\beta_k\mathfrak{g}\}_{\forall k\in[m]} = \Pi_{\mathsf{BatMul}}\left([\![\alpha]\!]; [\![\alpha]\!]\mathfrak{g}, \{\beta_k\mathfrak{g}\}_{\forall k\in[m]}\right) \tag{9}$$

One naïve approach is to compute $\alpha\beta_k\mathfrak{g}$ for each $k \in [m]$ separately. This would result in a per party communication cost of $O(nm)$ and a per-party computation cost of $O(nm)$ group multiplications.

We will next describe protocol Π_{BatMul}, which performs the above task with a per-party communication cost of $O(n+m)$

and the per-party computation cost of $O(m \log n)$ group multiplications. It does add a per-party computation cost of $O(n)$ (independent of m) bilinear pairing operations. We provide the pseudocode of Π_{BatMul} in Algorithm 3 and describe it next.

For simplicity, we will assume that $m = n - t$. For $m > n - t$, we can divide the inputs into batches of size $n - t$ and run Π_{BatMul} in parallel (with minor modifications) for each batch. Let $\beta(\cdot)$ be the polynomial of degree $n - t - 1$:

$$\beta(x) = \beta_1 + \beta_2 x + \beta_3 x^2 + \cdots + \beta_{n-t} x^{n-t-1} \tag{10}$$

Given $\{\beta_k \mathfrak{g}\}$ for $k \in [n - t]$, each party i locally computes $\beta(j)\mathfrak{g}$ for each $j \in [n]$, i.e., evaluate the polynomial $\beta(\cdot)$ at all indices in $[n]$. Parties can compute these values using $O(n \log n)$ group multiplications using the Number Theoretic Transform (NTT). Additionally, party i locally computes $[\![\alpha]\!]\beta(j)\mathfrak{g}$ for each $j \in [n]$ along with the dleq proof $\pi_{i,j}$, given as,

$$\pi_{i,j} = \mathsf{dleq.Prove}\left([\![\alpha]\!]_i, \mathfrak{g}, [\![\alpha]\!]_i \mathfrak{g}, \beta(j)\mathfrak{g}, [\![\alpha]\!]_i \beta(j)\mathfrak{g}\right)$$

Party i then sends a message $\langle \texttt{SHARE}, [\![\alpha]\!]_i \beta(j)\mathfrak{g}, \pi_{i,j}\rangle$ to party j. Upon receiving $\langle \texttt{SHARE}, \mathfrak{g}_j, \pi_{j,i}\rangle$ from party j, party i validates its correctness using dleq.Verify. Upon receiving $t + 1$ valid SHARE messages, party i computes $\alpha\beta(i)\mathfrak{g}$ as:

$$\alpha\beta(i)\mathfrak{g} = \sum_{j \in T} \lambda_j \mathfrak{g}_j \tag{11}$$

Party i then sends a message $\langle \texttt{EVAL}, \alpha\beta(i)\mathfrak{g}\rangle$ to all parties. Each party, upon receiving $\langle \texttt{EVAL}, \tilde{\mathfrak{g}}_j\rangle$ from party j, validates its correctness by checking:

$$e(\beta(j)\mathfrak{g}, \alpha\mathfrak{g}) = e(\tilde{\mathfrak{g}}_j, \mathfrak{g}) \tag{12}$$

where $e(\cdot, \cdot)$ is the bilinear pairing operation. Note that every party can locally compute $\alpha\mathfrak{g}$ and $\beta(j)\mathfrak{g}$ using the inputs of Π_{BatMul}. Upon receiving $n - t$ valid EVAL messages, party i computes $\{\alpha\beta_1\mathfrak{g}, \alpha\beta_2\mathfrak{g}, \ldots, \alpha\beta_{n-t}\mathfrak{g}\}$ using inverse NTT.

Handling $m > n-t$. For $m > n-t$, parties divide the inputs into batches of size $n-t$ each. Let there be ℓ batches, i.e., $\ell = \lceil m/(n-t) \rceil$, where $\beta^{(k)}(\cdot)$ is the polynomial corresponding to the k-th batch. Parties then run Π_{BatMul} for each chunk in parallel with the following changes.

Batching dleq *proofs of* SHARE *messages.* Each party i, instead of sending ℓ dleq proofs to every party j, sends only one dleq proof $\pi_{i,j}$ attesting the correctness of SHARE messages for all batches. More specifically, for each recipient j, party i computes $\mathfrak{a}_j$ and $\mathfrak{b}_{i,j}$ as

$$\mathfrak{a}_j = \sum_{k \in [\ell]} \gamma_k \beta^{(k)}(j)\mathfrak{g}; \text{ and } \mathfrak{b}_{i,j} = [\![\alpha]\!]_i \mathfrak{a}_j \tag{13}$$

here γ_k are uniformly random elements in $\mathbb{F}$ generated by querying the random oracle on input $\{\beta_1\mathfrak{g}, \beta_2\mathfrak{g}, \ldots, \beta_m\mathfrak{g}\}$. Let $\pi_{i,j}$ be the dleq proof given by

$$\pi_{i,j} = \mathsf{dleq.Prove}\left([\![\alpha]\!]_i, \mathfrak{g}, [\![\alpha]\!]_i \mathfrak{g}, \mathfrak{a}_j, \mathfrak{b}_{i,j}\right) \tag{14}$$

Each party i then sends the SHARE message to party j as $\langle \texttt{SHARE}, \{[\![\alpha]\!]_i \beta^{(k)}(j)\mathfrak{g}\}_{k \in [\ell]}, \pi_{i,j}\rangle$.

Party i, upon receiving $\langle \texttt{SHARE}, \{\tilde{\mathfrak{g}}_j^{(k)}\}_{k \in [\ell]}, \pi_{i,j}\rangle$ message from party j, locally computes $\mathfrak{a}_i$ using the publicly available information. Party i additionally computes $\tilde{\mathfrak{b}}_{j,i}$ as:

$$\tilde{\mathfrak{b}}_{j,i} = \sum_{k \in [\ell]} \gamma_k \tilde{\mathfrak{g}}_j^{(k)}$$

Party i then validates the correctness of the SHARE message by checking that

$$\mathsf{dleq.Verify}(\mathfrak{g}, [\![a]\!]_j \mathfrak{g}, \mathfrak{a}_i, \mathfrak{b}_{j,i}, \pi_{j,i}) = 1 \tag{15}$$

Intuitively, in equation (14), we use the observation that the prover (party i) needs to prove the equality of discrete logarithm of different statements, one for each $\beta^{(k)}(j)\mathfrak{g}$, that shares the same witness $[\![\alpha_i]\!]$. This enables us to batch all the statements into a single statement by taking their *random linear combination*. The check in equation (15) has a negligible error probability of $1/|\mathbb{F}|$ for each batch.

Batching checks of EVAL *messages.* Instead of checking equation (12) independently for every batch, parties combine them into a single check below:

$$e\left(\sum_{k \in [\ell]} \chi_k \beta^{(k)}(j)\mathfrak{g}, \alpha\mathfrak{g}\right) = e\left(\sum_{k \in [\ell]} \chi_k \tilde{\mathfrak{g}}_j^{(k)}, \mathfrak{g}\right) \tag{16}$$

Here, χ_k for each $k \in [\ell]$ are uniformly random field elements samples from $\mathbb{F}$.

Intuitively, in equation (16), party i, instead of individually checking the correctness of each EVAL message received from party j, takes a random linear combination of the values and checks them all at once. Similar to equation (15), the check in equation (16) also has a negligible error probability of $1/|\mathbb{F}|$ for each batch.

Analysis of Π_{BatMul}. For each batch of size $n - t$, each party sends a single SHARE and EVAL message to every other party. Hence, per party communication cost for a batch of size $n - t$ is $O(n)$. This implies that the per-party communication for a batch of size m is $O(n + m)$. Also, for each batch of size $n-t$, each party performs $O(n \log n)$ group multiplication due to NTT [42]. Hence, the total per-party computation cost for a batch of size m is $O(m \log n)$ group multiplications. Finally, due to the batch checking of all EVAL messages from a party, each party needs to perform only $O(n)$ pairing operations per batch. We reiterate that the number of pairing operations is independent of the batch size.

Using Π_{BatMul} to compute all powers. Recall from §V-B that, for any height h of the binary tree, we multiply with the identical τ^{2^h} to the values of every node at height h. Hence, our protocol computes them using Π_{BatMul} with $\alpha = \tau^{2^h}$ and the values at the nodes as $\{\beta_k \mathfrak{g}\}$ for $k = 1, 2, \ldots, 2^h$.

Since parties need to compute a total of q multiplications in the exponent, the total per-party communications cost is $O(q + n \log q)$. Also, each party performs $O(q \log n)$ group multiplications along with $O(n \log q)$ pairings in total.

Simulator $\mathcal{S}_{\mathsf{Sq}}$

Inputs. Set of malicious parties $\mathcal{C}$, secret shares $[\![a]\!]_i, [\![a^2]\!]_i$ for all $i \in \mathcal{C}$, set of threshold public keys $[\![a]\!]\mathfrak{g}$ and $[\![a^2]\!]\mathfrak{g}$.

INPUT GENERATION PHASE:

1) Sample a random polynomial $r(\cdot)$ of degree $2t$.
2) Compute $[\![z]\!]_i^{2t} = r(i) - [\![a]\!]_i[\![a]\!]_i$, $[\![z]\!]_i = r(0) - [\![a^2]\!]_i$ for each $i \in \mathcal{C}$. Also, compute $[\![z]\!]_j^{2t}\mathfrak{g} = (r(j) - [\![a]\!]_j[\![a]\!]_j)\mathfrak{g}$ and $[\![z]\!]_j\mathfrak{g} = (r(0) - [\![a^2]\!]_j)\mathfrak{g}$ for each $j \in [0, n]$, where $[\![a]\!]_0 = a$.

SIMULATION PHASE:

1) Run $\mathcal{S}_{\mathsf{Dou}}$ on input $[\![z]\!]_i, [\![z]\!]_i^{2t}$ for each $i \in [t]$, and $[\![z]\!]\mathfrak{g}, [\![z]\!]^{2t}\mathfrak{g}$.
2) For each emulated honest party j, compute $[\![a]\!]_j[\![a]\!]_j\mathfrak{g}$ using equation (18). Let $\pi_j = \mathcal{S}_{\mathsf{dleq}}(\mathfrak{g}, [\![a]\!]_j\mathfrak{g}, [\![a]\!]_j\mathfrak{g}, [\![a]\!]_j[\![a]\!]_j\mathfrak{g})$ be the simulated proof of equality of discrete logarithm.
3) On behalf of each emulated honest party j, send $\langle \mathtt{SQ}, [\![a]\!]_j[\![a]\!]_j\mathfrak{g}, r(j), \pi_j\rangle$ to every party.

Figure 4: Simulator for the protocol Π_{sq} for functionality $\mathcal{F}_{\mathsf{sq}}$.

VI. ANALYSIS

We first analyze the correctness and security of Π_{sq} and then prove the security of our overall protocol.

A. Analysis of the Squaring Protocol

In this subsection, we will assume that parties start with correct double sharing of uniform random value z.

Correctness. The verification of $\mathtt{SQ}$ messages ensures that honest parties only accept valid $\mathtt{SQ}$ messages. This implies that honest parties only interpolate correct shares and hence will correctly output $a^2 + z$. Finally, $a^2 + z - [\![z]\!]_i$ is a valid secret share of a^2 i.e., $[\![a^2]\!]$ since

$$\sum_j \lambda_j(a^2 + z - [\![z]\!]_j) = a^2 + z - \sum_j \lambda_j[\![z]\!]_j = a^2 \tag{17}$$

Security. We will prove the security of Π_{sq} by showing its *simulatability*. Specifically, we will illustrate that, for any static PPT adversary $\mathcal{A}$, that corrupts up to t parties and additionally observes $[\![a]\!]\mathfrak{g}$ and $[\![a^2]\!]\mathfrak{g}$, there exists a simulator $\mathcal{S}_{\mathsf{Sq}}$, that takes as input only the adversarial shares and the publicly available $[\![a]\!]\mathfrak{g}$ and $[\![a^2]\!]\mathfrak{g}$, and simulates a view that is indistinguishable from $\mathcal{A}$'s view in the real execution of the protocol.

In our proof, we assume the existence of a simulator $\mathcal{S}_{\mathsf{Dou}}$ for the protocol Π_{Dou}, that securely realizes the ideal functionality $\mathcal{F}_{\mathsf{Dou}}$ shown in figure 8. $\mathcal{S}_{\mathsf{Dou}}$ takes as inputs $[\![z]\!]_i$, $[\![z]\!]_i^{2t}$ for each corrupt party i, along with public values $z\mathfrak{g}$, $[\![z]\!]\mathfrak{g}$, and $[\![z]\!]^{2t}\mathfrak{g}$. $\mathcal{S}_{\mathsf{Dou}}$ then simulates one invocation of Π_{Dou} such that at the end of the protocol parties output double shares of z where the shares of the adversarial parties matches the input to $\mathcal{S}_{\mathsf{Dou}}$. The double sharing protocol of [21] does not immediately admits such a simulator. However, as we illustrate in Appendix B, their protocol can be modified with minor overhead to admit such a simulator. We summarize our simulator $\mathcal{S}_{\mathsf{Sq}}$ in Figure 4 and describe it next.

Without loss of generality, we assume that $\mathcal{A}$ corrupts the first t parties. Thus, $\mathcal{S}_{\mathsf{Sq}}$ will receive the shares $[\![a]\!]_i$ and $[\![a^2]\!]_i$ for each $i \in [t]$, along with public values $[\![a]\!]\mathfrak{g}$ and $[\![a^2]\!]\mathfrak{g}$. With these inputs, $\mathcal{S}_{\mathsf{Sq}}$ emulates the remaining $n - t$ honest parties and generates the protocol transcript as follows.

$\mathcal{S}_{\mathsf{Sq}}$ first computes $[\![a]\!]_j[\![a]\!]_j\mathfrak{g}$ for every $j \in [n]$ using its knowledge of $[\![a]\!]_i$ for each $i \in [t]$, $[\![a]\!]\mathfrak{g}$, and the following identity

$$[\![a]\!]_j[\![a]\!]_j = \left(\sum_{k=0}^{t} \mathcal{L}_k(j)[\![a]\!]_k\right)\left(\sum_{k=0}^{t} \mathcal{L}_k(j)[\![a]\!]_k\right) \tag{18}$$

$\mathcal{S}_{\mathsf{Sq}}$ then samples a random polynomial $r(\cdot)$ of degree $2t$ and computes:

$$\begin{aligned}
[\![z]\!]_j^{2t}\mathfrak{g} &= (r(j) - [\![a]\!]_j[\![a]\!]_j)\mathfrak{g} \;\forall j \in [0, n];\\
[\![z]\!]_i^{2t} &= r(i) - [\![a]\!]_i[\![a]\!]_i \;\forall i \in [t]\\
[\![z]\!]_i &= r(0) - [\![a^2]\!]_i \;\forall i \in [t]
\end{aligned}$$

$\mathcal{S}_{\mathsf{Sq}}$ then runs $\mathcal{S}_{\mathsf{Dou}}$ using the values computed above. Also, for each emulated honest party j, $\mathcal{S}_{\mathsf{Sq}}$ uses the NIZK simulator $\mathcal{S}_{\mathsf{dleq}}$ of the dleq protocol to compute the simulated proof $\pi_j = \mathcal{S}_{\mathsf{dleq}}(\mathfrak{g}, [\![a]\!]_j\mathfrak{g}, [\![a]\!]_j\mathfrak{g}, [\![a]\!]_j[\![a]\!]_j\mathfrak{g})$. Finally, on behalf of each emulated honest party j, $\mathcal{S}_{\mathsf{Sq}}$ multicasts the message $\langle \mathtt{SQ}, [\![a]\!]_j[\![a]\!]_j\mathfrak{g}, r(j), \pi_j\rangle$ to every party.

We now prove that the view of $\mathcal{A}$ in the simulated protocol is identical to $\mathcal{A}$'s view in the real execution of the protocol. Note that $r(\cdot)$ is a uniformly random polynomial of degree $2t$, and Chaum-Pedersen protocol for dleq is perfect zero-knowledge. This implies that the distribution of the values sent with $\mathtt{SQ}$ messages is identical to the real protocol execution. Finally, as we illustrate in Appendix B, $\mathcal{S}_{\mathsf{Dou}}$ perfectly simulates the random double-sharing protocol.

B. Correctness

Lemma 1 (Correctness)**.** *If all honest parties start the protocol, then every honest party will output correct q-SDH parameters, i.e., there exists a $\tau \in \mathbb{F}$ such that parties output $\{\mathfrak{g}, \tau\mathfrak{g}, \tau^2\mathfrak{g}, \ldots, \tau^q\mathfrak{g}\}$, except with $\mathsf{negl}(\kappa)$ probability.*

Proof. We will prove correctness in $\mathcal{F}_{\mathsf{ADKG}}$ and $\mathcal{F}_{\mathsf{Dou}}$ hybrid model. $\mathcal{F}_{\mathsf{ADKG}}$ guarantees that parties agree on a common public key $\tau\mathfrak{g}$, $[\![\tau]\!]\mathfrak{g}$, and each party i has $[\![\tau]\!]_i$. $\mathcal{F}_{\mathsf{Dou}}$ guarantees that parties output double shares of $\log q$ random field elements $\{z_1, z_2, \ldots, z_{\log q}\}$ along with $[\![z_k]\!]\mathfrak{g}$ and $[\![z_k]\!]^{2t}\mathfrak{g}$. With this setup, we will argue that all honest parties agree and output $\{\mathfrak{g}, \tau\mathfrak{g}, \tau^2\mathfrak{g}, \ldots, \tau^q\mathfrak{g}\}$.

In §VI-A, we prove that Π_{sq} securely realize $\mathcal{F}_{\mathsf{sq}}$. This implies that during the squaring phase, parties output and agree on $\tau^{2^k}\mathfrak{g}$ and $[\![\tau^{2^k}]\!]\mathfrak{g}$ for each $k \in [\log q]$. Moreover, each party i outputs $[\![\tau^{2^k}]\!]_i$. Finally, while computing the remaining powers of τ, parties only accept valid $\mathtt{SHARE}$ and

Reduction Adversary $\mathcal{A}_{\mathsf{qSDH}}$

Inputs. q-SDH parameters $\{\mathfrak{g}, \tau\mathfrak{g}, \tau^2\mathfrak{g}, \ldots, \tau^q\mathfrak{g}\}$

1) Sample uniform random $[\![\tau]\!]_i$ for each $i \in \mathcal{C}$ where $\mathcal{C}$ is the set of corrupted parties.
2) For each $k \in [\log q]$, sample uniformly at random $[\![\tau^{2^k}]\!]_i$ for each $i \in \mathcal{C}$.
3) Run the input generation phase of $\mathcal{S}_{\mathsf{Sq}}$ on inputs $[\![\tau^{2^{k-1}}]\!]_i, [\![\tau^{2^k}]\!]_i$ for each $i \in \mathcal{C}$, and public values $[\![\tau^{2^{k-1}}]\!]\mathfrak{g}, [\![\tau^{2^k}]\!]\mathfrak{g}$ for every $k \in [\log q]$.
4) Run the ADKG simulator $\mathcal{S}_{\mathsf{ADKG}}$ on input $\mathfrak{g}$, $\tau\mathfrak{g}$, $[\![\tau]\!]\mathfrak{g}$, and $\{[\![\tau]\!]_i\}$ for each $i \in \mathcal{C}$.
5) Run $\mathcal{S}_{\mathsf{Dou}}$ to simulate a protocol to generate double sharing of $\log q$ values, with inputs computed during the input generation phase of $\mathcal{S}_{\mathsf{Sq}}$.
6) While computing powers-of-two, run step 2 and 3 of the simulation phase of $\mathcal{S}_{\mathsf{Sq}}$.
7) During all powers phase, follow the honest protocol, except, whenever needed, generate the NIZK proof of equality of discrete logarithm using $\mathcal{S}_{\mathsf{dleq}}$.

Figure 5: Simulator for the protocol Π_{qSDH} simulating $\mathcal{F}_{\mathsf{qSDH}}$

EVAL messages, except with probability $O(1/|\mathbb{F}|)$. Since, $|\mathbb{F}|$ is super-polynomial in κ, the security parameter, this implies that all honest parties agree and output $\{\mathfrak{g}, \tau\mathfrak{g}, \tau^2\mathfrak{g}, \ldots, \tau^q\mathfrak{g}\}$, except with negligible probability. $\square$

C. Security

Security of our protocol relies on the hardness of q-SDH assumption for parameters generated by $\mathcal{F}_{\mathsf{qSDH}}$ defined in Figure 2. Specifically, we prove that if a *static* adversary $\mathcal{A}$ can break the q-SDH assumption for parameters generated by our protocol, then we can use $\mathcal{A}$ to build an adversary $\mathcal{A}_{\mathsf{qSDH}}$ such that $\mathcal{A}_{\mathsf{qSDH}}$ can break the q-SDH assumption on parameters generated by the ideal functionality $\mathcal{F}_{\mathsf{qSDH}}$. This implies that if the q-SDH assumption holds for parameters generated by $\mathcal{F}_{\mathsf{qSDH}}$, then the q-SDH assumption holds for parameters generated by TAURON. We summarize the reduction adversary $\mathcal{A}_{\mathsf{qSDH}}$ in Figure 5 and describe it next.

Let $\{\mathfrak{g}, \tau\mathfrak{g}, \tau^2\mathfrak{g}, \ldots, \tau^q\mathfrak{g}\}$ be the parameters generated by $\mathcal{F}_{\mathsf{qSDH}}$. $\mathcal{A}_{\mathsf{qSDH}}$ upon receiving these parameters, simulates an execution of our protocol for $\mathcal{A}$ that outputs $\{\mathfrak{g}, \tau\mathfrak{g}, \tau^2\mathfrak{g}, \ldots, \tau^q\mathfrak{g}\}$ as the q-SDH parameters.

Let $\mathcal{C}$ be the set of parties corrupted by $\mathcal{A}$. For each $k \in [0, \log q]$, $\mathcal{A}_{\mathsf{qSDH}}$ samples uniformly random shares $[\![\tau^{2^k}]\!]_i$ for each $i \in \mathcal{C}$. $\mathcal{A}_{\mathsf{qSDH}}$ then runs the input generation phase of $\mathcal{S}_{\mathsf{Sq}}$ on inputs $[\![\tau^{2^{k-1}}]\!]_i, [\![\tau^{2^k}]\!]_i$ for each $i \in \mathcal{C}$ and public values $[\![\tau^{2^{k-1}}]\!]\mathfrak{g}$ and $[\![\tau^{2^k}]\!]\mathfrak{g}$ for every $k \in [\log q]$. Intuitively, by doing so, $\mathcal{A}_{\mathsf{qSDH}}$ generates the inputs of the $\mathcal{S}_{\mathsf{Dou}}$.

$\mathcal{A}_{\mathsf{qSDH}}$ then runs the ADKG simulator $\mathcal{S}_{\mathsf{ADKG}}$ on input $\mathfrak{g}, \tau\mathfrak{g}$ and $[\![\tau]\!]_i$ for each $i \in \mathcal{C}$. $\mathcal{S}_{\mathsf{ADKG}}$ guarantees that parties output secret share of τ where adversarial shares matches the input to $\mathcal{S}_{\mathsf{ADKG}}$. Simultaneously, $\mathcal{A}_{\mathsf{qSDH}}$ runs $\mathcal{S}_{\mathsf{Dou}}$ on inputs generated by the input generation phase of $\mathcal{S}_{\mathsf{Sq}}$. This concludes simulating the setup phase of our protocol. Next, while computing powers-of-two, $\mathcal{A}_{\mathsf{qSDH}}$ simply runs the step (2) and (3) of $\mathcal{S}_{\mathsf{Sq}}$. Lastly, to compute all remaining powers of τ, $\mathcal{A}_{\mathsf{qSDH}}$ follows the honest protocol, except, whenever needed, generates the NIZK proof of equality of discrete logarithm using the $\mathcal{S}_{\mathsf{dleq}}$.

If $\mathcal{A}$ outputs $(c, (\tau + c)^{-1}\mathfrak{g})$ for $c \in \mathbb{F} \setminus \{-\tau\}$, then $\mathcal{A}_{\mathsf{qSDH}}$ outputs $(c, (\tau + c)^{-1}\mathfrak{g})$.

Lemma 2. *For any PPT adversary $\mathcal{A}$ that corrupts up to t parties, the view of $\mathcal{A}$ during its interaction with $\mathcal{A}_{\mathsf{qSDH}}$ is identically distributed to its view in the real protocol.*

Proof. This follows directly from the fact $\mathcal{S}_{\mathsf{ADKG}}$, $\mathcal{S}_{\mathsf{Dou}}$ and $\mathcal{S}_{\mathsf{Sq}}$ perfectly simulate the ideal functionalities $\mathcal{F}_{\mathsf{ADKG}}$, $\mathcal{F}_{\mathsf{Dou}}$ and $\mathcal{F}_{\mathsf{sq}}$, respectively. $\square$

The fact that $\mathcal{A}$ outputs $\{\mathfrak{g}, \tau\mathfrak{g}, \tau^2\mathfrak{g}, \ldots, \tau^q\mathfrak{g}\}$ as a result of its interaction with $\mathcal{A}_{\mathsf{qSDH}}$ and Lemma 2 immediately implies that if $\mathcal{A}$ outputs a valid tuple that breaks the q-SDH assumption with probability ε, $\mathcal{A}_{\mathsf{qSDH}}$ breaks the q-SDH assumption with probability ε as well.

D. Performance

Table III: Cost of each phases in TAURON for any given n and q.

Protocol Phase	Communication (Per Party)	Computation (Per Party)	Expected Latency
Setup	$O(n^2 \log q)$	$O(n^2 \log q)$	$O(\log n)$
Powers-of-two	$O(n \log q)$	$O(n \log q)$	$O(\log q)$
All powers	$O(q + n \log q)$	$O(q \log n)$	$O(\log q)$
Overall	$O(q + n^2 \log q)$	$O(n^2 \log q + q \log n)$	$O(\log(nq))$

Round complexity. The ADKG protocol and the protocol for generating double sharing of random values takes expected $O(\log n)$ rounds of interaction [21]. Computing the powers-of-two takes $O(\log q)$ rounds. Finally, using memoization, computation of all remaining powers takes $O(\log q)$ additional rounds. Thus, the total expected round complexity of TAURON is $O(\log q + \log n)$.

Communication cost. The ADKG protocol generates secret shares of τ and publicly outputs $\tau\mathfrak{g}$ and $[\![\tau]\!]\mathfrak{g}$ with per party expected per-party communication cost of $O(n^2)$ [22]. The per-party communication cost of generating the double shares of $\log q$ random elements is $O(n^2 \log q)$. The communication cost of generating the powers-of-two is $O(n \log q)$ (ref. §IV). Finally, using memoization and batching (ref. §V), the communication cost of computing the remaining powers is $O(q + n \log q)$. Combining all the above, the total per-party communication cost of TAURON for $O(q + n^2 \log q)$. When $q \gg n$, the total communication cost is $O(q)$, i.e., linear in the highest power of the q-SDH parameters. For example, with $n = 128$ and $q = 2^{20}$ i.e., about 1 Million, $q > n^2 \log q$.

Computation cost. We measure the computation cost as the number of group multiplications and pairing operations each party needs to perform in the entire protocol. During the ADKG protocol, each party performs $O(n^2)$ group multiplications [21]. Also, each party performs $O(n^2 \log q)$ group multiplications to compute the double-sharing of random values. While computing the powers-of-two, each party performs $O(n \log q)$ group multiplications in total. Finally, while computing all remaining powers of τ, for every height h in the memoization tree, each party performs $O(2^h \log n)$ group multiplications and $O(n)$ pairings. This implies that the per-party computation cost in TAURON is $O(q \log n)$ group multiplications and $O(n \log q)$ pairings.

External verification. Let $\mathcal{V}$ be an external verifier that wishes to verify the correctness and security of the q-SDH parameters generated by TAURON. Since we assume PKI, this is relatively straightforward. Each party signs the output using its signing key and sends the signature to $\mathcal{V}$. The $\mathcal{V}$ waits for $t+1$ valid signature on the matching output.

VII. IMPLEMENTATION

A. Implementation Details

We have implemented TAURON in `python 3.7.13` on top of the open-source asynchronous DKG codebase of [21]. We use `rust` libraries for elliptic curve operations and `asyncio` for concurrency, though our prototype is single-threaded at each party. We implement the random double-sharing protocol we describe in Appendix B.

Our implementation uses the `bls12381` elliptic curve and the implementation from Zcash [27] (with a `python` wrapper) for primitive elliptic curve operations. Recall that a pairing-friendly curve involves three groups $\mathbb{G}_1, \mathbb{G}_2$, and $\mathbb{G}_\text{T}$. We generate the q-SDH parameters in $\mathbb{G}_1$ as it is more efficient than $\mathbb{G}_2$ [30]. The size of each $\mathbb{G}_1$ and $\mathbb{G}_2$ group element after point compression is 48 Bytes and 96 Bytes, respectively.

For equality of discrete logarithm, we use Chaum-Pedersen's "Σ"-protocol. We choose the Chaum-Pedersen "Σ" protocol over the pairing-based check, as while benchmarking, we found that the Chaum-Pedersen's Σ protocol is approximately 2.75$\times$ computationally more efficient than the pairing-based check.

Recall from §V that parties need to compute coefficients of polynomials of degree $n-t-1$, given any arbitrary subsets of $n-t$ points (`EVAL` messages) on the polynomial. One approach to compute these coefficients is to first compute the evaluations at all points and then apply the inverse NTT transform to get the coefficients. However, this requires each party to perform $O(n^2)$ group multiplications for every polynomial. An asymptotically superior method is the FNT-based NTT implementation due to Soro and Lacan [42]. The latter runs in $O(n \log n)$ time. We implement both approaches, and our microbenchmark illustrates that the quadratic approach performs better for a smaller number of parties.

B. Evaluation Setup

We evaluate our implementation with a varying number of parties: 16, 32, 64, and 128. We evaluate with different values of q: 2^{14}, 2^{16}, and 2^{18}. We run all parties on Amazon Web Services (AWS) *t3x.large* virtual machines (VM) with one party per VM. Each VM has 4 vCPUs and 16GB RAM and runs Ubuntu 20.04.

Table IV: Runtime of different phases, for any (n, q), of TAURON (in % of total runtime). The setup phase represents the combined runtime of both ADKG and the random double-sharing phase.

Protocol Phase	$(16, 2^{14})$	$(16, 2^{18})$	$(128, 2^{14})$	$(128, 2^{18})$
Setup	4.3	0.4	41.0	5.6
Powers-of-two	1.4	0.1	1.3	0.2
All powers	94.3	99.5	57.7	94.2

We place parties evenly across eight different AWS regions: Canada, Ireland, N. California, N. Virginia, Oregon, Ohio, Singapore, and Tokyo. We create an overlay network in which all parties are pair-wise connected, i.e., they form a complete graph.

With this evaluation setup, we measure the *runtime* and per-party *bandwidth* usage. The runtime is measured from the start of the protocol to the time a party outputs the q-SDH parameters. Per-party bandwidth usage is the amount of data in Bytes sent by a party in the entire protocol.

Baselines. Our baseline is the sequential protocol with a synchronous broadcast channel we describe in §I. In the baseline, we implement the state-of-the-art update verification mechanism from [39], which reduces the verification cost of each update from $2q$ pairings to $2q$ group multiplications and 2 pairings.

The runtime of the baseline includes the time a sequence of n parties takes turns to update existing q-SDH parameters and verify all previous updates. Since the deployed versions skip verifications during the protocol, we implemented a pipelined verification step where every party verifies the update by party i as soon as party i finishes its update. This ensures the runtime of the baseline is the time it takes to perform $3nq$ group multiplications. Here, nq group multiplications for computing the q-SDH parameters and an additional $2nq$ group multiplications for verifying the parameters.

We approximate the bandwidth usage of the baseline as nq group elements. Note that this favors the baseline protocol, as in the synchronous protocol, each party will need to broadcast q group elements. Hence, each party will need to send at least nq group elements in the entire protocol.

Remark. The actual running time or bandwidth usage of the baseline will be higher than what we reported in the figures, as we only measure sub-components for the baseline.

C. Evaluation Results

With our evaluation, we demonstrate that TAURON scales well with the number of parties n and q, and that TAURON has reasonable runtime and bandwidth usage.

Runtime. We report the median runtime of TAURON and baseline, computed across the parties for a single run of each experiment, in Figure 7. The solid and dashed lines represent the runtime of TAURON and baseline, respectively. We also

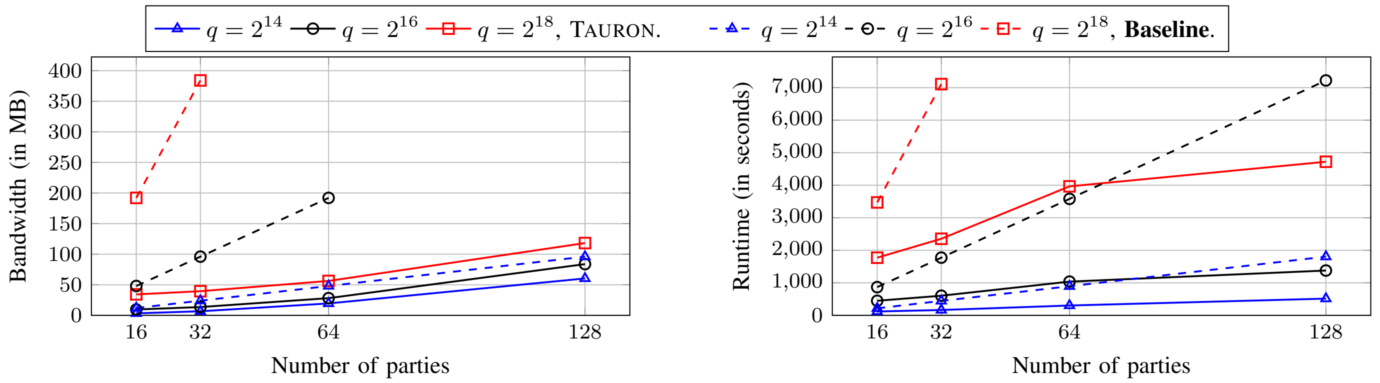

Figure 6: Median bandwidth usage, measured as the amount of data sent by a party in the entire protocol.

Figure 7: Median runtime, measured as the time between the start of the protocol and the time parties output the q-SDH parameters.

report the breakdown of runtime across different phases of Tauron in Table IV.

Our evaluation results corroborate our analysis in §VI-D. Specifically, for any fixed q, the runtime of Tauron grows logarithmically with the number of parties. Also, for any given n, the runtime grows linearly with q. Note that although the runtime should grow logarithmically for $n = 16$ and $n = 32$, we see a linear growth because, for $n = 16$ and $n = 32$, we implement a protocol with asymptotic runtime of $O(nq)$, but with smaller constants.

Our evaluation illustrates that Tauron is significantly more efficient than the baseline protocol. For example, with 64 parties and $q = 2^{16}$, Tauron takes approximately 1037 seconds, whereas the baseline protocol takes 3580 seconds (3.4× faster). Similarly, with 128 parties and $q = 2^{18}$, Tauron takes approximately 4721 seconds compared to 28883 seconds (not shown in the figure) taken by the baseline protocol, hence 6.1× faster than the baseline protocol. Note that, for the baseline, we only measure the runtime without any networking component. Thus, if we also include the runtime of the synchronous broadcast, the benefits of Tauron will be even bigger.

In Table IV, we break down the runtime of different phases in Tauron for different choices of n and q. In Table IV, the setup includes the runtime of both ADKG and random double-sharing protocol. We merge them as our implementation runs these phases such that they share some building blocks. Moreover, both ADKG and the random double-sharing phase use the same invocation of the consensus protocol. Based on Table IV, we conclude the following.

First, the powers-of-two phase takes less than 2% of the runtime in all experiments. Recall from Table III, during the powers-of-two phase, each party performs $O(n \log q)$ group multiplications and sends $O(n \log q)$ group elements. This is significantly smaller than the computation cost and bandwidth usage of setup and all powers phase. However, the powers-of-two phase requires $O(\log q)$ rounds of interaction, which is comparable to the setup and all powers phases. This illustrates that the number of communication rounds is not a bottleneck.

Second, for smaller n, the runtime of the all-powers phase contributes to almost all of the runtime of Tauron, especially

Table V: Bandwidth usage of different phases of Tauron (in % of total bandwidth usages). The setup phase corresponds to the combined bandwidth usages of both ADKG and the random double-sharing phase. The tuple represents (n, q).

Protocol Phase	$(16, 2^{14})$	$(16, 2^{18})$	$(128, 2^{14})$	$(128, 2^{18})$
Setup	34.4	4.1	95.2	68.8
Powers-of-two	2.2	0.3	0.9	0.7
All powers	63.4	95.6	3.9	30.5

for large q. This is as expected because each party needs to perform $O(n^2 \log q)$ and $O(q \log n)$ group multiplications during the setup phase and all powers phase, respectively. For larger q and smaller n, the latter is significantly larger than the former. However, with increasing n, as with the case of $n = 124, q = 2^{14}$, the computation cost of the setup phase also becomes significant. Finally, as expected, for $n = 128$, with larger $q = 2^{18}$, the runtime of the all powers phase again starts to dominate the total runtime.

Bandwidth usage. We report the per-party bandwidth usage, i.e., the amount of data (in Megabytes) each party sends during the entire protocol, in Figure 6. The solid and dashed lines represent the per-party bandwidth usage of Tauron and baseline, respectively. Consistent with the analysis from §VI, the bandwidth usage in Tauron increases quadratically with the number of parties and linear in the q. We also report the breakdown of bandwidth usages across different phases of Tauron in Table V.

Our evaluation illustrates that parties in Tauron incur significantly less bandwidth usage than the baseline. For example, with 32 parties and $q = 2^{16}$, each party needs to send 13.57 Megabytes (MB) of data in Tauron, compared to 96 MB of data in the baseline protocol (7× reduction). Similarly, with 128 parties and $q = 2^{18}$, the bandwidth usage in Tauron is 118.17 MB, compared to 1536 MB in the baseline protocol; hence, 13× less bandwidth usage than the baseline. Again, for the baseline, we only measure the bandwidth usage as $\kappa n q$ bytes, where $\kappa = 48$ is the size of each $\mathbb{G}_1$ element, and do not account for any bandwidth usage for the synchronous broadcast protocol. If we also include the bandwidth usage due to the synchronous broadcast channel, the benefits of Tauron will be even bigger.

Similar to the breakdown of runtime across different phases, in Table IV, we illustrate the breakdown of bandwidth usage across different phases of TAURON for different choices of n and q. Again, in Table V, the setup includes the runtime of both ADKG and the random double-sharing protocol. From Table V, we draw a few conclusions.

First, for the same reason as the runtime, the bandwidth usage of the powers-of-two phase takes less than 3% of the total bandwidth usage in all experiments.

Second, unlike runtime, the bandwidth usage in the setup phase is significant compared to the bandwidth usage of the all-powers phase. This is because the ratio between the bandwidth usage of the setup and the all-powers phase is greater, by a factor of $\log n$, than the ratio between the computation cost of the setup and the all-powers phase. Specifically, the former is approximately $O((n^2 \log q)/q)$ while the latter is $O((n^2 \log q)/q \log n)$. For this same reason, for large $n = 128$ and small $q = 2^{14}$, the bandwidth usage of the setup phase is more than 95% of the total bandwidth usage.

Finally, we conclude that the computation cost is the primary bottleneck of TAURON. With $n = 128$, the setup phase, despite contributing 95.2% and 68.8% of the bandwidth usage for $q = 2^{14}$ and $q = 2^{18}$, respectively, contributes only 41% and 5.5% of the total runtime.

VIII. RELATED WORK

Ben-Sasson et al. [8] proposed the first distributed protocol for sampling arbitrary structured public parameters in a secure manner. The protocol of [8] lays the foundation for the round-robin-style protocols for generating secure structured public parameters. Briefly, in [8], parties take turns to update intermediate parameters with local randomness. Bowe et al. [12] adopt the approach of [8] and present a protocol for generating q-SDH parameters. Their protocol, however, relied on a publicly verifiable, unpredictable, and bias-resistant randomness beacon for security. Kohlweiss et al. [33] illustrated that the parameters generated by [12] are secure even without a randomness beacon. Very recently, Nikolaenko et al., [39] designed a protocol to generate q-SDH parameters using Ethereum as the underlying sequential broadcast channel.

All these protocols assume a synchronous network, have high communication and computation costs, and require n sequential broadcasts. We refer the reader to §I and Table I for a detailed comparison of these schemes with our construction. Very recently, Cohen et al. [17] presented a generic compiler to reduce the round complexity of these protocols to $O(\sqrt{n})$ sequential broadcasts. However, their construction is very theoretical and has very high constants. They also present a compiler with better constants but $O(\sqrt{n} \log q)$ sequential broadcasts.

Practical deployments. The round-robin style protocols for generating q-SDH parameters have been already deployed in practice [1], [36], [2], [25], [3] All these deployments implement variants of [8]. However, as deployed, parties skip verifying the intermediate protocol transcript and verify the entire transcript only at the end. Despite these insecurities, they scale very poorly. For example, according to Semaphore [3], to generate q-SDH parameters for $q = 2^{28}$, each party needs to perform 24-hour long computation. Hence, with n parties, the protocol would run for at least n days.

Comparison with generic MPC. An alternate approach to generate q-SDH parameters in asynchronous networks is to use generic MPC. However, this has many disadvantages.

Let $\mathcal{C}$ be the circuit that outputs the q-SDH parameters, then $\mathcal{C}$ will consist of $O(q)$ multiplication gates. For large q, evaluating $O(q)$ multiplication gates in an asynchronous MPC can be prohibitively expensive. For example, asynchronous MPC protocols that rely on pre-processing either require threshold additive homomorphic encryptions [28], [29], [15] or can tolerate only $n/4$ malicious parties [16]. We want to emphasize that while generating q-SDH parameters, we also have to include the cost of the pre-processing phase in the cost of the overall protocol. Alternatively, protocols that do not rely on a pre-processing step require running $O(n)$ asynchronous complete secret sharing protocol for every multiplication gates [7], [4].

Another issue that arises in using generic MPC is due to the difference between the scalar and based field of elliptic curve groups. Typically, MPC protocols are defined over a single finite field, whereas q-SDH parameter generation involves working with both the scalar field $\mathbb{F}$, from where τ is sampled, and the base field, which is used to define the elliptic curve group elements. Since the scalar $\mathbb{F}$ is different from the base field; the MPC protocol needs to support operations across two distinct fields, which can be prohibitively expensive [20].

IX. CONCLUSION

In this paper, we presented TAURON, a distributed protocol to generate secure parameters for q-Strong Diffie-Hellman, also known as the q-SDH, interactability problem. In an asynchronous network of n parties, TAURON tolerates up to one-third of malicious parties. TAURON is also very efficient. For any given q, the highest degree of the q-SDH parameters, each party incurs a communication cost of $O(q + n^2 \log q)$, which is optimal whenever $q \geq n^2 \log q$. Moreover, TAURON only requires $O(\log q + \log n)$ rounds of interaction. Furthermore, in the entire protocol, each party performs only $O(q \log n)$ group multiplications and $O(n \log q)$ bilinear pairings. We have implemented all parts of TAURON and evaluated it using up to 128 geographically distributed AWS instances. Our evaluation results corroborate the practicality of our approach and its significant improvement over the state-of-the-art protocol.

In addition to improving the performance metrics, there are many other interesting future research directions to consider. One research direction is to design a protocol that achieves similar performance results without assuming bilinear pairings. Another research direction is to generate the parameters for the multi-variate generalization of the q-SDH parameters [40].

Acknowledgments

The authors would like to thank Amit Agarwal and Atsuki Momose for many helpful discussions related to the paper. Work of Sourav Das is supported by a Chainlink Labs Ph.D. fellowship.

References

[1] "Plumo ceremony," https://celo.org/plumo, 2017.

[2] "Universal crs setup," https://docs.zksync.io/userdocs/security/#universal-crs-setup, 2020.

[3] "Perpetual powers of tau," https://github.com/weijiekoh/perpetualpowersoftau, 2021.

[4] I. Abraham, G. Asharov, and A. Yanai, "Efficient perfectly secure computation with optimal resilience," in *Theory of Cryptography Conference*. Springer, 2021, pp. 66–96.

[5] N. Alhaddad, M. Varia, and H. Zhang, "High-threshold avss with optimal communication complexity," in *International Conference on Financial Cryptography and Data Security*. Springer, 2021, pp. 479–498.

[6] D. Beaver, "Efficient multiparty protocols using circuit randomization," in *Annual International Cryptology Conference*. Springer, 1991, pp. 420–432.

[7] M. Ben-Or, S. Goldwasser, and A. Wigderson, "Completeness theorems for non-cryptographic fault-tolerant distributed computation," in *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, ser. STOC '88, New York, NY, USA, 1988, p. 1–10.

[8] E. Ben-Sasson, A. Chiesa, M. Green, E. Tromer, and M. Virza, "Secure sampling of public parameters for succinct zero knowledge proofs," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 287–304.

[9] A. Bhat, N. Shrestha, Z. Luo, A. Kate, and K. Nayak, "Randpiper–reconfiguration-friendly random beacons with quadratic communication," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 3502–3524.

[10] G. R. Blakley, "Safeguarding cryptographic keys," in *1979 International Workshop on Managing Requirements Knowledge (MARK)*. IEEE, 1979, pp. 313–318.

[11] D. Boneh and X. Boyen, "Short signatures without random oracles and the sdh assumption in bilinear groups," *Journal of cryptology*, vol. 21, no. 2, pp. 149–177, 2008.

[12] S. Bowe, A. Gabizon, and I. Miers, "Scalable multi-party computation for zk-snark parameters in the random beacon model," *Cryptology ePrint Archive*, 2017.

[13] D. Chaum and T. P. Pedersen, "Wallet databases with observers," in *Annual International Cryptology Conference*. Springer, 1992, pp. 89–105.

[14] A. Chiesa, Y. Hu, M. Maller, P. Mishra, N. Vesely, and N. Ward, "Marlin: preprocessing zksnarks with universal and updatable srs," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2020, pp. 738–768.

[15] A. Choudhury and A. Patra, "Optimally resilient asynchronous mpc with linear communication complexity," in *Proceedings of the 2015 International Conference on Distributed Computing and Networking*, 2015, pp. 1–10.

[16] ——, "An efficient framework for unconditionally secure multiparty computation," *IEEE Transactions on Information Theory*, vol. 63, no. 1, pp. 428–468, 2016.

[17] R. Cohen, J. Doerner, Y. Kondi, and A. Shelat, "Guaranteed output in $O(\sqrt{n})$ rounds for round-robin sampling protocols," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2022, pp. 241–271.

[18] I. Damgård, Y. Ishai, M. Krøigaard, J. B. Nielsen, and A. Smith, "Scalable multiparty computation with nearly optimal work and resilience," in *Annual International Cryptology Conference*. Springer, 2008, pp. 241–261.

[19] I. Damgård and J. B. Nielsen, "Scalable and unconditionally secure multiparty computation," in *Annual International Cryptology Conference*. Springer, 2007, pp. 572–590.

[20] I. Damgard and R. Thorbek, "Efficient conversion of secret-shared values between different fields," *Cryptology ePrint Archive*, 2008.

[21] S. Das, Z. Xiang, L. Kokoris-Kogias, and L. Ren, "Practical asynchronous high-threshold distributed key generation and distributed polynomial sampling," *Cryptology ePrint Archive*, 2022.

[22] S. Das, T. Yurek, Z. Xiang, A. Miller, L. Kokoris-Kogias, and L. Ren, "Practical asynchronous distributed key generation," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 2518–2534.

[23] Ethereum, "Powers of tau specification," https://github.com/ethereum/kzg-ceremony-specs, 2022.

[24] A. Fiat and A. Shamir, "How to prove yourself: Practical solutions to identification and signature problems," in *Conference on the theory and application of cryptographic techniques*. Springer, 1986, pp. 186–194.

[25] A. Gabizon, "Perpetual powers of tau (for bls381)," https://github.com/arielgabizon/perpetualpowersoftau, 2020.

[26] A. Gabizon, Z. J. Williamson, and O. Ciobotaru, "Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge," *Cryptology ePrint Archive*, 2019.

[27] J. Grigg and S. Bowe, "zkcrypto/pairing," https://github.com/zkcrypto/pairing.

[28] M. Hirt, J. B. Nielsen, and B. Przydatek, "Cryptographic asynchronous multi-party computation with optimal resilience," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2005, pp. 322–340.

[29] ——, "Asynchronous multi-party computation with quadratic communication," in *International Colloquium on Automata, Languages, and Programming*. Springer, 2008, pp. 473–485.

[30] Y. E. Housni, "Benchmarking pairing-friendly elliptic curves libraries," https://hackmd.io/@gnark/eccbench, 2020.

[31] I. Karantaidou and F. Baldimtsi, "Efficient constructions of pairing based accumulators," in *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*. IEEE, 2021, pp. 1–16.

[32] A. Kate, G. M. Zaverucha, and I. Goldberg, "Constant-size commitments to polynomials and their applications," in *International conference on the theory and application of cryptology and information security*. Springer, 2010, pp. 177–194.

[33] M. Kohlweiss, M. Maller, J. Siim, and M. Volkhov, "Snarky ceremonies," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2021, pp. 98–127.

[34] E. Kokoris Kogias, D. Malkhi, and A. Spiegelman, "Asynchronous distributed key generation for computationally-secure randomness, consensus, and threshold signatures." in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1751–1767.

[35] M. Maller, S. Bowe, M. Kohlweiss, and S. Meiklejohn, "Sonic: Zero-knowledge snarks from linear-size universal and updatable structured reference strings," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 2111–2128.

[36] A. Miller and S. Bowe, "Announcing the world's largest multi-party computation ceremony," https://zfnd.org/announcing-the-worlds-largest-multi-party-computation-ceremony/, 2017.

[37] A. Momose and L. Ren, "Optimal communication complexity of authenticated byzantine agreement," in *35th International Symposium on Distributed Computing*, 2021.

[38] K. Nayak, L. Ren, E. Shi, N. H. Vaidya, and Z. Xiang, "Improved extension protocols for byzantine broadcast and agreement," in *34th International Symposium on Distributed Computing, DISC 2020*, 2020.

[39] V. Nikolaenko, S. Ragsdale, J. Bonneau, and D. Boneh, "Powers-of-tau to the people: Decentralizing setup ceremonies," *Cryptology ePrint Archive*, 2022.

[40] C. Papamanthou, E. Shi, and R. Tamassia, "Signatures of correct computation," in *Theory of Cryptography Conference*. Springer, 2013, pp. 222–242.

[41] A. Shamir, "How to share a secret," *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[42] A. Soro and J. Lacan, "Fnt-based reed-solomon erasure codes," in *2010 7th IEEE Consumer Communications and Networking Conference*. IEEE, 2010, pp. 1–5.

[43] S. Srinivasan, A. Chepurnoy, C. Papamanthou, A. Tomescu, and Y. Zhang, "Hyperproofs: Aggregating and maintaining proofs in vector commitments," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, 2022, pp. 3001–3018.

[44] T. Xie, J. Zhang, Y. Zhang, C. Papamanthou, and D. Song, "Libra: Succinct zero-knowledge proofs with optimal prover computation," in *Annual International Cryptology Conference*. Springer, 2019, pp. 733–764.

Appendix A
Zero knowledge Proof of Equality of Discrete Logarithm

Given a group $\mathbb{G}$ with scalar field $\mathbb{F}$ of prime order p, two uniformly random generators $\mathfrak{g}, \mathfrak{h} \in \mathbb{G}$ and a tuple $(\mathfrak{g}, \mathfrak{a}, \mathfrak{h}, \mathfrak{b}) \in \mathbb{G}^4$, a prover $\mathcal{P}$ wants to prove to a probabilistic polynomial time verifier $\mathcal{V}$, in zero-knowledge, the knowledge of a witness $\alpha \in \mathbb{F}$ such that $\mathfrak{a} = \alpha\mathfrak{g}$ and $\mathfrak{b} = \alpha\mathfrak{h}$.

Protocol for equality of discrete logarithm. We use the Chaum-Pedersen Σ-protocol [13], that assumes the hardness of the Discrete Logarithm in $\mathbb{G}$ and proceeds as follows.

1) $\mathcal{P}$ samples a random element $\beta \leftarrow \mathbb{F}$ and sends $(\mathfrak{a}_1, \mathfrak{a}_2)$ to $\mathcal{V}$ where $\mathfrak{a}_1 = \beta\mathfrak{g}$ and $\mathfrak{a}_2 = \beta\mathfrak{g}$.
2) $\mathcal{V}$ sends a challenge $e \leftarrow \mathbb{F}$.
3) $\mathcal{P}$ sends a response $z = \beta - \alpha e$ to $\mathcal{V}$.
4) $\mathcal{V}$ checks whether $\mathfrak{a}_1 = z\mathfrak{g} + e\mathfrak{a}$ and $\mathfrak{a}_2 = z\mathfrak{h} + e\mathfrak{b}$ and accepts if and only if both the equality holds.

This protocol guarantees completeness, knowledge soundness, and zero-knowledge. The knowledge soundness implies that if $\mathcal{P}$ convinces the $\mathcal{V}$ with non-negligible probability, there exists an efficient (polynomial time) extractor that can extract α from $\mathcal{P}$ non-negligible probability.

This above protocol can be made non-interactive in the Random Oracle model using the Fiat-Shamir heuristic [24]. For any given tuple $(\mathfrak{g}, \mathfrak{a}, \mathfrak{h}, \mathfrak{b})$ where $\mathfrak{a} = \alpha\mathfrak{g}$ and $\mathfrak{b} = \alpha\mathfrak{h}$, $\mathsf{dleq.Prove}(\alpha, \mathfrak{g}, \mathfrak{a}, \mathfrak{h}, \mathfrak{b})$ generates the non-interactive zero proof π. The proof π is $O(\kappa)$ bits long. Given a proof π and $(\mathfrak{g}, \mathfrak{a}, \mathfrak{h}, \mathfrak{b})$, $\mathsf{dleq.Verify}(\pi, \mathfrak{g}, \mathfrak{a}, \mathfrak{h}, \mathfrak{b})$ verifies the proof.

Simulating a proof without the secret. We will use programmability of random oracle to generate an convincing NIZK proof without having access to the corresponding secret. Given the tuple $(\mathfrak{g}, \mathfrak{a}, \mathfrak{h}, \mathfrak{b}) \in \mathbb{G}^4$ where $\mathfrak{a} = \alpha\mathfrak{g}$ and $\mathfrak{b} = \alpha\mathfrak{h}$, the simulator works as follows.

1) Sample uniformly random $z, e \in \mathbb{F}$.
2) Compute $\mathfrak{a}_1 = z\mathfrak{g} + e\mathfrak{a}$ and $\mathfrak{a}_2 = z\mathfrak{h} + e\mathfrak{b}$.
3) Program the random oracle such that $\mathsf{RO}(\mathfrak{a}_1, \mathfrak{a}_2) := e$.
4) Output $\pi = (\mathfrak{a}_1, \mathfrak{a}_2, z)$

where $\mathsf{RO}(\cdot)$ denotes query to the random oracle.

Note that the distribution of the simulated proof is identical to the distribution of the proof generated by an honest prover.

Appendix B
Asynchronous Random Double Shairng

Tauron uses $\log q$ double sharing of random values as per the ideal functionality $\mathcal{F}_{\mathsf{Dou}}$ in Figure 8. We make minor modifications to the double sharing protocol of [21] to admit a simulation based security proof. We want to note that the double sharing protocol of [21] is secure in the context they consider in their paper. Let Π_{Dou} be our protocol. Before we describe our modifications, we provide a brief overview of the double sharing protocol of [21].

Double sharing protocol of [21]. The main observation in [21] is that double sharing of a random element z is equivalent to sampling two random polynomials of degree t and $2t$, respectively, that have the same constant term z. Let $\hat{z}(\cdot)$ and $z(\cdot)$ be the polynomials defined as:

$$\hat{z}(x) = z + \hat{z}_1 x + \hat{z}_2 x^2 + \cdots + \hat{z}_t x^t$$
$$z(x) = z + z_1 x + z_2 x^2 + \cdots + z_{2t} x^{2t}$$

where all $z, \hat{z}_j, z_k$ are uniformly random element in $\mathbb{F}$. Each party i then receives $z(i)$ and $\hat{z}(i)$.

Difficulty in proving simulation security of [21]. The protocol of [21] samples the polynomials $\hat{z}(\cdot)$ and $z(\cdot)$ using different approaches. Their protocol has four phases: Sharing, Agreement and Randomness Extraction, and Key Derivation. They sample the polynomial $z(\cdot)$ in a way such that $\mathcal{A}$ learns no information about $z(i)$ for adversarial parties i until the Agreement phase finishes at least one honest party. However, for polynomial $\hat{z}(\cdot)$, $\mathcal{A}$ learns $\hat{z}(i)$ during the sharing phase.

Functionality $\mathcal{F}_{\mathsf{Dou}}$

- Let $\mathbb{G}$ be a elliptic curve group with scalar field $\mathbb{F}$ and let $\mathfrak{g}$ be a uniformly random generator of $\mathbb{G}$.
- Wait for $\mathcal{C}$, the set of adversarial parties and the $(\mathtt{start}, q)$ message from $\mathcal{A}$.
- Wait for $(\mathtt{init}, q)$ from all honest parties.
- Sample a uniformly random element $z \in \mathbb{F}$. Generate $(n, t+1)$ and $(n, 2t+1)$ shares of z, denoted with $[\![z]\!]$ and $[\![z]\!]^{2t}$, respecitvely.
- Compute $[\![z]\!]\mathfrak{g}$ and $[\![z]\!]^{2t}\mathfrak{g}$ and send the tuple $(\mathfrak{g}, [\![z]\!]_i, [\![z]\!]^{2t}_i, [\![z]\!]\mathfrak{g}, [\![z]\!]^{2t}\mathfrak{g})$ to party i.

Figure 8: The functionality for random double sharing

Simulator $\mathcal{S}_{\mathsf{Dou}}$

Inputs. Set of adversarial parties $\mathcal{C}$, $[\![z]\!]_i, [\![z]\!]^{2t}_i$ for all $i \in \mathcal{C}$, and threshold public keys $[\![z]\!]\mathfrak{g}, [\![z]\!]^{2t}\mathfrak{g}$.

1) Sample a random $\alpha \in \mathbb{F}$ and let $\mathfrak{h} = \alpha\mathfrak{g}$.
2) Let $\mathcal{H} = [n] \setminus \mathcal{C}$ be the set of emulated honest parties. For each $j \in \mathcal{H}$, run the Sharing and Agreement phase as per the protocol specification.
3) Sample two random polynomials $\hat{z}(\cdot)$ and $z(\cdot)$ of degree t and $2t$, respectively, such that:
   - $\hat{z}(0)\mathfrak{g} = z(0)\mathfrak{g} = z\mathfrak{g}$
   - $\hat{z}(i) = [\![z]\!]_i$ and $z(i) = [\![z]\!]^{2t}_i$, for each $i \in \mathcal{C}$,
   - $[\![z]\!]_j\mathfrak{g} = \hat{z}(j)\mathfrak{g}$ and $[\![z]\!]^{2t}_j\mathfrak{g} = z(j)\mathfrak{g}$
4) Use α and the extractability of the Sharing phase to emulate a Randomness extraction phase such that each malicious party $i \in \mathcal{C}$ outputs $[\![z]\!]_i$ and $[\![z]\!]^{2t}_i$ as its random double share.
5) Use $\mathcal{S}_{\mathsf{dleq}}$ to compute the NIZK proofs required during the key derivation phase.

Figure 9: Simulator for the protocol for random double sharing.

Our approach. At a high-level, we modify the protocol such that both $\hat{z}(\cdot)$ and $z(\cdot)$ are sampled such that $\mathcal{A}$ learns no information its shares on both $\hat{z}(\cdot)$ and $z(\cdot)$ before the agreement phase terminates. This enables the simulator $\mathcal{S}_{\mathsf{Dou}}$ to match the shares of the adversarial parties with the shares received from the $\mathcal{F}_{\mathsf{Dou}}$. Specifically, we make the following changes:

1) During Sharing phase, each party secret shares three random values instead of two as in [21].
2) During the Randomness Extraction phase, in addition to $[\![z]\!]$ and $[\![z_j]\!]$ for $j \in [2t]$, parties also generate $[\![\hat{z}_k]\!]$ for each $k \in [t]$ using the random extractor. Parties assist each party i to additionally compute $\hat{z}(i)$.
3) During the Key Derivation phase, parties additionally output $\hat{z}(i)\mathfrak{g}$ for all $i \in [n]$.

Analysis. We describe the simulator $\mathcal{S}_{\mathsf{Dou}}$ in Figure 9, that on inputs adversarial double shares of a random values from $\mathcal{F}_{\mathsf{Dou}}$ simulates the protocol Π_{Dou}. The indistinguishability of the $\mathcal{A}$'s view follows from perfect hiding of the Sharing phase and perfect zero-knowledge of the dleq protocol.

Appendix C
Asynchronous Distributed Key Generation

Functionality $\mathcal{F}_{\mathsf{ADKG}}$

- Let $n \geq 3t+1$ be the total number of parties. Let $\mathbb{G}$ be a elliptic curve group with a random generator $\mathfrak{g}$.
- Wait for $(d, \mathcal{C})$, the degree of the secret sharing and the set of adversarial parties and the $(\texttt{start}, q)$ message from $\mathcal{A}$. Check that $d > |\mathcal{C}|$ and $|\mathcal{C}| \leq t$.
- Wait for $(\texttt{init}, q)$ from all honest parties.
- Sample a uniformly random element $z \in \mathbb{F}$. Generate $(n, d+1)$ shares of z, denoted with $[\![z]\!]$.
- Compute $[\![z]\!]\mathfrak{g}$ and send $(\mathfrak{g}, z\mathfrak{g}, [\![z]\!]_i, [\![z]\!]\mathfrak{g})$ to party i.

Figure 10: Asynchronous Distributed Key Generation functionality