ETHDKG: Distributed Key Generation with Ethereum Smart Contracts

Philipp Schindler
SBA Research

Aljosha Judmayer
SBA Research

Nicholas Stifter
SBA Research, TU Wien

Edgar Weippl
SBA Research, TU Wien

Abstract

Distributed key generation (DKG) is a fundamental building block for a variety of cryptographic schemes and protocols, such as threshold cryptography, multi-party coin tossing schemes, public randomness beacons and consensus protocols. More recently, the surge in interest for blockchain technologies, and in particular the quest for developing scalable protocol designs, has renewed and strengthened the need for efficient and practical DKG schemes. Surprisingly, given the broad range of applications and available body of research, fully functional and readily available DKG protocol implementations still remain limited. We hereby aim to close this gap by presenting an open source, fully functional, well documented, and economically viable DKG implementation[1] that employs Ethereum's smart contract platform as a communication layer. The efficiency and practicability of our protocol is demonstrated through the deployment and successful execution of a DKG contract in the Ropsten testnet. Given the current Ethereum block gas limit, it is possible to support up to 256 participants, while still ensuring that the key generation process can be verified at smart contract level. Further, we present a generalization of our underlying DKG protocol that is suitable for distributed generation of keys for discrete logarithm based cryptosystems.

1 Introduction

Distributed key generation (DKG) protocols serve as a key building block for threshold cryptography. The goal of a DKG scheme is to agree on a common secret/public keypair such that the secret key is shared among a set of n participants. Only a subset of $t+1 \leq n$ parties can use or reveal the generated secret key, while t collaborating parties cannot learn any information about it. In this regard DKG is related to secret sharing protocols, as first introduced by Shamir [39] and Blakley [4]. However, in contrast to secret sharing, DKG protocols do not rely on a (trusted) dealer which generates, knows and distributes the secret key, and hence avoid this single point of failure. Instead, the keypair is generated using a multi-party computation in a way that no single party learns the secret that is being shared.

Distributed key generation is a topic that has been studied and discussed for over two decades [6, 21, 22, 26, 27, 30, 33]. However, the extensive body of literature is currently not matched by a single clear, succinct, and practical protocol design template that reflects the state of the art and leverages on recent technical developments such as distributed ledgers. Moreover, real-world open source implementations of DKG protocols are still rare, and often not well documented.

We aim to close this gap by providing and evaluating a lightweight, scalable, and well-documented protocol design and open source implementation of a DKG protocol. Our design is based on the Joint-Feldman DKG protocol [21] and incorporates the enhancements proposed by Neji et al. [30] to address biasing attacks [21], without requiring two distinct secret sharing rounds. Additionally, we describe and implement a new mechanism that handles disputes during the protocol execution more efficiently. The resulting protocol design is described in its generality for any discrete logarithm based cryptosystem, and we demonstrate that our protocol improvements enable the verification of the key generation process within Ethereum, and similar smart contract platforms, in a cost effective manner. Our evaluation results show that the total costs incurred by each participant in a scenario with $n = 256$ parties vary between 0.67 USD and 2.46 USD, depending on the behavior of the adversary.

Augmenting our DKG protocol with the capabilities provided by smart contracts enables us to dynamically define the set of participating entities, incentivize participation, and penalize adversarial behavior. Further, we are able to ensure that any security deposits provided by participants following the protocol rules always remain safe, even if the DKG protocol itself is executed by a majority of adversarial participants.

[1] The source code, documentation, and logs of a successful execution in the Ropsten testnet are publicly available at [link blinded for review].

1.1 Structure of this Paper

We continue this paper by describing application scenarios for our DKG protocol in section 2. Section 3 introduces and compares related work to our approach. We describe our system model, including assumptions concerning the network infrastructure, the capabilities of the adversary as well as the security properties expected from DKG protocols, in section 4. Our generalized protocol design for discrete logarithm based cryptosystems is presented and analysed in sections 5 and 6. Section 7 provides implementation specific details, while section 8 describes our evaluation results. Finally, we discuss and conclude the paper in sections 9 and 10.

2 Application Scenarios

We outline several application domains for DKG that have garnered increased interest, in particular as a result of the rising popularity of cryptocurrencies and blockchain technologies, and describe how our DKG protocol can prove beneficial in this context.

Consensus Protocols Recent improvement proposals for blockchain and other distributed ledger protocols, e.g. randomized Byzantine fault tolerant (BFT) consensus protocols such as Honeybadger [29], the Dfinity blockchain protocol [25] or Calypso [28], are increasingly relying on threshold cryptography as one of their core building blocks. In these scenarios, having to rely on a single centralized entity for the necessary protocol setup would likely contradict the overall design goals of avoiding single points of trust and failure. Hence, DKG can provide improved guarantees and our protocol seems particularly suited for such scenarios, as it can be executed on the readily available Ethereum platform, provides flexible means for registration of the participants, and tolerates faulty or adversarial behavior of any minority of participants.

Custodian and Escrow Schemes An Ethereum related use case of our DKG includes multi-signature wallet (contracts) or exchanges, where the collaboration of multiple stakeholders is required to legitimize actions, such as performing transactions. Using our approach, the stakeholders can set up a shared keypair to be used within a threshold signature scheme such as Boneh-Lynn-Shacham (BLS) [5,7,8]. The BLS public key is registered within the wallet contract and signature validation within the smart contract is possible to verify the authenticity of initiated actions. As signature aggregation can be performed off-chain, the number of on-chain computational steps for the corresponding verification procedure does not depend on the number of involved stakeholders.

Randomness Beacon Our smart contract based DKG protocol can also be used to bootstrap a variety of interesting applications on the Ethereum platform itself. A candidate example is a source of publicly-verifiable, bias-resistant and unpredictable randomness, or in short, a randomness beacon [34]. By levering the security and uniqueness properties of BLS threshold signatures, the construction of a randomness beacon follows naturally. Within the Ethereum platform, such trustworthy randomness beacons are particularly useful, as there are currently no built-in mechanisms for deriving randomness with the aforementioned characteristics. Consequently, Ethereum smart contracts largely rely on less secure sources of randomness (such as the block hash) or even depend on trusted third parties. Beyond an application in Ethereum smart contracts, unpredictable bias-resistant public randomness also plays an important role in a broad range of fields, including proof-of-stake and sharding protocols, privacy preserving messaging services, e-voting protocols, as well as gambling and lottery services [38]. Additionally, consensus protocols such as Cachin et al. [10] and Dfinity [25] rely on threshold cryptography to build a *common coin* as a core component of their protocol designs.

Threshold and Time-Lock Encryption We can also envision our protocol to be employed for setting up the keypairs required for threshold public key encryption schemes [17]. In such a scenario, a client could encrypt a message under the generated master public key such that the decryption of the message requires the collaboration of a threshold of participants from the DKG. Furthermore, utilizing the capabilities of threshold encryption schemes in combination with a cryptocurrency supporting smart contracts, enables the construction of time-lock encryption protocols [37]. Here, an incentive structure, and security deposits provided by the stakeholders during the setup process with the DKG protocol, ensure economic guarantees for the decryption of messages at specified points in time.

3 Related Work

The first protocol for DKG was introduced by Pedersen [33] in 1991, and was subsequently built upon within a wide range of publications in the field of threshold cryptography. A popular variant is the so called Joint-Feldman DKG protocol, introduced by Gennaro et al. [21] as a simplification of Pedersen's work. The core idea of the Pedersen (and the Joint-Feldman) protocol is that each party executes Feldman's [19] verifiable secret sharing (VSS) protocol, acting as a dealer in order to share a randomly chosen secret among all parties. After a verification step, ensuring the participants shared their secrets correctly, the resulting group private key is defined by the sum of the properly shared secrets. This private key is unknown to the individual participants, but may be obtained by a collaborating group of parties. The corresponding public key can be computed using the commitments published during the sharing phase with Feldman's VSS protocol and is the public result of executing the DKG protocol.

However, as described in great detail by the works of Gennaro et al. [21, 22], keys generated using (a wide range of variants of) the Pedersen protocol, are not guaranteed to be uniformly distributed over the respective keyspace. An adversary can bias bits of the resulting key by selectively denouncing one or more of the parties it controls. Consequently, the set of parties which properly shared their secrets, and thus define the resulting key, is influenced as the denounced parties are excluded. The critical[2] issue in this case is that honest parties have provided all the information required to compute the resulting public key *before* agreement on the set of shares that are used to create the master key is reached, allowing the adversary to influence the final outcome.

Gennaro et al. [21] presents mitigation strategies to protect against these kind of attacks. However, their approach adds complexity as it requires an additional secret sharing step using Pedersen VSS protocol [32]. Canetti et al. [12] extend the solution from Gennaro et al. to cope with adaptive adversaries, which may corrupt parties based on prior knowledge gathered during the protocol execution. More recently, Neji et al. [30] describe a different countermeasure which we adopt in this paper, avoiding these drawbacks.

Kate and Goldberg [26] were the first to study DKG in an asynchronous communication model, whereas previously synchronous message delivery was assumed. In order to support these weaker assumptions on the communication model, they require a network of $n \geq 3t+2f+1$ participants, out of which t are controlled by the adversary and thus considered Byzantine and f parties may fail in the crash-stop model. This is in contrast to works in the style of Gennaro et al. and our protocol, which require synchrony but can tolerate ($n \geq 2t+1$) Byzantine adversaries. In a subsequent extension of their work [27], Kate et al. provide an implementation, tested with up to 70 parties distributed over multiple continents. A crucial distinction between Kate and Goldberg's work and the approach followed by Gennaro et al. and also this paper, is that that the former also implement a Byzantine agreement protocol alongside the DKG, whereas the consensus protocol is not part of the DKG specification in the latter. We outline the advantages and drawbacks of both design decisions in our discussion (see section 9).

To the best of our knowledge, the DKG protocol developed by the Orbs Network team [31] is the only publicly available protocol targeting a similar deployment scenario, namely, an implementation of a DKG protocol using the Ethereum platform. However, the presented prototypical implementation appears to be incomplete and has not been updated since 5th August, 2018. A scientific publication outlining the details of the protocol is also not available at the time of writing. Furthermore, this approach, in comparison to the works of Gennaro et al., Kate and Goldberg, and our work, fails to guarantee liveness under adversarial behavior, as it requires a protocol restart even if a single adversarial participant sends an invalid share.

4 System Model

Using our protocol, a set of n participants $\mathcal{P} = \{P_1, P_2, ..., P_n\}$ wish to jointly generate a master secret/public keypair of the form $mpk = g^{msk}$ for a discrete logarithm based threshold cryptosystem. We use g and h to denote two independently[3] selected generators of the group $\mathbb{G}_q$ with prime order q and assume that computing discrete logarithms in $\mathbb{G}_q$ is hard. The master public key mpk is the (public) output of the protocol. The corresponding (virtual) secret key msk is shared among the participants, and may be obtained by pooling the shares from $t+1$ collaborating parties. Depending on the use case scenario, it may not be desirable or even necessary to ever obtain msk. For instance, by employing BLS threshold signatures [8], a signature verifying under the master public key mpk can be obtained by aggregating signature shares without recovering msk first.

4.1 Communication Model

We assume all parties can monitor and broadcast messages on a shared public and authenticated communication channel. Further, all participants are in agreement on a common view and ordering of these broadcast messages. We assume synchrony in the sense that, any message that is broadcast by a participant during some protocol phase is received by all other parties before the next phase starts. In this regard, our communication model is closely related to the notion of public bulletin boards [16].

In contrast to Gennaro et al. [21], we do not consider pairwise private communication links between parties. Instead, we assume that each participant $P_i \in \mathcal{P}$ generates a fresh secret/public keypair $\langle sk_i, pk_i \rangle$ of the form $pk_i = g^{sk_i}$ prior to the protocol start and knows the public keys of all other participants. These keys are used to derive keys for a symmetric encryption algorithm, which is used to ensure secrecy of broadcast messages during the sharing phase of the protocol. Instead of assuming a priori knowledge of the other parties' keys, an additional registration phase (see section 7.6) to exchange public keys may also be used.

Blockchain protocols, which allow inclusion of arbitrary data, and other BFT state machine replication and distributed ledger protocols present suitable candidates for such communication channels. In practice, we leverage the Ethereum blockchain as a public authenticated communication channel and consensus protocol, where agreement on message ordering is ensured through the common prefix property [20].

[2] We refer to the works of Gennaro et al. [22,23] for an in-depth discussion of the implications of a non-uniform distribution.

[3] I.e. the discrete logarithm $dlog_g(h)$ between g and h is unknown.

Together with the client software, which enforces appropriate stabilization times to ensure agreement with high probability, the desired guarantees can be achieved. We refer to section 7.1 for additional details on how we instantiate such a communication channel.

4.2 Adversarial Model

To ensure secrecy of the generated secret key *msk*, we assume that an adversary controls at most t participants, whereas a collaboration of $t+1$ participants is required to derive *msk*. A node controlled by the adversary may deviate arbitrarily from the specified protocol. We consider an *adaptive* adversary, in the sense that it can decide which parties to corrupt based on prior observations. However, the adversary is *not mobile*, once a party is corrupted it is considered compromised for the entire protocol execution. To guarantee both, secrecy of the generated key as well as liveness, i.e. that the protocol completes successfully, the adversary must not control more than $t < n/2$ parties. These are the optimal bounds one can hope to achieve in our setting [21].

4.3 Security Properties

In the following, we reiterate on the security properties we aim for and expect from a DKG protocol. Hereby, we follow the definitions given by Gennaro et al. [21] and Neji et al. [30] for *correctness* and *secrecy*. The *uniformity* property highlights a shortcoming identified by Gennaro et al. [21] that was not covered by the original Joint-Feldman protocol. Because recent DKG implementations appear to not consider this property, e.g., the Ethereum-based DKG implementation in [31], we use a distinct category to further emphasize this characteristic. *Robustness* ensures that a subset of parties, which want to recover the master secret key, is able to do so under adversarial influence. The definitions of uniformity and robustness follow the correctness definitions C3 and C1' from Gennaro et al. We also add a definition for *liveness*, which was not explicitly stated in Gennaro et al.'s work.

Correctness

(C1) All sets of $t+1$ correct key shares define the same unique master secret key *msk*.

(C2) All honest parties agree on the common value of the master public key $mpk = h^{msk}$.

Uniformity The master secret key *msk* is uniformly distributed in $\mathbb{Z}_q$, and hence the master public key *mpk* is uniformly distributed in $\mathbb{G}_q$.

Robustness There is an efficient procedure that, on input of the public information of the DKG protocol and n submitted shares, outputs *msk*, even if up to t invalid shares have been submitted by malicious or faulty participants.

Liveness An adversary controlling up to $n-t-1$ nodes cannot prevent the protocol from completing successfully.

5 Protocol Description

In this section, we present our generalized DKG protocol design for discrete logarithm based cryptosystems. We start by giving a brief overview of our three consecutive protocol phases, and then describe each phase in detail in sections 5.1, 5.2 and 5.3. For implementation specific details we refer to section 7.

Sharing Phase During the first phase, each participant in $P_i \in \mathcal{P}$ selects a randomly chosen secret $s_i \in_R \mathbb{Z}_q$ and subsequently uses Feldman's VSS to share this secret among all parties, such that $t+1$ collaborating parties can recover s_i, in case a malicious party withholds the required information during the key derivation phase. The verification procedure of Feldman's protocol enables the parties to check that received shares are indeed valid.

Dispute Phase During the dispute phase, each party that received one or more invalid shares in the previous phase uses a non-interactive proof technique to convince other parties about the fact that the issuer violated the protocol.

Key Derivation Phase At the beginning of the last phase, a set of qualified parties $\mathcal{Q} \subseteq \mathcal{P}$ is formed. A party P_i is part of $\mathcal{Q}$ if and only if it (i) broadcasted the required information during the sharing phase and (ii) no party broadcasted a valid dispute against P_i during the dispute phase. In other words, the set $\mathcal{Q}$ contains all parties which correctly shared their secret and should thus contribute to form the master keypair $\langle msk, mpk \rangle$. Finally, for all parties $P_i \in \mathcal{Q}$ the values h^{s_i}, related to the randomly chosen secrets s_i, are either revealed or recovered and used to derive the master public key *mpk*. Using Lagrange interpolation, *msk* can be computed after pooling the shares from $t+1$ parties. However, depending on the use case scenario, it may not be desirable or necessary to ever obtain *msk*.

5.1 Sharing Phase

Share Generation At the beginning of the sharing phase, each party $P_i \in \mathcal{P}$ executes the first step of the Joint-Feldman DKG protocol [21]. In order to share a randomly chosen secret $s_i \in_R \mathbb{Z}_q$ among all[4] registered parties, P_i acts as the dealer in a (n,t) Feldman VSS protocol [19]. For this purpose it picks a secret polynomial $f_i : \mathbb{Z}_q \to \mathbb{Z}_q$ with coefficients

[4] For ease of exposition, we assume that P_i also provides one share for itself.

$c_{i0} = s_i$ and $c_{i1}, c_{i2}, \ldots, c_{it}$ drawn uniformly at random from $\mathbb{Z}_q$:

$$f_i(x) = c_{i0} + c_{i1}x + c_{i2}x^2 + \ldots + c_{it}x^t \pmod{q} \tag{1}$$

Then P_i computes the shares $s_{i \to j} = f_i(j)$ for all $P_j \in \mathcal{P}$, and the commitments $C_{i0} = g^{c_{i0}}, C_{i1} = g^{c_{i1}}, \ldots, C_{it} = g^{c_{it}}$ to the coefficients of $f_i(\cdot)$. These commitments are used in the verification process for the shares and implicitly define P_i's public polynomial $F_i : \mathbb{Z}_q \to \mathbb{G}_q$:

$$F_i(x) = C_{i0} \cdot C_{i1}^{x} \cdot C_{i2}^{x^2} \cdot \ldots \cdot C_{it}^{x^t} \tag{2}$$

Share Transmission Next, each P_i has to securely send its shares $s_{i \to j}$ to all other parties $P_j \in \mathcal{P}$. Contrary to the original description of the Joint-Feldman DKG, we do not assume access to private communication channels between parties, but rather realize the secure sending of the shares using encryption over our public broadcast channel. We use a symmetric key encryption algorithm $\mathsf{Enc}_{k_{ij}}(\cdot)$ to ensure secrecy of a sent share from P_i to P_j. The corresponding encryption key k_{ij} can be derived non-interactively by both parties:

$$k_{ij} = {pk_j}^{sk_i} = {pk_i}^{sk_j} = g^{sk_i sk_j} \tag{3}$$

Notice that this approach is inspired by the techniques used in the Diffie Hellman key exchange protocol [36] and the ElGamal encryption scheme [18].

Finally, P_i broadcasts the encrypted shares $\overline{s_{i \to j}} = \mathsf{Enc}_{k_{ij}}(s_{i \to j})$ for all $i \neq j$ as well as the commitments $C_{i0}, C_{i1}, \ldots, C_{it}$ from Feldman's VSS. Each party P_j monitors the communication channel for messages broadcasted by other participants. Upon receiving encrypted shares and commitments from P_i, P_j decrypts its share to obtain $s_{i \to j} = \mathsf{Dec}_{k_{ij}}(\overline{s_{i \to j}})$.

Share Verification P_j employs the verification procedure of Feldman's VSS to check the validity of each share $s_{i \to j}$. A share is valid if and only if the following share verification condition holds:

$$g^{s_{i \to j}} = F_i(j) \tag{4}$$

In case $s_{i \to j}$ is found invalid, further actions are required in the dispute phase. As P_i only expects to receive a single message from each party, only the first message is processed, any additional messages from the same sender are ignored. In our smart contract based implementation (see section 7), the smart contract itself ensures that parties can only broadcast a single message during the sharing phase.

5.2 Dispute Phase

In case a party P_j notices that it received an invalid share $s_{i \to j}$ from P_i in the previous phase, P_j must broadcast a dispute claim in order to ensure that P_i is excluded from further steps of the protocol execution. Intuitively, P_i must be excluded because its secret s_i may not be recoverable by a collaboration of $t+1$ correct parties.

In the original description of the Joint-Feldman DKG protocol, an adversarial P_j can always issue an (unsupported) claim stating that it received an invalid share from a correct P_i, requiring P_i to prove adherence to the protocol rules. We flip this notion in the sense that it is P_j's obligation to show that P_i indeed violated the protocol. To accomplish this we use a non-interactive proof technique described below, and can consequently reduce the required number of interactions between parties.

Issuing a Dispute Claim The key idea how P_j is able to prove that P_i provided an invalid share $s_{i \to j}$ is to publish the key k_{ij} used for encryption and decryption of the share. Using this key, other parties are able to decrypt the previously distributed share $\overline{s_{i \to j}}$ and can, in the same way as P_j did, verify that $s_{i \to j}$ is indeed invalid. To ensure that an adversarial P_j cannot just publish an invalid key k'_{ij}, which would again lead to a false accusation of P_i, it is required that P_j proves the correctness of k_{ij}. We use a common non-interactive zero-knowledge (NIZK) proof technique for showing the equality of the two discrete logarithms [11,15] to show the correctness of k_{ij}. The corresponding proving and verification procedures are denoted by $\mathsf{DLEQ}(x_1, y_1, x_2, y_2, \alpha)$ and $\mathsf{DLEQ\text{-}verify}(x_1, y_1, x_2, y_2, \pi)$.

Procedure 1: $\mathsf{DLEQ}(x_1, y_1, x_2, y_2, \alpha)$.
To show that $dlog_{x_1}(y_1) = dlog_{x_2}(y_2)$ holds without revealing the discrete logarithm α, a prover proceeds as follows:

1. compute $t_1 = x_1^w$ adding $t_2 = x_2^w$ for $w \in_R \mathbb{Z}_q$
2. compute $c = \mathsf{H}(x_1, y_1, x_2, y_2, t_1, t_2)$
3. compute $r = w - \alpha c \pmod{q}$
4. output $\pi = \langle c, r \rangle$

Instantiating the above procedure, P_j can prove the correctness of the decryption key k_{ij} by providing $\pi(k_{ij}) = \mathsf{DLEQ}(g, pk_j, pk_i, k_{ij}, sk_j)$ in addition to k_{ij}.

Verifying a Dispute Claim Upon receiving a dispute claim $\langle k_{ij}, \pi(k_{ij}) \rangle$ against P_i, issued by P_j, one can use $\mathsf{DLEQ\text{-}verify}(g, pk_j, pk_i, k_{ij}, \pi(k_{ij}))$ to check the validity of the received key k_{ij}.

Procedure 2: DLEQ-verify$(x_1, y_1, x_2, y_2, \pi)$.
To check the correctness of a proof $\pi = \langle c, r \rangle$, showing that $dlog_{x_1}(y_1) = dlog_{x_2}(y_2)$ holds, a verifier proceeds as follows:

1. compute $t'_1 = x_1^r y_1^c$ and $t'_2 = x_2^r y_2^c$
2. output VALID if $c = \mathsf{H}(x_1, y_1, x_2, y_2, t'_1, t'_2)$ holds output INVALID otherwise

If the key is found invalid, the dispute claim is invalid. Otherwise, the verification procedure continues by decrypting the corresponding share $s_{i\to j} = \mathsf{Dec}_{k_{ij}}(\overline{s_{i\to j}})$ and checking its correctness according to the share verification condition specified in equation 4. The dispute is valid if and only if k_{ij} is found valid but the verification condition does not hold.

The protocol ensures that: (i) In case a correct participant received an invalid share from another party, the share issuer is considered disqualified by all (correct) parties at the end of the dispute phase. (ii) An adversary cannot wrongly accuse any correct party of providing it with an invalid share. (iii) The adversary does not gain any additional information when a party P_j reveals the values k_{ij} and $\pi(k_{ij})$, because the adversary can always compute (and therefore publish) $k_{ij} = {pk_j}^{sk_i}$ using P_i's secret key, and the NIZK proof $\pi(k_{ij})$ does not reveal additional information apart from the correctness of the statement.

5.3 Key Derivation

Deriving the Set of Qualified Nodes The first step in the key derivation phase is determining the set of qualified parties $\mathcal{Q} \subseteq \mathcal{P}$, describing which parties should contribute to the resulting keypair $\langle msk, mpk \rangle$. If we recall the current protocol state at the beginning of the key derivation phase, we observe that each $P_i \in \mathcal{P}$ has either:

1. *correctly* shared its secret s_i with all other parties.
2. *incorrectly* shared its secret s_i.
3. did not share its secret s_i at all.

We say a secret was correctly shared by P_i, if and only if no valid dispute claim against P_i was filed during the dispute phase. Parties which incorrectly shared their secrets, or did not share their secrets at all, are disqualified and excluded from the upcomming protocol steps. The remaining parties form the set $\mathcal{Q}$. In other words, a node $P_i \in P$ is only part of $\mathcal{Q}$ if (i) it published the values $C_{i0}, C_{i1}, \ldots, C_{it}$ and $\overline{s_{i\to j}}$ for all $i \neq j$ during the sharing phase and (ii) no node P_j filed a valid dispute against P_i during the dispute phase.

Bias when Computing the Keys Directly Using this definition of the set $\mathcal{Q}$, the resulting group public key mpk could be derived by following the description of the Joint-Feldman protocol:

$$mpk = \prod_{P_i \in \mathcal{Q}} C_{i0} = \prod_{P_i \in \mathcal{Q}} g^{s_i} \tag{5}$$

However, as described in great detail by the works of Gennaro et al. [21, 22], the above approach does not ensure that the resulting keypair is uniformly distributed. An adversary can bias bits of the resulting key by selectively denouncing one or more of its nodes, which influences the set $\mathcal{Q}$ and thus the resulting key. The critical[5] issue here is, that all information required to compute the resulting public key is known to the adversary *before* the set $\mathcal{Q}$ is fixed.

Protection against Biasing of the Generated Keys We adopt a recent countermeasure described by Neji et al. [30] to ensure the resulting key is uniformly distributed. The key idea to ensure uniformity is to instead compute mpk as follows:

$$mpk = \prod_{P_i \in \mathcal{Q}} h^{s_i} \tag{6}$$

Here, h is used to denote an additional generator of the group $\mathbb{G}_q$, such that $dlog_g(h)$ is unknown. The required values h^{s_i} used to compute mpk are published by the parties in $\mathcal{Q}$, whereas each P_i shows the correspondence between the values h^{s_i} and $C_{i0} = g^{s_i}$ using the NIZK proof $\pi(h^{s_i}) = \mathsf{DLEQ}(g, g^{s_i}, h, h^{s_i}, s_i)$ as introduced in section 5.2. In case any (adversarial) party $P_i \in \mathcal{Q}$ does not reveal its value h^{s_i} and a valid proof $\pi(h^{s_i})$ by the end of the key derivation phase, a set of $t+1$ correct parties is always able to use the recovery procedure of Feldman's VSS to obtain s_i and consequently h^{s_i} anyway. Without loss of generally, let $\mathcal{R} \subseteq \mathcal{Q}$ denote a set of $t+1$ correct parties. Then, s_i is obtained via Lagrange interpolation:

$$s_i = \sum_{P_j \in \mathcal{R}} s_{i\to j} \prod_{\substack{P_k \in \mathcal{R} \\ j \neq k}} \frac{k}{k-j} \tag{7}$$

Deriving the Keys Finally, the common master public key mpk can be derived as specified in equation 6 using the published or recovered values $h^{s_i} \mid P_i \in \mathcal{Q}$. Additionally, each $P_j \in \mathcal{Q}$ can compute its individual group keypair $\langle gsk_j, gpk_j \rangle$:

$$gsk_j = \sum_{P_i \in \mathcal{Q}} s_{i\to j} \qquad gpk_j = h^{gsk_j} \tag{8}$$

In order to enable a third party to verify gpk_j, P_j provides the values g^{gsk_j} as well as a correctness proof $\mathsf{DLEQ}(g, g^{gsk_j}, h, gpk_j, gsk_j)$. The verifier accepts gpk_j as valid if checking of the proof via DLEQ-verify$(\cdot)$ succeeds,

[5] See [22, 23] for an in-depth discussion on the implications of non-uniform distribution.

and the verification of g^{gsk_j} using the previously committed public polynomials is successful:

$$g^{gsk_j} = \prod_{P_i \in \mathcal{Q}} F_i(j) \tag{9}$$

The corresponding master secret key msk is shared among all nodes in $\mathcal{Q}$ and can be obtained as follows:

$$msk = \sum_{P_i \in \mathcal{Q}} s_i \tag{10}$$

In case P_i does not reveal its secret s_i, it can always be computed by $t+1$ collaborating parties, because each $P_i \in \mathcal{Q}$ has correctly shared s_i among the parties during the first protocol phase. Alternatively, a set of $t+1$ collaborating parties, denoted by $\mathcal{R}$, can also derive the master secret key msk via Lagrange interpolation from their group secret keys:

$$msk = \sum_{P_j \in \mathcal{R}} gsk_j \prod_{\substack{P_k \in \mathcal{R} \\ j \neq k}} \frac{k}{k-j} \tag{11}$$

However, for many threshold cryptographic applications msk might never be computed at a single location. Considering, e.g. BLS threshold signatures, $t+1$ collaborating parties might produce a signature σ on message m which verifies under the public key mpk. For this purpose, each of these parties P_j uses its individual group signing key gsk_j to issue a partial signature for m, which upon aggregation form σ. There is no need to compute the master secret key msk in order to issue the signature in this scenario.

6 Security Analysis

We omit a detailed analysis of the guarantees in regard to *correctness* (C1, C2) and *uniformity* (U1) in this paper. The corresponding security proofs have been provided by the works of Gennaro et al. [21] and Neji et al. [30] and directly apply to our protocol.

Secrecy In order to apply the original security proofs for secrecy, we proceed by showing that the dispute process we introduce as alternative to the steps described by Neji et al. [30] does not provide the adversary with any additional information. Specifically, any information a correct node P_i secretly transfers to another correct node P_j must remain hidden from the adversary. The critical point, where this information is exchanged, is the share transmission step (see section 5.1). Here a correct party P_i always encrypts the share $s_{i \to j}$ it sends to P_j, using a symmetric key encryption algorithm $\mathsf{Enc}_{k_{ij}}(\cdot)$. Under the computational Diffie-Hellman assumption, the corresponding shared key k_{ij} can only be derived using secret information sk_i or sk_j from node P_i or P_j. However, neither P_i nor P_j reveal this information or k_{ij} itself during the protocol execution if they are both honest.

The only point in time a honest node P_i would publish k_{ij} and the corresponding correctness proof $\pi(k_{ij}) = \mathsf{DLEQ}(g, pk_i, pk_j, k_{ij}, sk_i)$ is during the process of issuing a dispute claim (section 5.2). A correct P_i, however, only issues such a claim if P_j provided an invalid share during the distribution phase – violating the assumption that P_j is correct. If we consider an adversarial P_j instead, no additional information is revealed when P_i publishes k_{ij}, as the adversary was already able to derive $k_{ij} = {pk_i}^{sk_j}$ (and thus obtain $s_{i \to j}$). Furthermore, e.g. as outlined by Camenisch and Stadler [11], the NIZK proof $\pi(k_{ij})$ does not reveal any information in addition to correctness of k_{ij}, in particular does not reveal any information about sk_i.

Robustness Robustness requires an efficient procedure, that recovers the master secret key msk from a set of at least $t+1$ correct shares. However, this set may additionally contain up to t invalid shares provided by the adversary. We obtain such a procedure, by first checking the validity of a provided share gsk_i using the verification condition specified in equation 9. Then we use Lagrange interpolation to compute msk from any set of $t+1$ valid shares (see equation 11)

Liveness In our synchronous system model, the protocol always reaches the beginning of the key derivation phase, as the sharing and dispute phases always end after a fixed amount of steps (the respective number of blocks per phase). Consequently, the completion of the key derivation phase (and thus the completion of the protocol), depends on the nodes' ability to gather all the information required to compute mpk from the values h^{s_i} provided by all $P_i \in \mathcal{Q}$. Each correct node in the set of qualified nodes $\mathcal{Q}$, publishes this value at the beginning of the phase. However, up to t adversarial nodes, which completed the sharing and dispute phase successfully, and are thus part of $\mathcal{Q}$, might not reveal the respective values. In this case, the correct parties obtain all missing values h^{s_i} by recovering s_i using Lagrange interpolation from their shares for s_i (see section 5.3 for additional details). This process requires the collaboration of at least $t+1$ correct nodes, and thus completes successfully for configurations where the adversary controls at most $n-t-1$ nodes.

7 Implementation

To highlight the feasibility and practicality of our approach, we present a prototype implementation. It consists of two parts: (i) an Ethereum smart contract serving as the communication and verification platform, and (ii) a Python client implementation, executed locally by each participant. The implementation of both parts is open source and publicly available at [link blinded for review].

In the following, we describe the steps required to apply our generalized protocol description for the concrete use case of deriving keypairs to be used with the BLS signature scheme. Thereby, we outline (i) how our communication model can be realized, (ii) which techniques are necessary to efficiently implement the required cryptographic primitives, and (iii) how the protocol execution can be verified at the smart contract level, despite the limitations of the Ethereum platform. We do not only target the BLS signature scheme because Ethereum has built-in support for a pairing friendly elliptic curve which can be used with BLS, but also due to the wide range of desirable properties this signature scheme provides for different application scenarios. These properties include short signature size, non-interactive aggregation capabilities as well as signature uniqueness.

When using our protocol for BLS signatures, a set of parties first executes our DKG protocol to compute a master BLS keypair $\langle msk, mpk \rangle$. The public key mpk is published and verified within the smart contract, whereas the (virtual) secret key msk is shared among the parties. Each party P_i is then capable of using its individual signing key gsk_i to sign messages with BLS. Any set of $t+1$ valid[6] signatures on a common message can be combined to form a threshold signature, which verifies under mpk, for that message. This aggregation process can be performed without necessitating on-chain transactions within Ethereum. Furthermore, the cost of verifying the resulting threshold signature within the smart contract does not depend on the number of participants or signers.

7.1 Realizing our Communication Model

Revisiting the assumptions from our protocol description (see section 4.1), we require a shared agreed-upon authenticated broadcast channel and adherence to certain synchrony assumptions to separate the different protocol phases. These assumptions are realized as follows:

Ethereum as a Broadcast Channel In our implementation, each participant of the DKG protocol actively monitors the Ethereum blockchain. In particular, the clients watch all transactions to the address of the pre-deployed DKG contract. A message is broadcast by issuing an Ethereum transaction, which effectively executes a function within the DKG smart contract when the transaction is mined within a block in the Ethereum network. Upon being called successfully, the contract triggers Ethereum events, which are processed by the client implementation.

Agreement After detecting the emission of a new event, the client software of each participant waits for a sufficient number[7] of confirming blocks. This ensures that all nodes agree on a common history of blocks, and consequently on the triggered events and their order w.h.p, before they react to the events. This requirement is a direct consequence of the fact that the Ethereum blockchain may fork and thus does not provide immediate agreement on newly mined blocks.

Message Authentication The requirements in regard to message authenticity are directly supported by Ethereum. In fact, Ethereum enforces that all transactions are cryptographically signed by the issuer in order to be processed.

Synchrony Assumptions Our synchrony assumptions can be realized by specifying the start and end of each protocol phase based on appropriate relative Ethereum block heights. Liveness, i.e. ensuring the protocol completes successfully even under adversarial conditions, critically depends on the ability of correct nodes to timely disseminate information. Consequently, it has to be ensured that any transaction a node issues at the beginning of a protocol phase is confirmed, and consequently received by all other correct nodes, by the beginning of the next phase. The required phase durations depend on a range of factors including: the number of participants, the state of the Ethereum network, and the amount of transaction fees the participants are willing to pay. Thus they need to be analysed on a case by case basis or selected conservatively.

7.2 Cryptographic Primitives

When leveraging a smart contract-based DKG implementation that is capable of performing the verification steps on-chain, an efficient implementation of the underlying cryptographic primitives can be crucial for a low cost protocol design. Within the Ethereum platform, only a limited range of so called pre-compiled contracts for elliptic curve cryptography are available. The supported operations target the groups $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$ of prime order q, defined on the elliptic curve BN254 [2, 3] and include point/point addition ($\mathbb{G}_1 \times \mathbb{G}_1 \rightarrow \mathbb{G}_1$), point/scalar multiplication ($\mathbb{G}_1 \times \mathbb{Z}_q \rightarrow \mathbb{G}_1$) and a verification procedure for the pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$. We rely upon these operations to efficiently implement the verification procedures for our DKG, targeting the generation of keys for the BLS signature scheme.

As BLS public keys reside in $\mathbb{G}_2$, most of the operations required for our protocol would use group $\mathbb{G}_2$, if we directly apply our general protocol description. However, as of the current Ethereum release, computations in $\mathbb{G}_2$ are not natively supported, and implementing the required operations using available Ethereum Virtual Machine (EVM) opcodes would lead to very high gas consumption and thus render the approach inefficient.[8] Fortunately, the corresponding operations in group $\mathbb{G}_1$ and a verification procedure for the pairing e

[6] The process is robust in the sense that the validity of an individual signature can also be checked using the issuer's public key.

[7] For an in depth discussion on the required number of confirmations we refer to the works of Gervais et al. [24] and Sompolinsky and Zohar [40].

[8] A Solidity implementation of a single multiplication of a group element from $\mathbb{G}_2$ with a 256 bit scalar requires approximately 2,000,000 gas [1],

exist as pre-compiled contracts in Ethereum [9,35]. This allows us to efficiently perform operations in $\mathbb{G}_1$ and verify the corresponding element in $\mathbb{G}_2$ using the pairing check within the smart contract. In the following sections 7.3, 7.4 and 7.5, we outline the details for incorporating this approach into our protocol design.

7.3 Sharing Phase

During the sharing phase, each participant $P_i \in \mathcal{P}$ proceeds as specified in our general protocol description (see section 5.1). In particular, P_i shares a secret $s_i \in_R \mathbb{Z}_q$ among all parties in $\mathcal{P}$ using Feldman's VSS protocol. The commitments $C_{i0}, C_{i1}, \ldots, C_{it}$ are group elements from $\mathbb{G}_1$: $C_{ik} = g_1^{c_{ik}} \mid 0 \leq k \leq t$, where g_1 denotes a generator of $\mathbb{G}_1$. Because there are no primitives for symmetric encryption available within Ethereum as of August 2019, we realize the encryption and decryption algorithms $\mathsf{Enc}_{k_{ij}}(\cdot)$ and $\mathsf{Dec}_{k_{ij}}(\cdot)$ using a one time pad, where we derive a unique key from k_{ij} and j by using a cryptographic hash function[9] $\mathsf{H}(\cdot)$:

$$\mathsf{Enc}_{k_{ij}}(s_{i\to j}) = s_{i\to j} \oplus \mathsf{H}(k_{ij} \mid\mid j)$$
$$\mathsf{Dec}_{k_{ij}}(\overline{s_{i\to j}}) = \overline{s_{i\to j}} \oplus \mathsf{H}(k_{ij} \mid\mid j)$$

Here, adding the the index of the receiver j ensures that the one time pad is indeed only used once, i.e. the $s_{i\to j}$ and $s_{j\to i}$ are xored with different values. To publish the required information, namely the encrypted shares $\overline{s_{i\to j}}$ for all $i \neq j$ and the commitments $C_{i0}, C_{i1}, \ldots, C_{it}$, the client software constructs and broadcasts the corresponding Ethereum transaction, invoking the pre-deployed smart contract.

The smart contract ensures that only *eligible* parties, i.e. $P_i \in \mathcal{P}$ may provide a *single*, *well-formed* message. The set of eligible parties is either specified statically at the time of creation of the smart contract, or via a dynamic registration process as described in section 7.6. A message is considered well-formed, if it contains exactly $n-1$ encrypted shares, and $t+1$ commitments to the coefficients of the secret sharing polynomial. Upon receiving a well-formed transaction from an eligible party, the smart contract notifies all other participants about the published information using an Ethereum event. The contents of the encrypted shares and the validity of the commitments are not verified at this point in time. Instead, the verification is only performed on demand, i.e. in case a dispute is submitted in the next protocol phase. In order to verify a potential dispute in the next phase, the smart contract stores a cryptographic hash of the message content. As we see in section 7.4, the hash is sufficient to fully verify a potential dispute. It would also be possible to store the entire message instead of the digest. However, storing only the hash significantly reduces the amount of on-chain storage required, and thus lowers transactions fees, in particular for large n.

7.4 Dispute Phase

In case a party P_j finds that P_i provided an invalid share for s_i, P_j follows our general protocol description to publish a dispute. For this purpose, it constructs a transaction which, in addition to k_{ij} and $\pi(k_{ij})$, includes the message content sent by P_i in the previous protocol phase. This enables the smart contract to recompute and compare the hash of P_i's message with the stored value. If the hashes do not match, the dispute is found invalid and the smart contract aborts. Otherwise the smart contract has all information required to perform a full verification. In particular, it knows that the encrypted share $\overline{s_{i\to j}}$ present in the dispute transaction is indeed the share P_i previously distributed. The verification continues as stated in section 5.1. The corresponding computations can efficiently be performed using the Ethereum pre-compiled contracts [35] for arithmetic in $\mathbb{G}_1$. If the dispute is considered valid, the share issuer is flagged as adversarial and thus excluded from the set $\mathcal{Q}$ in the key derivation phase. Additionally, the smart contract triggers a corresponding event to notify all parties about the successful dispute. Optionally the issuer may be economically punished, and a security deposit could be used to refund the disputer for its transaction fees. Similarly, an adversarial disputer could be penalized for submitting an invalid dispute. In either case, the contract may not process a dispute transaction against an already disqualified participant. In fact, in this scenario, our implementation of the smart contract aborts immediately in order to save transaction fees.

7.5 Key Derivation Phase

Again, we closely follow our protocol specification from section 5.3 to implement the key derivation phase. Similar to the definition of h, we use $h_1 \in \mathbb{G}_1$ and $h_2 \in \mathbb{G}_2$ to denote independently selected generators for the groups $\mathbb{G}_1$ and $\mathbb{G}_2$.

As a first step, each $P_i \in \mathcal{Q}$ computes the values ${h_1}^{s_i}$ and the corresponding NIZK proof $\pi({h_1}^{s_i})$ showing its correctness. The corresponding computations are performed in group $\mathbb{G}_1$. However, as the master public key $mpk = {h_2}^{msk}$ is an element of $\mathbb{G}_2$, P_i is also required to map its key share ${h_1}^{s_i}$ to $\mathbb{G}_2$, i.e. compute $h_2^{s_i}$. Then, P_i crafts and publishes a transaction, containing ${h_1}^{s_i}$, $\pi({h_1}^{s_i})$ and $h_2^{s_i}$. As described, a collaboration of $t+1$ parties recovers s_i (and thus ${h_1}^{s_i}$, $\pi({h_1}^{s_i})$ and $h_2^{s_i}$) in case P_i does no publish the required information by the end of the key derivation phase. After completing the recovery, any one of the involved parties can issue the corresponding transaction on behalf of P_i. Either way, it is ensured that ${h_1}^{s_i}$, $\pi({h_1}^{s_i})$ and $h_2^{s_i}$ become public and available for the smart contract for all $P_i \in \mathcal{Q}$. The smart contract can verify the provided information with the $\mathsf{DLEQ\text{-}verify}(\cdot)$ procedure and use the precompiled pairing contract [9] to check the validity

[9] In our implementation, the value $s_{i\to j}$ and the output of the used cryptographic hash function are 256 bits each.

of ${h_2}^{s_i}$. The value ${h_2}^{s_i}$ is considered correct if $e({h_1}^{s_i}, h_2) = e(h_1, {h_2}^{s_i})$ holds.

Finally, any party can compute and publish the master public key $mpk = \prod_{P_i \in Q} h_2^{s_i}$ and $mpk^* = \prod_{P_i \in Q} h_1^{s_i}$. The smart contract can recompute mpk^* and use the pairing $e(\cdot)$ to verify the correctness of mpk.

7.6 Dynamic Participation

The utilization of a smart contract platform such as Ethereum also enables us to readily implement dynamic participation strategies. If the choice is made to employ this protocol feature, the set of participants $\mathcal{P}$ which run the DKG protocol is not defined a priori, but rather obtained in an additional *registration phase*, executed at the beginning of the protocol. For this purpose, the creator of the corresponding smart contract specifies a set of participation rules at the time of contract creation. A participation rule specifies under which condition a particular Ethereum account is allowed to "join" the set $\mathcal{P}$. Within the limitations of the Ethereum platform, arbitrary smart contract code can be used to define participation rules. In the following, we provide basic examples for participation rules while more elaborate and robust schemes against adversarial behavior are left to future work.

1. *First come, first serve*: Only the first N parties to register are allowed to join the protocol.

2. *Security deposit*: Only parties, which provide a security deposit of at least X Ether are allowed to join the protocol.

3. *Highest bidding*: The N parties, which provided the highest amount of security deposit are allowed to join the protocol.

For conditions 1 and 2 the participation rules are checked as soon as a registration transaction is included in an Ethereum block. Only upon success is the issuer of the transaction added to the set $\mathcal{P}$, tracked within the smart contract. The implementation of condition 3 is rendered slightly more complex. In this case, the smart contract keeps track of the set $\mathcal{P}$ consisting of up to N participants and their provided security deposits. Upon registration of party P_{N+1}, the registration is accepted if the deposit provided is bigger than the smallest deposit received so far. If this is the case, the registration is accepted by adding P_{N+1} to the set $\mathcal{P}$ and removing the participant with the smallest deposit from $\mathcal{P}$. Otherwise the registration is rejected and $\mathcal{P}$ remains unchanged.

8 Evaluation

As previously outlined, the correct execution of our DKG protocol implementation can be verified with the corresponding Ethereum smart contract. To demonstrate the practicability of our solution, we evaluate the computational costs for all interactions between the parties and the smart contract platform. Figure 1, provides the measured gas consumption per executed transaction for different numbers of parties participating in the DKG protocol. We observe that (i) gas costs for contract deployment $(3,126,281)$, for registration $(107,489)$, and key share submission $(517,770)$ do not depend on the number of participants, (ii) costs for recovery linearly depend on the number of recovered parties, whereas (iii) the costs for the other operations increase linearly with increasing numbers of participants. Figure 1 reports the measured costs for (ii) and (iii) in the worst case for different numbers of participants (n). We use a setup with $n = 2t+1$ participants, where t parties are executing adversarial actions, i.e. they either provide invalid shares, handled by issuing dispute transactions, or they withhold the required values during the key generation phase, leading to a recovery of the missing information.

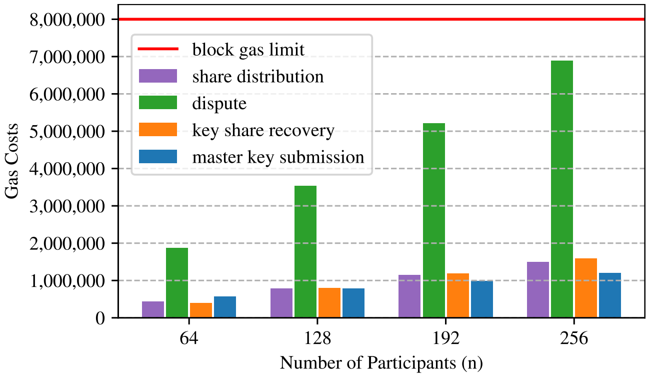

Figure 1: Computational costs, measured in gas per transaction, for all the interactions with the smart contract

The most critical operation in terms of gas consumption is the execution of a dispute transaction. In the most demanding scenario with $n = 256$ participants, a dispute consumes approximately 6,9 million gas. At a current exchange of 1 ETH $\approx$ 200 USD and a recommended[10] gas price of 1.3 Gwei[11], this amounts to approximately 1.80 USD in transaction fees. If we consider a full protocol execution with $n = 256$ parties, the transaction fees each participant has to pay sum up to approximately 0.67 USD in the optimal case (no disputes and no recovery), and 2.46 USD in the worst case (including a dispute and a recovery of missing key shares).

The costs for a dispute are largely dominated by the internal share verification procedure (see equation 4). In particular, the required elliptic curve multiplications are currently relatively expensive, at 40,000 gas each. Consequently our protocol would benefit from the EIP-1108 proposal [13], which aims to reduce gas costs for these operations. However, even with

[10] The estimates are based on the recommend gas prices provided by https://ethgasstation.info/ for average confirmation times at the time of writing.

[11] 1 Gwei $= 10^{-9}$ ETH

the current gas cost computations, and for at least 256 participants, our protocol is able to perform all required operations for handling a dispute within the current Ethereum block gas limit of approximately $8,000,000$ gas. In addition, a dispute transaction has to be executed only once per adversarial party, independent of the number of invalid shares the adversary distributed. In case all participants behave according to the protocol, no transactions for dispute, and key share recovery are executed. Additionally, the adversaries security deposit might be seized and used to cover the expenses for the transaction fees. While issuing a dispute transaction against the same party a second time is much cheaper, as the contract aborts prematurely the additional costs can be mitigated: A potential strategy is to continuously monitor for dispute transactions and only issue dispute transactions on demand at randomized points in time within the bounds of the dispute phase.

In order to keep costs for the share distribution low, we minimize the amount of data stored within the smart contract. In particular, we do not store the transaction data, i.e. $n-1$ encrypted shares and $t+1$ commitments to the secret sharing polynomial, in the smart contract. Instead, only a cryptographic hash of the above information is stored, whereas triggering a corresponding Ethereum event renders the full data easily accessible to all clients. During the verification of a dispute, this cryptographic hash is recomputed and compared to the stored value to ensure that the disputer's information is correct.

9 Discussion

Model In our DKG protocol, we follow the model described in the theoretical works of Gennaro et al. [21]. Consequently, we inherit three important characteristics for our protocol: (i) the synchronous communication model, (ii) the separation of the underlying consensus platform and the DKG protocol itself, (iii) the optimal threshold t, i.e. secrecy and liveness for all $t < n/2$. These are in contrast to the properties of the more recent works by Kate et al. [26, 27], which consider an asynchronous communication model. While these works still require a weak synchrony assumption [14] to ensure liveness, the protocol's safety guarantees do not depend on timing assumptions of the underlying message delivery network. To mitigate this risk in a synchronous protocol design, the corresponding timings, i.e. the number blocks in each protocol phase for our protocol, have to be selected appropriately.

A drawback of moving to the asynchronous model, is a reduced resilience against Byzantine adversaries. In the hybrid failure model ($n = 3t + 2f + 1$), described by Kate et al., the protocol can only tolerate less than $1/3$ Byzantine parties (t), and less than $1/2$ crashed participants (f). Here, our protocol design can prove advantageous as it ensures the desired security properties, in particular secrecy and liveness, with up to $n = 2t + 1$ participants.

Secrecy / Liveness Trade-off Our protocol design enables the use of different values for the parameter t, specifying the threshold for the underlying secret sharing protocol, depending on the specific application scenario. The choice of t directly incurs a trade-off between liveness and secrecy. If an adversary controls at most t nodes, secrecy is ensured, whereas at least $t+1$ honest nodes are required to guarantee liveness. For example, setting $t = n$, ensures that as long as there is at least one honest participant, the master secret key msk cannot be learned by the adversary. On the contrary, even a single adversarial node can prevent successful completion of the protocol. In practice the choice of t is directly related to the application scenario. If we consider, for example, the use for a synchronous BFT protocol in a setting with $n = 2f + 1$ participants, t is set to equal f, whereas a typical requirement in asynchronous or particularly synchronous BFT protocols, i.e. that more than $2/3$ of the parties have to sign a particular state or message, is supported by setting $t = \lceil 2/3n \rceil - 1$.

Uniform Key Distribution During the key derivation phase, we follow Neji et al. [30] to implement a protection mechanism, which prevents the adversary from biasing bits of the generated keypair. While the implemented countermeasure does not require a full additional secret sharing round, it requires up to two[12] additional transactions issued by all participants. To save these costs and reduce the protocols complexity, one might decide to omit the additional steps required to ensure uniform distribution of the keypair. Instead, each party P_i publishes a commitment $H(C_{i0})$ to the value C_{i0} prior to the sharing phase. The values C_{i0}, published during the sharing phase, are only accepted if they match the corresponding commitment. During the key derivation phase, the master public key mpk is directly computed as described for the Joint-Feldman protocol (see equation 5). Such a design decision may be useful e.g. in a deployment scenario, where we expect the DKG protocol to complete without any errors, i.e. in a scenario where we assume that it is very likely that all participants follow the protocol accordingly. However, as described in section 8 and shown in figure 1, the additional costs required to achieve uniformity do not add much overhead to the overall protocol execution. Consequently, we recommend to use our protocol design without this modification for most practical scenarios.

Ethereum as Communication Infrastructure As described in section 4.1, a key component necessary for the implementation of our DKG protocol is a suitable communication layer. Using an existing distributed ledger that provides Byzantine fault tolerance and agreed upon total ordering of exchanged messages. Although our approach may also be used on top of traditional BFT protocols or other avail-

[12] one transaction for publishing the key share h^{s_i} and proof $\pi(h^{s_i})$, and potentially an additional message for recovering any missing key shares

able blockchain platforms, we decided to use an existing blockchain platform, namely Ethereum, instead of deploying our own communication infrastructure. If we compare our solution to the protocol described by Kate et al. we observe a key difference in the design approach: whereas in our protocol, the core functions of the DKG protocol are separated from the the underlying consensus mechanism, Kate et al. describe their protocol in a standalone setting, intertwining a custom BFT protocol with the DKG logic. We see advantages in both approaches, depending on the application scenario. While the technique we present can benefit from an easier deployment and a simplified protocol design due to the separation of concerns, the security of Kate et al.'s approach does not depend on an external consensus mechanism and can hence operate in a stand-alone setting.

Benefits of On-Chain Verification While on-chain verification is not required for the core functionality of the protocol, it immediately provides a range of benefits: e.g. other applications on the Ethereum platform can be assured that the master public key was correctly computed, and can thus safely use this key to verify threshold signatures issued under the corresponding (shared) secret key. Furthermore, including monetary incentive mechanisms allows us to define a wide range of interesting dynamic participation models. It is no longer required to define the set of parties $\mathcal{P}$, executing the protocol, prior to the protocol start. Instead, the smart contract logic can be used to specify under which conditions a party is allowed to join the protocol. For example, participation could require a security deposit that is only returned if a party executes all steps of the protocol correctly. Otherwise this deposit can be seized to economically punish/disincentivize adversarial behavior (and unresponsiveness[13]).

Implementation and Scalability Part of the motivation for this work was the lack of public available DKG protocol implementations. In particular and to the best of our knowledge, there are no implementations of a DKG protocol following Gennaro et al.'s design, despite the extensive theoretical research in this direction. On the contrary, the protocol design by Kate and Goldberg [26] was implemented and evaluated in subsequent work [27], performing tests of their implementation with up to 70 nodes on the PlanetLab platform. Our protocol implementation was realized and evaluated using the Ethereum platform as a communication layer. Consequently, the scalability of our approach is limited by the transaction fees required to execute transactions on Ethereum. Nevertheless, our measurements (see section 8) show that even in a demanding scenario with 256 participants, all transactions can be executed well within Ethereum's current block gas limit at reasonable costs.

To support an even higher number of participants, our solution would require a cost reduction in the underlying elliptic curve primitives on the Ethereum platform, e.g. an implementation of the EIP-1108 proposal [13]. Alternatively, our protocol is able to use Ethereum (or a different platform) as communication layer only. This leads to reduced gas costs and increased scalability as the gas otherwise consumed for the on-chain verification is saved.

While the clients can still fully verify the protocol execution off-chain, the lack of on-chain verification also comes with the disadvantage, namely that seizing a security deposit becomes more difficult without placing honest clients at risk. It is no longer possible to seize the deposit automatically during the submission process of a dispute, as the smart contract does not perform the corresponding verification steps. A partial mitigation strategy is that a majority of the participants of the DKG verify a dispute off-chain and confirm its validity. However, this leads to the issue that a honest party's security deposit may be seized if the DKG protocol is run by an adversarial majority. This is is contrast to the approach with on-chain verification, which always ensures that the deposit of correct party remains safe.

10 Conclusion

We present ETHDKG, a new state of the art protocol for distributed key generation. Thereby, we demonstrate how to efficiently implement an improved variant of Gennaro et al.'s [21] theoretical work. Our enhancements include a new mechanism to resolve disputes, which arise if certain parties violate the protocol rules, as well as a range of techniques improving the performance of our implementation in practice. We outline that our protocol design is simple enough to be used with minimal costs on existing blockchain infrastructures. In particular, we show that all verification steps required during the protocol execution can be performed efficiently within the constrained environment of the Ethereum platform. By levering the Ethereum blockchain, or an alternative platform with similar guarantees, we are able to decouple the implementation of the underlying consensus protocol and the cryptographic components at the core of the DKG protocol itself. This approach simplifies the protocol design and security analysis, while at the same time enabling new novel features, such as dynamic participation and support for economic incentives, by utilizing the capabilities of the Ethereum smart contract platform. Furthermore, we show that our approach does not compromise scalability by demonstrating that our protocol can support at least $n = 256$ participants with full support for on-chain verification.

[13] The decision to actually seize a security deposit in case of unresponsiveness should be taken with great care. For example, a high temporary rate of transactions in the Ethereum network or an active denial of service attack could limit the ability of honest parties to get their transaction confirmed, or force them to pay very high transaction fees.

Availability

This paper is supported by an open source implementation of the described protocol. All artifacts in regard to the implementation are public available at [linked blinded for review]. These artifacts include:

- the source code of the DKG protocol client
- the source code of the Ethereum smart contract
- an extensive test suite for both parts of the implementation, including simulations of adversarial behavior
- evaluation scripts for measuring gas costs
- additional implementation specific documentation

References

[1] Mustafa Al-Bassam. Implementation of elliptic curve operations on G2 for alt_bn128 in Solidity. https://github.com/musalbas/solidity-BN256G2, 2019. Accessed: 2019-08-21.

[2] Diego F Aranha, Koray Karabina, Patrick Longa, Catherine H Gebotys, and Julio López. Faster explicit formulas for computing pairings over ordinary curves. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 48–68. Springer, 2011.

[3] Jean-Luc Beuchat, Jorge E González-Díaz, Shigeo Mitsunari, Eiji Okamoto, Francisco Rodríguez-Henríquez, and Tadanori Teruya. High-speed software implementation of the optimal ate pairing over barreto–naehrig curves. In *International Conference on Pairing-Based Cryptography*, pages 21–39. Springer, 2010.

[4] George Robert Blakley. Safeguarding cryptographic keys. *Proc. of the National Computer Conference*, 48:313–317, 1979.

[5] Dan Boneh, Manu Drijvers, and Gregory Neven. Compact multi-signatures for smaller blockchains. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 435–464. Springer, 2018.

[6] Dan Boneh and Matthew Franklin. Efficient generation of shared rsa keys. In *Annual International Cryptology Conference*, pages 425–439. Springer, 1997.

[7] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 416–432. Springer, 2003.

[8] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 514–532. Springer, 2001.

[9] Vitalik Buterin and Christian Reitwiessner. EIP 197: Precompiled contracts for optimal ate pairing check on the elliptic curve alt_bn128. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-197.md, 2017. Accessed: 2019-08-21.

[10] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography. In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, pages 123–132. ACM, 2000.

[11] Jan Camenisch and Markus Stadler. Proof systems for general statements about discrete logarithms. *Technical report/Dept. of Computer Science, ETH Zürich*, 260, 1997.

[12] Ran Canetti, Rosario Gennaro, Stanisław Jarecki, Hugo Krawczyk, and Tal Rabin. Adaptive security for threshold cryptosystems. In *Annual International Cryptology Conference*, pages 98–116. Springer, 1999.

[13] Antonio Salazar Cardozo and Zachary Williamson. EIP 1108: Reduce alt_bn128 precompile gas costs. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-1108.md, 2018. Accessed: 2019-08-21.

[14] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

[15] David Chaum and Torben Pryds Pedersen. Wallet databases with observers. In *Annual International Cryptology Conference*, pages 89–105. Springer, 1992.

[16] Arka Rai Choudhuri, Matthew Green, Abhishek Jain, Gabriel Kaptchuk, and Ian Miers. Fairness in an unfair world: Fair multiparty computation from public bulletin boards. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 719–728. ACM, 2017.

[17] Ronald Cramer, Rosario Gennaro, and Berry Schoenmakers. A secure and optimally efficient multi-authority election scheme. *European transactions on Telecommunications*, 8(5):481–490, 1997.

[18] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE transactions on information theory*, 31(4):469–472, 1985.

[19] Paul Feldman. A Practical Scheme for Non-interactive Verifiable Secret Sharing. In *Foundations of Computer Science, 1987., 28th Annual Symposium on*, pages 427–438. IEEE, 1987.

[20] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Advances in Cryptology-EUROCRYPT 2015*, pages 281–310. Springer, 2015.

[21] Rosario Gennaro, Stanisław Jarecki, Hugo Krawczyk, and Tal Rabin. Secure distributed key generation for discrete-log based cryptosystems. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 295–310. Springer, 1999.

[22] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Revisiting the distributed key generation for discrete-log based cryptosystems. *RSA Security'03*, pages 89–104, 2003.

[23] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Secure applications of pedersen's distributed key generation protocol. In *Cryptographers' Track at the RSA Conference*, pages 373–390. Springer, 2003.

[24] Arthur Gervais, Ghassan O Karame, Karl Wüst, Vasileios Glykantzis, Hubert Ritzdo rf, and Srdjan Capkun. On the security and performance of proof of work blockchains. In *Proceedings of the 2016 ACM SIGSAC*, pages 3–16. ACM, 2016.

[25] Time Hanke, Mahnush Movahedi, and Dominic Williams. Dfinity technology overview series consensus system, 2018. Rev. 1.

[26] Aniket Kate and Ian Goldberg. Distributed key generation for the internet. In *2009 29th IEEE International Conference on Distributed Computing Systems*, pages 119–128. IEEE, 2009.

[27] Aniket Kate, Yizhou Huang, and Ian Goldberg. Distributed key generation in the wild. *IACR Cryptology ePrint Archive*, 2012:377, 2012.

[28] Eleftherios Kokoris-Kogias, Enis Ceyhun Alp, Sandra Deepthy Siby, Nicolas Gailly, Linus Gasser, Philipp Jovanovic, Ewa Syta, and Bryan Ford. Calypso: Auditable sharing of private data over blockchains. Cryptology ePrint Archive, Report 2018/209, 2018.

[29] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 31–42. ACM, 2016.

[30] Wafa Neji, Kaouther Blibech, and Narjes Ben Rajeb. Distributed key generation protocol with a new complaint management strategy. *Security and communication networks*, 9(17):4585–4595, 2016.

[31] Orbs Network. DKG for BLS threshold signature scheme on the EVM using solidity. https://github.com/orbs-network/dkg-on-evm, 2018. Accessed: 2019-08-21.

[32] Torben Pryds Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *Annual International Cryptology Conference*, pages 129–140. Springer, 1991.

[33] Torben Pryds Pedersen. A threshold cryptosystem without a trusted party. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 522–526. Springer, 1991.

[34] Michael O Rabin. Transaction protection by beacons. *Journal of Computer and System Sciences*, 27(2):256–267, 1983.

[35] Christian Reitwiessner. EIP 196: Precompiled contracts for addition and scalar multiplication on the elliptic curve alt_bn128. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-196.md, 2017. Accessed: 2019-08-21.

[36] Eric Rescorla. Rfc 2631: Diffie-hellman key agreement method. Technical report, RFC, IETF, June, 1999.

[37] Ronald L Rivest, Adi Shamir, and David A Wagner. Time-lock puzzles and timed-release crypto. 1996.

[38] Philipp Schindler, Aljosha Judmayer, Nicholas Stifter, and Edgar Weippl. Hydrand: Practical continuous distributed randomness. In *Proceedings of IEEE Symposium on Security and Privacy (IEEE S&P)*. IEEE, 2020. to appear.

[39] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[40] Yonatan Sompolinsky and Aviv Zohar. Bitcoin's security model revisited. *arXiv preprint arXiv:1605.09193*, 2016.