A Decentralized Distributed Key Generation Primitive for Blockchain Applications

Daniel Marin

***Abstract*—We present a decentralized framework for carrying out Distributed Key Generation (DKG) protocols useful for blockchain applications like Threshold Signatures on elliptic curves. In particular, we implement Pedersen's DKG protocol using Tendermint as the underlying Byzantine-Fault-Tolerant inter-node atomic broadcast channel.**

I. Introduction

Blockchains are authenticated stateful systems that use cryptographic signatures for executing transactions. Most blockchains support some specific cryptographic signature scheme that allows end-users to sign and broadcast transactions to a peer-to-peer network. The ability of multiple parties to share a single private key for authentication in blockchain systems arises from multiple fronts: increased key-management security for end-users and institutional parties, and the existence of newer technologies like decentralized blockchain oracles.

These use cases with multiple parties sharing a single private key can be achieved via a cryptographic technique called Secret Sharing [1], in which a *dealer* distributes shares of a single private key among a set of n parties using information-theoretic properties of polynomials on finite fields and allowing any subset of $t < n$ parties to reconstruct the key. This allows the instantiation of Threshold Signature schemes in which any subset of $t < n$ parties are able to generate a signature as if it was produced by using the original master secret key. The secret-sharing approach however, provides only a centralized solution: there needs to be a trusted dealer who has full knowledge of the private key.

In order to generate a shared private key without ever having to trust a single party, a Distributed Key Generation protocol is needed. At their core, DKG protocols provide a way to generate a private-public master key pair that is shared in a threshold manner among n parties. That is, any subset of parties of size $t < n$ are able to recover n. Most DKG protocols need the primitive of a "broadcast channel", a medium of inter-node communication that allows nodes to reliably "publish" a set of values needed to identify dishonest parties. Assuming the existance of a reliable broadcast channel has made most work on DKG protocols remain in the theoretical domain.

In this paper we present an implementation of a DKG protocol that uses the Tendermint [8] consensus protocol as the underlying abstraction of a Byzantine-fault-tolerant broadcast channel, thus providing a primitive for instantiating Threshold Signature schemes suitable for blockchain applications.

II. System Model

Let $\{P_1, \ldots, P_n\}$ be a set of n computer nodes, where up to $t < n/3$ are dishonest, and which are connected by a complete network of private, authenticated, point-to-point channels.

Δ Synchrony Model. We assume a *partially synchronous* communication model. That is, the computation proceeds in synchronized rounds and messages are received by their recipients within an specified time bound Δ. In addition, we assume the parties possess synchronized clocks, and they start executing any protocol within Δ time from each other.

PKI. We assume the existence of a public key infrastructure by which every party P_i has a public key known to all other parties. This could be achieved, for instance, by having the parties register and publish their keys on a blockchain.

Setup. Let $\mathbb{G}$ be a group of prime order q with generator $g \in \mathbb{G}$ for which the discrete-log assumption [4] holds, and let $\mathbb{Z}_q$ denote its corresponding scalar field. We assume q and $\mathbb{G}$ are known to all parties.

III. Byzantine Broadcast

A broadcast channel is an abstraction of a *consensus* protocol, an instance of the well-known problem of Byzantine Agreement, which is itself an instance of the BFT-SMR consensus problem. Byzantine Agreement

protocols in the partially synchronous setting operate with at most f corrupt parties and can withstand $f < n/3$ corruptions, which is optimal. Any such protocol satisfies the following properties:

- **Agreement.** All honest parties output the same value.
- **Liveness.** All honest parties output some value.
- **Validity.** If the sender is honest, then all honest parties output the value the sender broadcasted.

A. Tendermint

We use the Tendermint [8] consensus protocol to instantiate a Byzantine Broadcast channel among nodes, providing the basis for inter-node communication based on the need of an atomic-broadcast abstraction for our DKG protocol. Tendermint is a leader-based protocol inspired by the PBFT [7] state-machine-replication (SMR) protocol, proceeding in rounds. However, unlike PBFT, Tendermint is easier to understand thanks to its single mode of execution: the protocol proceeds equally each round regardless of leader corruption.

Tendermint provides exactly the Byzantine-fault-tolerant abstraction needed for building the DKG broadcast channel, satisfying all of the agreement, liveness and validity properties under the $f < n/3$ assumption of faulty processes. The open-source Go implementation of Tendermint uses threshold signatures, which allow it to achieve a performance of $O(n)$ communication complexity both in the presence of honest and faulty leaders.

IV. Distributed Key Generation

A. Shamir's Secret Sharing Scheme

Sharmir's scheme is a 2-tuple of algorithms $(G_{\text{sh}}, C_{\text{sh}})$ (generate, combine) that form a t-out-of-n secret sharing scheme over $\mathbb{Z}_q$ defined as follows:

- $G_{\text{sh}}(n, t, \alpha)$: choose random $c_1, \ldots, c_t \xleftarrow{R} Z_q$ and let $f : \mathbb{Z}_q \to \mathbb{Z}_q$ be the **secret polynomial** defined as

$$f(x) := \alpha + c_1 x + c_2 x^2 + \cdots + c_t x^t \in \mathbb{Z}_q[x].$$

  Note that $f(0) = \alpha$ and f is of degree at most t. For $i = 1, \ldots, N$ compute the shares $\alpha_i \leftarrow f(i) \in \mathbb{Z}_q$ and output $\{\alpha_i\}_{i \in [n]}$.
- $C_{\text{sh}}(T, \{\alpha_j\}_{j \in T})$: interpolate the polynomial f from the $t+1$ points $\{(j, \alpha_j)\}_{j \in T}$ and output $\alpha := f(0)$.

B. Feldman's Verifiable Secret Sharing Scheme

Verifiable secret sharing schemes extend secret sharing schemes by allowing the recipients of shares to verify the validity of their shares and to filter out incorrect shares submitted by dishonest parties in the reconstruction phase.

Feldman's scheme [2] $(G_{\text{F}}, C_{\text{F}}, E_{\text{F}}, V_{\text{F}})$ (generate, combine, encrypt, verify) is a t-out-of-n verifiable secret sharing scheme over $\mathbb{Z}_q$ defined as follows:

- $(G_{\text{F}}, C_{\text{F}}) := (G_{\text{sh}}, C_{\text{sh}})$.
- $E_{\text{F}}(\alpha)$: For the coefficients $(\alpha, c_1, \ldots, c_t) \in \mathbb{Z}_q^{t+1}$ of the secret polynomial f from algorithm G_{F}, output the verification values $\mathbf{C} := (C_k)_{k \in \{0\} \cup [t]}$ defined as $C_k \leftarrow g^{c_k} \in \mathbb{G}$.
- $V_{\text{F}}(\mathbf{C}, i, \alpha_i)$: Let $F : \mathbb{Z}_q \to \mathbb{G}$ be the **public polynomial** defined as

$$F(x) := \prod_{k=0}^{t} C_k^{x^k}.$$

  Output 1 if $F(i) = g^{\alpha_i}$ and 0 otherwise.

C. Pedersen's DKG Protocol

Definition IV.1 (Pedersen's DKG)**.** Pedersen's DKG protocol [3] is a (n, t)-DKG protocol $G_{\text{DKG-P}}$ defined as follows:

1) **Sharing Phase.** Each party P_i acts as a dealer during one of n parallel executions of Feldman's VSS scheme. That is, each party P_i chooses a random $\alpha_{i0} \xleftarrow{R} \mathbb{Z}_q$ and computes the shares $(\alpha_{i1}, \ldots, \alpha_{in}) \leftarrow G_{\text{F}}(n, t, \alpha_{i0})$ and the public values $(C_{i0}, \ldots, C_{it}) \leftarrow E_{\text{F}}(\alpha_{i0})$, which define P_i's secret and public polynomials $f_i : \mathbb{Z}_q \to \mathbb{Z}_q$ and $F_i : \mathbb{Z}_q \to \mathbb{G}$. Then, P_i sends α_{ij} directly to each P_j and broadcasts the values $\mathbf{C}_i := (C_{ik})_{k \in \{0\} \cup [t]}$.
2) **Verification Phase.** Every P_j verifies the shares he received by computing $V_{\text{F}}(\mathbf{C}_i, j, \alpha_{ij})$ for $i = 1, \ldots, n$. For each failed verification at index i, P_j broadcasts $(\texttt{complaint}, i)$.
3) **Dispute Phase.** For each party P_j complaining against P_i, party P_i broadcasts $(\texttt{dispute}, j, \alpha_{ij})$, thus revealing share α_{ij}. Each other P_k then independently verifies the validity of this share by computing $V_{\text{F}}(\mathbf{C}_i, j, \alpha_{ij})$. If verification fails, then party P_i is marked *disqualified*.
4) **Key Derivation Phase.** Let $\texttt{QUAL}$ be defined as the set of non-disqualified parties, and let $f := \sum_{i \in \texttt{QUAL}} f_i$ and $F := \prod_{i \in \texttt{QUAL}} F_i$ define the shared private and public polynomials. Then:

The master secret key (not computed by anyone) is defined as

$$\alpha := \sum_{i \in \mathtt{QUAL}} \alpha_{i0} = \sum_{i \in \mathtt{QUAL}} f_i(0) = f(0). \quad (1)$$

The master public key $y = g^{\alpha}$ is computed by every party via

$$g^{\alpha} = \prod_{i \in \mathtt{QUAL}} g^{\alpha_{i0}} = \prod_{i \in \mathtt{QUAL}} g^{f_i(0)} = F(0). \quad (2)$$

Each party computes their share α_i of α by

$$\alpha_i := \sum_{j \in \mathtt{QUAL}} s_{ji} = \sum_{j \in \mathtt{QUAL}} f_j(i) = f(i). \quad (3)$$

Each party computes the public-key share g^{α_i} of every other party P_i via

$$g^{\alpha_i} := \prod_{j \in \mathtt{QUAL}} g^{s_{ji}} = \prod_{j \in \mathtt{QUAL}} F_j(i) = F(i). \quad (4)$$

Note that $\alpha = f(0)$ can be recovered by any $t+1$-sized subset of honest parties by interpolating the t-degree polynomial f with $t+1$ shares $\alpha_i = f(i)$.

V. Implementation

The nodes in our network are implemented as NodeJS servers written in TypeScript. We found this stack appealing because of NodeJS's event-driven approach to server-side applications. Nodes communicate via the gRPC open-source RPC framework by implementing gRPC servers and clients with a minimal interface.

Our development environment includes Tendermint Core's Go implementation, which uses the ABCI RPC-based interface to interact with the underlying consensus engine, also built using gRPC.

For the elliptic curve and cryptographic primitives we used the MCL pairing-based cryptography library and its WebAssembly interface compatible with NodeJS. This provided the basic elliptic-curve primitives for multiplication and exponentiation over the BN-254 [5] elliptic curve.

A master script instantiates all nodes, generating for each a public-private key pair $sk \xleftarrow{R} \mathbb{Z}_q$, $pk \leftarrow g^{sk}$. Then the script spawns independent child node processes that execute the protocol in isolated threads. Each node is assigned an ID and and an HTTP port on the same machine, and is given information about each other nodes' IDs and ports. Figures 1 and 2 provide example logs of simulations where all parties are honest and where one process is faulty, respectively, demonstrating that our implementation follows the expected behaviour in both cases.

```
info: DDKGNode #0: Initiating DKG, n: 3, t: 2, sessionId: 1146427605, port: 50051, inde
x: 0, server: 0.0.0.0:50051
info: DDKGNode #0: starting server
info: DDKGNode #1: Initiating DKG, n: 3, t: 2, sessionId: 1146427605, port: 50052, inde
x: 1, server: 0.0.0.0:50052
info: DDKGNode #1: starting server
info: DDKGNode #2: Initiating DKG, n: 3, t: 2, sessionId: 1146427605, port: 50053, inde
x: 2, server: 0.0.0.0:50053
info: DDKGNode #2: starting server
info: DDKGNode #0: executing sharing phase
info: DDKGNode #1: executing sharing phase
info: DDKGNode #2: executing sharing phase
info: DDKGNode #0: server stopped
info: DDKGNode #0: executing verfication phase
info: DDKGNode #1: server stopped
info: DDKGNode #1: executing verfication phase
info: DDKGNode #2: server stopped
info: DDKGNode #2: executing verfication phase
info: DDKGNode #0: executing dispute phase
info: DDKGNode #1: executing dispute phase
info: DDKGNode #2: executing dispute phase
info: DDKGNode #0: executing key derivation phase
info: DDKGNode #1: executing key derivation phase
info: DDKGNode #0: QUAL: 0,1,2, disqualified: , mpk: 19154254693645969184369268678687127
31676918201313836397618399243403829962090 73,722809380745652474229169026978684875301 1205
17705276983810185067255580178804 7,889449195493099756347772362465249639647650210081596 86
74825876402132501656789,144577503302839510451671679673599047758830953739705029458687462
85021585381900
info: DDKGNode #1: QUAL: 0,1,2, disqualified: , mpk: 19154254693645969184369268678687127
31676918201313836397618399243403829962090 73,722809380745652474229169026978684875301 1205
17705276983810185067255580178804 7,889449195493099756347772362465249639647650210081596 86
74825876402132501656789,144577503302839510451671679673599047758830953739705029458687462
85021585381900
info: DDKGNode #2: executing key derivation phase
info: DDKGNode #2: QUAL: 0,1,2, disqualified: , mpk: 19154254693645969184369268678687127
31676918201313836397618399243403829962090 73,722809380745652474229169026978684875301 1205
17705276983810185067255580178804 7,889449195493099756347772362465249639647650210081596 86
74825876402132501656789,144577503302839510451671679673599047758830953739705029458687462
85021585381900
```

Fig. 1. Example run with $n = 3$ and $t = 2$, when all nodes are honest and follow the protocol. Note that each node agrees on the set of qualified parties QUAL, as well as the value of the master public key mpk.

VI. Conclusion

Distributed key generation protocols provide a useful cryptographic primitive for building decentralized applications built on top of threshold signature schemes. We implemented a self-sufficient implementation of Pedersen's DKG protocol for elliptic curve cryptosystems with the aid of Tendermint as a Byzantine-fault-tolerant atomic broadcast channel. Future research includes testing the scalability of our implementation as a function of n and t, as well as the number of dishonest parties participating in the DKG protocol.

References

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

[2] P. Feldman, "A practical scheme for non-interactive verifiable secret sharing," in 28th Annual Symposium on Foundations of Computer Science. IEEE, 1987, pp. 427–438.

[3] T. P. Pedersen, "A threshold cryptosystem without a trusted party," in Workshop on the Theory and Application of Cryptographic Techniques. Springer, 1991, pp. 522–526.

[4] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin, "Secure distributed key generation for discrete-log based cryptosystems," in International Conference on the Theory and Applications of Cryptographic Techniques. Springer, 1999, pp. 295–310.

[5] Barreto, Paulo S. L. M., and Michael Naehrig. Pairing-Friendly Elliptic Curves of Prime Order. 133, 2005. ePrint IACR, https://eprint.iacr.org/2005/133.

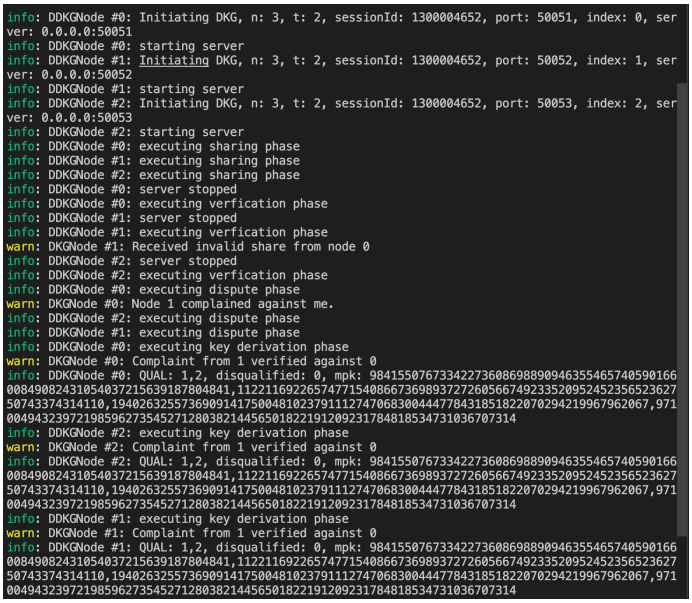

Fig. 2. Example run with $n = 3$ and $t = 2$, with node 0 being faulty. Note that each non-faulty node agrees on the set of qualified parties QUAL $= \{1, 2\}$, the set of faulty parties $\neg$QUAL $= \{0\}$, and the value of the master public key mpk.

[6] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. 1982. The Byzantine Generals Problem. ACM Trans. Program. Lang. Syst. 4, 3 (1982), 382–401. https://doi.org/10.1145/357172.357176

[7] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine Fault Tolerance. In Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA, February 22-25, 1999. 173–186. https://dl.acm.org/citation.cfm?id=296824

[8] Buchman, Ethan, Jae Kwon, and Zarko Milosevic. "The latest gossip on BFT consensus." arXiv preprint arXiv:1807.04938 (2018).