Ferveo: Threshold Decryption for Mempool Privacy in BFT networks

Joseph Bebel[1], Dev Ojha[2]

Anoma, joe@heliax.dev, Osmosis Labs, dev@osmosis.team

Abstract. A distributed network has Mempool Privacy if transactions remain encrypted until their inclusion is finalized, and inclusion guarantees decryption and execution. Mempool Privacy is highly desirable to prevent transaction censorship and a broad class of MEV attacks.

We present Ferveo, a *fast* protocol for Mempool Privacy on BFT consensus blockchains, such as those based on Tendermint. Blockchain validators use new Distributed Key Generation and Threshold Public Key Encryption schemes to decrypt transactions encrypted to a threshold public key, closely aligning security assumptions with Tendermint and providing concrete scalability up to thousands of transactions per block.

The blockchain security and efficiency models are quite different than typically studied in the academic literature, requiring several new ideas for both the abstract scheme and implementation.

Keywords: No keywords given.

1 Introduction

Decentralized finance (DeFi) is one of the most exciting applications for public blockchains. However, the public and decentralized nature of most blockchains present significant challenges such as censorship, front-running, content aware transaction re-ordering, and arbitrage stealing, collectively commonly referred to as Miner Extractable Value:

"Miner extractable value (MEV) is a measure devised to study consensus security by modeling the profit a miner (or validator, sequencer, or other privileged protocol actor) can make through their ability to arbitrarily include, exclude, or re-order transactions from the blocks they produce." [18]

The focus of this paper is on tying consensus layer security assumptions with threshold cryptography, to achieve strong mempool layer privacy guarantees. We achieve having all transaction contents being encrypted to all network participants, until the transaction is finalized within a block. Upon transaction finalization, the content is guaranteed to be decrypted in the same order by consensus. This eliminate the vast majority of MEV seen on blockchains today. MEV mitigation approaches deployed today don't solve the underlying problem: The miner or transaction orderer can see the contents of the transactions it reorders.

Ferveo is an *efficient* distributed key generation and threshold public key encryption protocol for Tendermint-based Proof of Stake blockchains with minimal assumptions over those used to achieve consensus. The assumption alignment and concrete efficiency are the key features, achieving only tens of milliseconds of amortized compute overhead per transaction as seen in Figure 1.

Assumption alignment is a critically important property. If assumptions are not aligned between the consensus and mempool privacy mechanisms, then the the whole system will

rely on the union of all assumptions. For example, using threshold decryption on a Proof of Work blockchain can be quite difficult to simultaneously guarantee mempool privacy protections and the consensus properties of the blockchain.

By encrypting all transactions contained in the mempool, arbitrary entities are far more restricted in their ability to risklessly extract MEV from the mempool. Although the validator set collectively retains decryption capability, and therefore can collectively continue to extract MEV, this requires collusion of at least 2/3 of the voting power of the network; therefore mitigating the most common forms of MEV. On top of this, generic techniques can be applied to achieve further transaction ordering properties within a block. The details of Ferveo mirror the security assumptions of the underlying blockchain very closely. Ferveo achieves this by using a single non-interactive round for each validator in both the DKG and threshold decryption schemes.

Assumption	Tendermint	Ferveo
Liveness	$\geq 2/3$ of voting power is live	$\geq 2/3 - \epsilon$ voting power is live
Security	$< 1/3$ of voting power is faulty	$< 1/3 - \epsilon$ is faulty
Synchrony	Partial	Partial
Mempool privacy	None	$< 1/3 - \epsilon$ is faulty
Censorship resistance	$< 1/3$ of voting power is faulty	$< 1/3 - \epsilon$ is faulty
Misbehavior	Penalties (Reportable)	Silent (Unreportable)

Comparison of assumptions between Tendermint and Ferveo. ϵ is as defined in the section Approximation of voting power by shares.

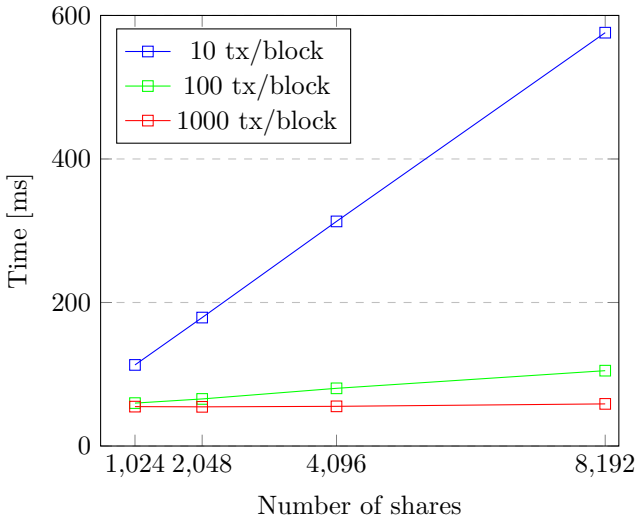

Figure 1: Amortized TPKE decryption time per transaction, single core. This task is fully parallelizable, and only needs to be executed by a block proposer, who produces a result easily verifiable by all nodes in the network

1.1 Motivation

The primary source for miner-extractable value derives from public or semi-public knowledge of the contents of transactions prior to their execution. The existence of miner-extractable

value negatively affects financial applications on blockchain through increased fees, reduction of arbitrage profits, selective transaction censorship, etc. Furthermore, it prohibits blockchains from achieving strong guarantees around one of its fundamental goals, censorship resistance. Transactions typically enter the public mempool, where they are publicly visible, prior to execution, giving an opportunity for actors scanning the mempool for useful information to obtain MEV. The amount of MEV can be reduced by introducing *mempool privacy*, where the contents of transactions within the mempool are not publicly visible and are plausibly not revealed until after an execution order for those transactions has been committed to on the blockchain.

To solve this, blockchain protocols can include distributed key generation (DKG) and threshold public key encryption (TPKE) schemes for transaction contents [20, 24]. Based on the principles of Shamir Secret Sharing, a distributed key generation protocol [19] generates a public key along with corresponding n private key shares held by a predetermined set of entities. For a fixed threshold t, a TPKE scheme allows for every subset of at least t shares to decrypt messages while every subset of at most $t-1$ shares cannot decrypt. Transaction contents can be encrypted to the public key; once a transaction is committed to the blockchain, the entities can begin the threshold decryption protocol; therefore, the contents of the transaction remain private until holders of at least t shares begin the threshold decryption protocol.

Entity selection when using threshold public key encryption is critical. The decryption protocol is particularly dependent on how block proposers are selected and how transactions are executed, and ideally the blockchain is designed with threshold decryption in mind. In this paper, we consider adding threshold encryption to fast finality BFT blockchains, such as Tendermint. In this setting, the choice of entities is natural, namely the validators already active for consensus. We amend their requirements in consensus from solely voting on blocks to also include creating and gossiping decryption shares. In the case of block proposers, their role now also includes aggregating the decryption shares into a block.

A key requirement for these networks is efficiency of the threshold decryption process during consensus. A large focus of this work is designing this, to jointly achieve low bandwidth overhead during consensus, and efficient properties for full nodes, validators and the block proposer. For instance, in a typical instantiation with 100 validators, the extra bandwidth overhead for a threshold encrypted transaction is around 5 kB (due to gossiping decryption shares), but the resultant block space overhead is under 100 bytes. As can be seen in Figure 1, the execution time overhead is quite sub-linear per transaction, still allowing for block proposers to prepare a block in under a second. The computational overhead for full nodes is negligible relative to other transaction verification logic, e.g. signatures, as it primarily consists of symmetric cryptography operations.

1.2 Alternative approaches to Mempool Privacy

There are at least two other approaches to mempool privacy being explored within the blockchain space, Trusted Execution Environment Encryption (TEE Encryption), and Timelock Encryption. We describe these below.

Solution method	Security cost	Delay	Additional Assumptions
TEE Encryption	Cost of breaking TEE	Block Finality	TEE manufacturer honesty
Timelock Encryption	Timelock hardware	Many blocks	All txs delayed
Threshold Encryption	2/3rds of validators	Block Finality	BFT Proof of Stake

- **Timelock Encryption**: Forms of time-lock encryption, typically based on Verifiable Delay Functions (VDFs) [4, 27, 23] can be used to encrypt transactions such that

they require T sequential timesteps to decrypt. Benefits of a time-lock encryption based approach include potential alignment of assumptions in proof of work networks, more decentralization of the decrypters, and lower risk of MEV obtained by secret collusion of miners or validators. The primary downsides of timelock encryption are the hardware assumptions for this specialized problem, and the latencies this forces for all transaction executions.

The timelock period must account for the length a transaction typically spends in the mempool, as an adversary can start decrypting as soon as they see the transaction. It also must include a multiplicative factor for better timelock hardware risk, and account for the expected time until there is "sufficient" probabilistic finality of the chain to avoid re-org concerns. This ends up requiring a significant delay before transactions can be decrypted and executed on the chain, prohibiting many latency-sensitive applications. This timelock period cannot be opt-in, every transaction in the system must be forced to go through it, otherwise non-encrypted transaction could front-run encrypted ones.

- **TEE Encryption**: The idea of TEE (Trusted Execution Environment) encryption for mempool privacy is to have every node in the network have a TEE (i.e. Intel SGX), and for them to all share an encryption key. Then all transactions are encrypted to the enclaves. They get decrypted and executed once they are in a finalized block on-chain.

  The security assumption this depends on the cost to attack a given Trusted Execution Environment, whereas the worth of exploitable MEV is upwards of $750 million USD [8], which is conjectured to be far greater than the reasonable cost of breaking any existing Trusted Execution Environment.

- **Witness Encryption**: Witness encryption [9] essentially enables the same functionality as TEE Encryption, but solely using cryptographic assumptions. Namely, creating an encrypted tx that can only be decrypted upon presenting inclusion of its hash into a finalized block. However, the known constructions are not practically efficient, often relating to Indistinguishability Obfuscation, and to our knowledge have not been implemented in practice.

2 Techniques and Design Goals

Encrypting the contents of the mempool requires some desirable properties in order to be useful: The desired properties are mempool encryption are the following.

1. Information safety: It is infeasible to decrypt the contents of a transaction prior to finalization of the block that contains it
2. Execution Guarantee: Once a valid transaction is committed to a block, it must be executed
3. Efficiency: the scheme must be computationally and communication efficient enough to not substantially limit the throughput and latency of the network

In the specific case of BFT consensus-based blockchains, such as those based on the Tendermint protocol, there is a natural correspondence between the network validators and owners of private key shares. In typical Tendermint instantiations, holders of a staking token delegate their tokens to one or more of a fixed number of network validators. Validators propose and vote on potential blocks in proportion to the amount of stake delegated to each one; once at least 2/3 of delegated stake has voted for a proposed block, the block is finalized and executed.

Since such a BFT consensus-based blockchain already depends on an assumption that at least 2/3 of validators (by delegated stake, or *voting power*) are following the protocol, ideally we want transactions to be encrypted such that they can be decrypted only by at least 2/3 of validators (by voting power). The blockchain protocol can add threshold decryption as a required step in the block voting process, aligning the voting and decryption steps precisely: blocks are finalized if and only if a sufficient amount of decryption shares for all contained transactions become public.

2.1 Design Goals

2.1.1 Weighting

Validators' delegated stake can be highly nonuniform. In existing proof of stake networks support delegation, we see a clear power law distribution of stake amongst the validators. Therefore 1 private key share per validator does not align the decrypter set with the validator set. While it is theoretically possible to issue 1 private key share for each minimal unit of the staking token, as a practical matter this would create a completely impractical number of private key shares. DKG computational complexity scales significantly in the number of shares. This creates a tension between making Information Safety align as close to possible with consensus safety, and making efficiency properties achievable. A compromise approach is to give validators private key shares according to their approximate relative staking weight – allowing a tradeoff between efficiency and precise correlation to the consensus assumption. Concretely, 100 or 200 validators may collectively hold 4096 or 8192 private key shares.

2.1.2 Approximation of voting power by shares

Since a validator's voting power must be approximated by an allocation of a small number of private key shares, we use a partitioning algorithm PartitionDomain that attempts to minimize the error in this approximation. By multiplying each validators voting power by the total number of private key shares, and dividing by the total voting power of the network, we get an optimal fractional assignment of key shares. Each validator receives at least the integer floor number of key shares.

This ensures that with n validators and m private key shares, the approximation error is at most n private key shares. In the case of 100 validators holding 8192 private key shares, then a decryption threshold $t = 2 \cdot 8192/3 - 100 = 5362$ implies at least 65.5% of voting power is needed to decrypt.

2.1.3 Public fees

Transaction fees and gas limits must remain public, both for spam prevention (ensuring that transactions pay for their overhead) and to ensure blocks don't exceed their overall gas limits. In Ethereum, a block producer can execute txs and know exactly how much gas they consumed. Therefore, a user only needs to be charged fees for the gas they used. However in this setting, the fees must be collected / block space reserved *prior* to a tx being executed. Thus every tx must be charged for the gas limit it allocates. While this does leak a small amount of information about each transaction, this is far less than the information contained in the actual contents of the transaction, and can in large part be mitigated by padding or lower-granularity gas limits. (At the expense of higher fees)

2.1.4 Consensus Efficiency

The threshold encryption scheme presented in [2] assumes one private key share per party and has excellent theoretical and concrete efficiency in this model. Since there is one

private key share per validator, and therefore one decryption share per validator per transaction, this scheme is close to optimal.

However, our weighted share distribution model introduces some efficiency issues with using [2] for Ferveo. Multiple private key shares per validator mean multiple decryption shares are needed per validator per transaction.

Therefore, a significant problem with using a large number of private key shares is potential overhead due to a large number of decryption shares for each transaction. For example, in a hypothetical worst case scenario, if a single decryption share is 48 bytes and 5398 (2/3 of 8192) decryption shares are needed, then over 259 kB of decryption shares could be needed for each transaction, a significant overhead. However, this can be mitigated in two ways:

1. In case of successful decryption, old decryption shares can be discarded or pruned. Only the valid 32-byte symmetric key needs to be kept.

2. By using a re-keying trick, our TPKE scheme only needs one $\mathbb{G}_1$ decryption share per validator and not one decryption share per private key share, a substantial savings. Hypothetically, if a hundred validators issue a single 48 byte decryption share per transaction, then 4.8 kB of decryption shares is much more reasonable.

While overhead of the DKG and TPKE schemes is somewhat unavoidable, much of the overhead can be restricted to validator nodes only, and not full nodes in the network. Full nodes do not need to participate at all in the threshold decryption protocol, or verify any steps, other than verifying decryption shares in case of invalid transactions.

Since validator nodes are rewarded through fees for participating in the network, they can invest in the CPU and bandwidth resources necessary for executing the DKG and TPKE protocols; whereas full nodes only have modestly increased overhead.

2.1.5 Handling maliciously crafted transactions

A more significant issue for censorship resistance is that maliciously crafted transactions must be detectable. The protocol requires that every transaction committed to a finalized block must be executed; however, if a purposefully undecryptable “garbage“ transaction is submitted, then we do not want the block to be discarded, the blockchain halted, or any validators to be penalized for failure to decrypt. This means that encrypted transaction validity must be publicly verifiable, such that either transactions are always successfully decrypted and executed, or everyone can agree that the transaction is invalid.

This can be achieved with our TPKE scheme combined with a key-committing AEAD scheme; validators publicly reveal decryption shares for a transaction during the voting phase, which can be combined to obtain the shared symmetric key. In case the symmetric key does not match the committed key, the validity of each individual decryption share can be verified. If all decryption shares are valid, then the obtained symmetric key is valid, and any mismatch with the committed key (or any other problems with the transaction contents) is the fault of the transaction creator, and the transaction can be safely ignored without execution without compromising censorship resistance.

2.2 Network model

2.2.1 Synchronous vs asynchronous

Much of the research effort on practical distributed key generation, including in the blockchain domain, has focused on the asynchronous network model [15]. The asynchronous model is more realistic for protocols running on the public internet, but it is much more difficult to achieve resilience when adversaries and communication failures can create

Byzantine faults. It is nontrivial to obtain DKGs with good concrete and asymptotic complexity, in particular obtaining $O(n \log n)$ communication complexity for n entities holding n private key shares.

However, in the case where the DKG protocol passes messages on an already synchronized blockchain, the DKG protocol can be substantially conceptually simpler as the protocol inherits properties, such as censorship resistance, from the underlying blockchain. There is also substantial benefit from reusing already existing gossip protocols. However, the relatively large amount of data that a DKG protocol exchanges is a complicating issue when the underlying blockchain cannot support enough bandwidth. This does put significant load on the blockchain's gossip protocol. Ideally, the blockchain supports pruning the DKG data once the generated public key becomes obsolete, which reduces the long term storage costs.

While our $O(nm)$ communication synchronous DKG protocol with n validators and m private key shares is worse than the best achievable $O(m \log n)$ communication complexity, the concrete performance remains good for reasonably sized (100-200) validator sets. Future work can substantially improve performance in this area.

2.2.2 Epoched staking

Our DKG will assume that the validator set and each validator's voting power is fixed during each epoch, consisting of some number of blocks.

The validator set and voting powers might differ dramatically between epochs, which requires running the DKG again for each epoch to generate a new public key (at a cost of safety, the same public key may be reused for more than 1 epoch)

The "handover" between epochs is a subtle situation which requires close analysis. Ideally, the next epoch's validator set engages in the DKG protocol during the current epoch, so that the next epoch's public key is ready for transactions beginning with the first block of the next epoch. This presents a theoretical risk that the current epoch's validator set adversarially censors all DKG messages, leading to the network halting (unable to execute new transactions) at the new epoch until the DKG protocol can run. However, since the threshold (at least 1/3 of voting power) to censor DKG messages is equal to the power to halt the network anyway, this risk does not substantially add to the power of the previous epoch's validator set.

A more significant issue is how to handle transactions submitted near the epoch boundary. The transaction submitter must choose which public key to encrypt to, which may be difficult to choose at this time.

3 Cryptography overview

Publicly verifiable distributed key generation (DKGs) and efficient threshold public key encryption ("Threshold Encryption") are desirable as primitives for mitigating censorship, and preventing front-running on blockchains. Using threshold encryption in a blockchain moves censorship and front-running from being risklessly executable by any node in the network, to requiring breakage of security assumptions close to the blockchain's underlying security assumptions.

Ideally, both the distributed key generation protocol and the threshold decryption protocol should as closely mirror the security properties of the underlying blockchain as closely as possible, to avoid having to weaken the underlying assumptions of the entire system.

In particular, it is desirable to have a publicly verifiable DKG, so that a party's lack of liveness during a DKG complaint round does not affect their risk of getting invalid key shares.

The primary problem with existing publicly verifiable distributed key generation protocols [14] is their generation of private keys (and private key shares) that are group elements (elliptic curve points) instead of traditional finite field elements.

Although publicly verifiable distributed key generation protocols exist that generate finite field private keys [13] their overall complexity makes them less desirable at the current time. In addition, our threshold encryption scheme issues only one decryption share per validator, a significant advantage that actually appears to be possible because the private key shares are group elements.

Instead, we introduce a new threshold public key encryption scheme, based on standard assumptions, that is compatible with the private key shares generated by our publicly verifiable distributed key generator.

3.1 Our Results

Ferveo consists of:

- A publicly verifiable DKG scheme with acceptable performance
- A threshold public key encryption scheme with excellent performance that can scale to thousands of transactions: Figure 1
- A pure-Rust implementation of both the DKG and TPKE schemes, built on arkworks [1]
- Naturally integrate-able into Tendermint based blockchains

3.2 Related Work

The primary basis for our threshold public key encryption scheme is [2], a scheme that can be adapted to use modern pairing-friendly elliptic curves. However, this scheme uses finite field private keys.

Another existing scheme, [3] is designed to avoid the use of random oracles and is also a useful scheme because it uses group element private keys (via a construction based on threshold Identity Based Encryption).

The scheme presented below relies on random oracles, uses group element private keys, and is slightly conceptually and computationally simpler than using a generic threshold IBE based construction.

4 Cryptographic Schemes

4.1 Publicly Verifiable Distributed Key Generation

Most research into distributed key generation has focused on Verifiable Secret Sharing (VSS) where secret shares are field elements and Pedersen or KZG polynomial commitments [16, 26] are used. The recent "Aggregatable DKG" [14] demonstrated the usefulness of Publicly Verifiable Secret Sharing (PVSS) to achieve Publicly Verifiable Distributed Key Generation (PVDKG). Similar to [14], we use the "SCRAPE" PVSS scheme [7] to share a secret group element in a publicly verifiable way. A distributed key can be generated by the standard method: each party acts as a trusted PVSS dealer, and all validly dealt PVSS instances are summed elementwise to obtain the final distributed public key and key shares.

Unlike the "Aggregatable DKG", for conceptual simplicity we avoid the aggregation step and all parties verify each PVSS instance. This adds extra computation load to validators and full nodes, but does not add much latency since a single validator would

have to perform the aggregation step anyway. While it would be desirable to optimize this step further, it is not necessary to achieve reasonable concrete performance levels.

For reference, the details of the SCRAPE PVSS scheme used are described below:

4.2 SCRAPE Publicly Verifiable Secret Sharing

4.2.1 Initialization

All parties have encryption keys $ek_i \in \mathbb{G}_2$ for each party P_i. Party P_i knows the corresponding private dk_i such that $ek_i = [dk_i]H$.

4.2.2 Dealing

$Scrape.Deal(bp, ek_1, \ldots, ek_n) \to pvss$

To deal a PVSS instance, the dealer samples a uniformly random polynomial $f(x) = a_0 + a_1 x + \ldots + a_t x^t$ of degree t, computes a Pedersen commitment $F_0, \ldots, F_t$ to f

1. $(a_i, \ldots, a_t) \leftarrow \mathbb{F}^t$
2. $F_0, \ldots, F_t \leftarrow [a_0]G, \ldots, [a_t]G$
3. $A_1, \ldots, A_n \leftarrow [f(\omega_1)]G, \ldots, [f(\omega_n)]G$
4. $Y_1, \ldots, Y_n \leftarrow [f(\omega_1)]ek_1, \ldots, [f(\omega_n)]ek_n$
5. $pvss \leftarrow (F_0, \ldots, F_t, A_1, \ldots, A_n, Y_1, \ldots, Y_n)$

$pvss$ is called a PVSS transcript of this instance.

4.2.3 Verifying

A PVSS transcript $pvss$ may be verified by anyone using this verification procedure:

$Scrape.Verify(bp, ek_1, \ldots, ek_n, pvss) \to \{0, 1\}$

1. $(F_0, \ldots, F_t, A_1, \ldots, A_n, Y_1, \ldots, Y_n) \leftarrow parse(pvss)$
2. choose uniformly random $\alpha \leftarrow \mathbb{F}$
3. check $\prod_{j=1}^{n} A_j^{\ell_j(\alpha)} = \prod_{j=0}^{t} F_j^{\alpha^j}$
4. check $e(G, Y_j) = e(A_j, ek_j)$ for all $1 \le j \le n$

where ω_i is an nth root of unity.

F_0 is the public key of a PVSS instance associated with the private key $[a_0]H$.

4.2.4 Homomorphic addition

Two PVSS transcripts $pvss_1$ and $pvss_2$ may be added to get $pvss$ by elementwise addition.

4.3 Final DKG protocol

The Ferveo DKG requires no trusted party. We achieve this in the standard way, by having each validator act as a trusted PVSS dealer and then adding all distributed shares together.

It is well known that this kind of DKG may output a biased key [12, 11]; in particular the final PVSS dealer can bias the final key. Instead of attempting to unbias the final key, which is itself nontrivial, we instead show that our threshold decryption scheme is re-keyable as defined in [14] and therefore remains secure with a biased key.

for each validator V_i **do**
  V_i generates a uniformly random session key dk_i.
  V_i announces its public session key $ek_i = [dk_i]H$ on the blockchain
end for

Each validator runs PartitionDomain
for each validator V_i **do**
  V_i computes $pvss_i \leftarrow$ Scrape.Deal
  V_i posts $pvss_i$ on the blockchain
end for
for each validator V_i **do**
  for e **do**ach validator V_j, $i \neq j$
    V_i runs Scrape.verify($pvss_j$)
    V_i computes $pvss \leftarrow pvss + pvss_j$
  end for
  When 2/3 by voting power of validators have posted valid $pvss_j$, exit **for**
end for
All validators agree on $pvss$

Note that this DKG protocol will always succeed as long as at least 2/3 of voting power in the network follows the DKG protocol. In case that at least 1/3 does not follow the protocol, this DKG protocol will never finish.

The final public key generated by the DKG is F_0 of $pvss$.

4.4 Threshold Decryption Scheme

The primary contribution of this section is an efficient and simple threshold decryption scheme that is compatible with keys generated by SCRAPE PVSS-based DKG protocols.

4.4.1 Encryption

$TPKE.Encrypt(Y, aad)$ creates a new, random ciphertext (U, W, aad) encrypted to the public key Y, and a corresponding ephemeral shared secret S such that the private key associated with Y can efficiently compute S from the ciphertext (U, W, aad). Additional authenticated data 'aad' may be attached to the ciphertext, for example key commitment data.

1. Let r be a uniformly random scalar.
2. Let $S = e([r]Y, H)$
3. Let $U = [r]G$
4. Let $W = [r]H_{\mathbb{G}_2}(U, aad)$

The ephemeral shared secret S can be used to derive a shared symmetric encryption key.

4.4.2 Ciphertext Validity Checking

In order to have Chosen Ciphertext security (IND-CCA) it must not be possible to request decryption of invalid ciphertexts. The $TPKE.CheckCiphertextValidity(U, W, aad)$ operation tests if (U, W) is a valid ciphertext.

The ciphertext (U, W, aad) is valid if and only if:

$$e(U, H_{\mathbb{G}_2}(U, aad)) = e(G, W)$$

4.4.3 Creating Decryption Shares

$TPKE.CreateDecryptionShare(dk_i, U, W, aad) -> D_i$

Prior to creating a decryption share, the validity of the ciphertext (U, W, aad) should be checked using $TPKE.CheckCiphertextValidity(U, W, aad)$. The decryption share D_i of ciphertext (U, W, aad) and party P_i is defined as:

$$D_i = [dk_i^{-1}]U$$

4.4.4 Verifying Decryption Shares

Prior to running $TPKE.CombineDecryptionShares(U, D_i)$, the validity of a decryption share can also be checked using $TPKE.VerifyDecryptionShare(ek_i, U, D_i) \rightarrow \{0, 1\}$, which outputs 1 if and only if:

$$e(D_i, ek_i) = e(U, H)$$

If all D_i pass this check, then $TPKE.CombineDecryptionShares(U, \{D_i\})$ is guaranteed to succeed.

4.4.5 Combining Decryption Shares

Once $t + 1$ valid decryption shares are available, they can be combined to obtain the same shared secret S as the encrypter.

$$TPKE.CombineDecryptionShares(U, \{D_i\}) \rightarrow S = \prod_i e(D_i, [\lambda_{\omega_i}(0)]Y_i)$$

where $\lambda_{\omega_i}(0)$ is the Lagrange coefficient necessary for interpolation and $Y_i = [f(\omega_i)]ek_i$ is the private key share from the PVSS instance.

4.4.6 Weighted shares optimization

In the case where the parties P_i each own several shares (because of a relative weighting among the parties) a single decryption share is sufficient for each ciphertext. If party P_i owns both shares i and j, then by setting $dk_i = dk_j$ (therefore $ek_i = ek_j$) the value:

$$e(D_i, [\lambda_{\omega_i}(0)]ek_i)e(D_j, [\lambda_{\omega_i}(0)]ek_i) = e(D_i, [\lambda_{\omega_i}(0)]Y_i + \lambda_{\omega_j}(0)]Y_j)$$

which reuses the same decryption share, lowering the communication complexity, and also simplifying the pairing. This leverages the fact that the "private" key share Y_i are actually public (although blinded by a dk_i factor), and multiplying U by dk_i^{-1} effectively "re-keys" the ciphertext to the publicly known key share Y_i, allowing anyone to complete the share combine process with the *pvss* transcript produced by the DKG.

4.4.7 Symmetric cryptography

The shared secret S can be used to derive a symmetric key used to encrypt the transaction payload. Use of a key-committing AEAD scheme ensures that once the symmetric key is derived, verified, and posted on the blockchain, none of the decryption shares need to be preserved.

4.4.8 Transaction decryption protocol

Validator V_p is selected as block proposer
V_p fills block B with encrypted txs from mempool
for each validator V_j **do**
  for each ciphertext (U_i, W_i) in B **do**
    V_j checks the validity of (U, W)
    V_j computes its decryption share $D_{i,j} = [dk_j^{-1}]U_i$
  end for

V_j includes all $D_{i,j}$ in its vote for block B
end for
for each validator V_j **do**
for each ciphertext (U_i, W_i) in B **do**
for each decryption share $D_{i,k}$ from validator V_k **do**
V_j checks the validity of decryption share $D_{i,k}$
end for
end for
If all V_k share's are valid, then V_k's vote for B is counted
end for
Once B has valid votes totaling at least 2/3 of voting power:
for each validator V_j **do**
for each ciphertext (U_i, W_i) in B **do**
V_j combines all decryption shares $D_{i,k}$ to get S_i
V_j derives the symmetric key k_i from S_i
end for
end for
B is finalized including k_i for every valid tx and $D_{i,k}$ for every invalid tx

Full nodes can verify a block by checking that at least 2/3 of voting power has signed the block (without symmetric keys) and that all symmetric keys are included (except for invalid transactions, where all decryption shares are included)

4.5 Optimizations

4.5.1 Deserialization

The DKG protocol requires validators to deserialize a large number of $\mathbb{G}_2$ points, and so using arkworks' fast subgroup check for BLS curve points [6] [22] substantially speeds up verifying each PVSS transcript.

4.5.2 Combining decryption shares

Combining decryption shares is the most computationally expensive part of the threshold decryption process, but this cost can be optimized in a few ways. In particular, Lagrange coefficients must be computed for each block, as the coefficients depend on which subset of validators submitted decryption shares. Therefore, in the worst case these Lagrange coefficients must be recomputed for each block.

1. The interpolation domain, while always a subset of roots of unity, will not in general be a closed subgroup, and so an FFT is not useful in computing the Lagrange coefficients. However, it is still possible to compute all the necessary Lagrange coefficients in $O(n \log n)$ time using a Subproduct Tree algorithm, achieving the required asymptotic performance [10, 26].

2. Since decryption shares for different transactions within a block share the same validator subset, the Lagrange coefficients only need to be computed once per block, not once per transaction

3. Once $\lambda_{\omega_i}(0)$ is computed, then $[\lambda_{\omega_i}(0)]Y_i$ can also be computed (per block) and reused for all transactions within that block

4. All line functions of $[\lambda_{\omega_i}(0)]Y_i$ can also be computed once per block and reused for each product of pairings for each transaction. This computation is done by converting $[\lambda_{\omega_i}(0)]Y_i$ to 'G2Prepared' type in arkworks.

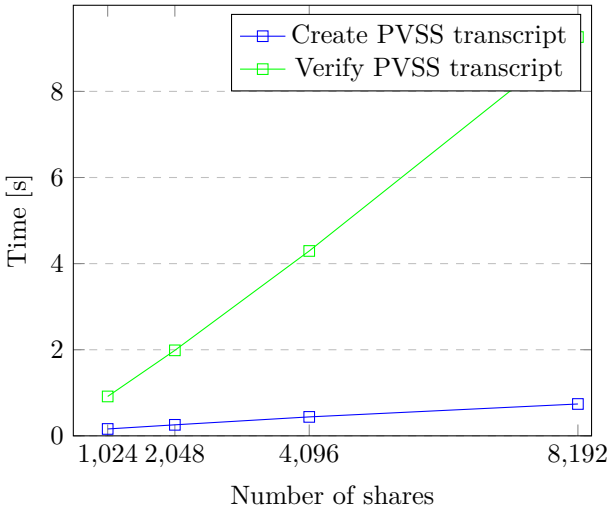

Figure 2: PVSS benchmarks

5. Although in the worst-case the subset of validators used to reach the 2/3 threshold may be different for every block, in practice liveness in Tendermint based protocols is excellent and often >99% of validators vote on each block. Therefore, the 'G2Prepared' value may be cached and reused between blocks as long as the subset of validators used remain online.

These optimizations are critical for scalability. As shown in Figure 1, because many of the per-block compute costs can be amortized across many transactions, the incremental compute time needed per additional transaction is minimal or even insignificant compared to the fixed costs.

As shown in Figure 4, the cost to combine decryption shares is dependent on the number of validators submitting decryption shares rather than the number of private key shares.

The total compute load per-block is therefore $O(m + t)$ where m is the number of private key shares and t is the number of transactions, providing excellent scalability when both m and t are increased to useful levels.

5 Benchmarks

All benchmarks were performed on a 2021 MacBook Pro 16" with M1 Max CPU and 64 GB RAM. Note that arkworks supports inline assembly on the x86-64 architecture for faster finite field and elliptic curve performance.

6 Future Work

Threshold decryption bandwidth: The threshold decryption approach used now requires high amounts of bandwidth overhead. Namely, every validator must gossip a unique piece of data for every encrypted transaction. There are two methods for mitigating this:

- Making transactions encrypted to a particular block height. This weakens several guarantees around not being able to discover the tx content though. We detail this more in the appendix

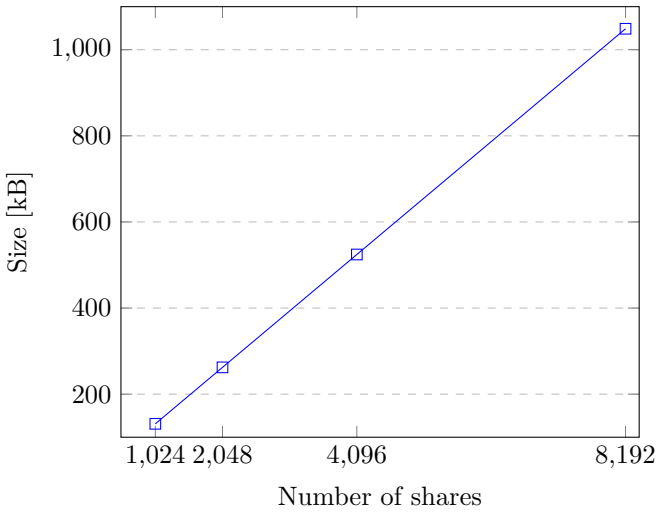

Figure 3: Estimated PVSS transcript size

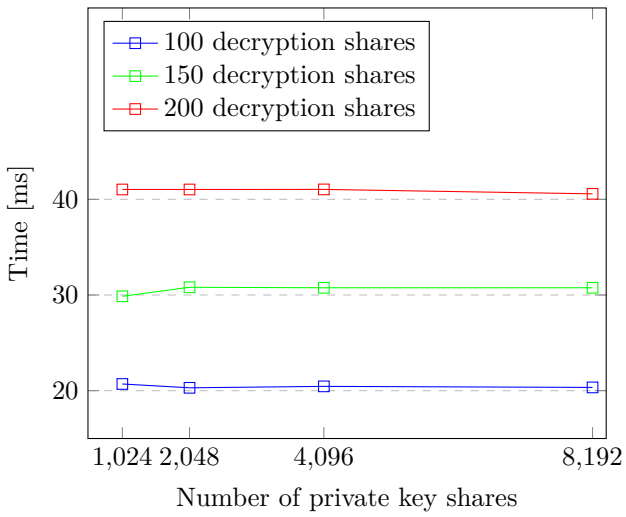

Figure 4: Decryption share combine time per tx

- Allowing aggregation of validator-specific decryption shares, within the p2p layer. This only matters if validator to validator communication is done over a flood gossip p2p topology.

DKG bandwidth: The most significant deficiency in our protocol is the high bandwidth requirement for the DKG due to the $O(nm)$ communication complexity of the DKG. This should be improvable to $O(m \log n)$ using an approach similar to [14] where multiple gossiped PVSS transcripts are aggregated prior to commitment in a block; the main open question is how to best achieve this while retaining most of the benefits of using a synchronous DKG protocol. In particular, reuse of the existing gossip protocol for blocks and transactions

Our protocol also lacks reportability or penalties for validators failing to follow the protocol. There is nothing preventing validators with at least 2/3 voting power from silently engaging in an alternative out-of-band protocol to decrypt transactions early and extract MEV. While early release of decryption shares might potentially be reportable in some way, it is difficult to protect innocent validators if the PVSS-dealing validator set chooses to fabricate early decryption shares.

Additional measures such as SGX or other compute enclaves could be used to further enforce protocol fidelity.

7 Acknowledgements

Thanks to Christopher Goes for guidance on the goals of this project, Christian Cachin for DKG discussions, Simon Masson for discussions about pairing-friendly elliptic curves and his arkworks contribution of fast subgroup checking for BLS curves, Alin Tomescu for advice on scaling DKGs, and Ash Manning, Georgios Gkitsas, and Jacob Turner for contributions to the Rust implementation of the DKG and TPKE crates. Special thanks to Kobi Gurkan for feedback on the TPKE scheme and his suggestion for how to create one decryption share per validator.

References

[1] *Arkworks*. URL: https://github.com/arkworks-rs.

[2] Joonsang Baek and Yuliang Zheng. "Simple and efficient threshold cryptosystem from the Gap Diffie-Hellman group". In: *GLOBECOM '03. IEEE Global Telecommunications Conference (IEEE Cat. No.03CH37489)*. Vol. 3. 2003, 1491–1495 vol.3. DOI: 10.1109/GLOCOM.2003.1258486.

[3] Dan Boneh, Xavier Boyen, and Shai Halevi. "Chosen Ciphertext Secure Public Key Threshold Encryption without Random Oracles". In: *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology*. CT-RSA'06. San Jose, CA: Springer-Verlag, 2006, pp. 226–243. ISBN: 3540310339. DOI: 10.1007/11605805_15. URL: https://doi.org/10.1007/11605805_15.

[4] Dan Boneh et al. *Verifiable Delay Functions*. Cryptology ePrint Archive, Report 2018/601. https://ia.cr/2018/601. 2018.

[5] Sean Bowe. *BLS12-381: New ZK-snark elliptic curve construction*. Mar. 2017. URL: https://electriccoin.co/blog/new-snark-curve/.

[6] Sean Bowe. *Faster Subgroup Checks for BLS12-381*. Cryptology ePrint Archive, Report 2019/814. https://ia.cr/2019/814. 2019.

[7] Ignacio Cascudo and Bernardo David. *SCRAPE: Scalable Randomness Attested by Public Entities*. Cryptology ePrint Archive, Report 2017/216. https://ia.cr/2017/216. 2017.

[8] Flashbots core developers. *MEV-explore*. URL: https://explore.flashbots.net/.

[9] Sanjam Garg et al. *Witness Encryption and its Applications*. Cryptology ePrint Archive, Report 2013/258. https://ia.cr/2013/258. 2013.

[10] Joachim von zur Gathen and Jürgen Gerhard. *Modern computer algebra (2. ed.)*. Cambridge University Press, 2003, pp. I–XIII, 1–785. ISBN: 978-0-521-82646-4.

[11] Rosario Gennaro, Stanislaw Jarecki, and Hugo Krawczyk. "Revisiting the Distributed Key Generation for Discrete-Log Based Cryptosystems". In: 2007.

[12] Rosario Gennaro et al. "Secure Distributed Key Generation for Discrete-Log Based Cryptosystems". In: *Proceedings of the 17th International Conference on Theory and Application of Cryptographic Techniques*. EUROCRYPT'99. Prague, Czech Republic: Springer-Verlag, 1999, pp. 295–310. ISBN: 3540658890.

[13] Jens Groth. *Non-interactive distributed key generation and key resharing*. Cryptology ePrint Archive, Report 2021/339. https://ia.cr/2021/339. 2021.

[14] Kobi Gurkan et al. *Aggregatable Distributed Key Generation*. Cryptology ePrint Archive, Report 2021/005. https://ia.cr/2021/005. 2021.

[15] Aniket Kate, Yizhou Huang, and Ian Goldberg. *Distributed Key Generation in the Wild*. Cryptology ePrint Archive, Report 2012/377. https://ia.cr/2012/377. 2012.

[16] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. "Constant-Size Commitments to Polynomials and Their Applications". In: *Advances in Cryptology - ASIACRYPT 2010*. Ed. by Masayuki Abe. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 177–194. ISBN: 978-3-642-17373-8.

[17] Chelsea Komlo and Ian Goldberg. *FROST: Flexible Round-Optimized Schnorr Threshold Signatures*. Cryptology ePrint Archive, Report 2020/852. https://ia.cr/2020/852. 2020.

[18] Alex Obadia. *Flashbots: Frontrunning the MeV crisis*. Nov. 2020. URL: https://medium.com/flashbots/frontrunning-the-mev-crisis-40629a613752.

[19] Torben Pryds Pedersen. "Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing". In: *Advances in Cryptology — CRYPTO '91*. Ed. by Joan Feigenbaum. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 129–140. ISBN: 978-3-540-46766-3.

[20] Philipp Schindler et al. *ETHDKG: Distributed Key Generation with Ethereum Smart Contracts*. Cryptology ePrint Archive, Report 2019/985. https://ia.cr/2019/985. 2019.

[21] C. P. Schnorr. "Efficient Identification and Signatures for Smart Cards". In: *Advances in Cryptology — CRYPTO' 89 Proceedings*. Ed. by Gilles Brassard. New York, NY: Springer New York, 1990, pp. 239–252. ISBN: 978-0-387-34805-6.

[22] Michael Scott. *A note on group membership tests for $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$ on BLS pairing-friendly curves*. Cryptology ePrint Archive, Report 2021/1130. https://ia.cr/2021/1130. 2021.

[23] StarkWare. *Presenting: Veedo*. June 2020. URL: https://medium.com/starkware/presenting-veedo-e4bbff77c7ae.

[24] Oliver Stengele et al. *ETHTID: Deployable Threshold Information Disclosure on Ethereum*. 2021. arXiv: 2107.01600 [cs.CR].

[25] Douglas R. Stinson and Reto Strobl. "Provably Secure Distributed Schnorr Signatures and a (t, n) Threshold Scheme for Implicit Certificates". In: *Information Security and Privacy*. Ed. by Vijay Varadharajan and Yi Mu. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 417–434. ISBN: 978-3-540-47719-8.

[26] A. Tomescu et al. "Towards Scalable Threshold Cryptosystems". In: *2020 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2020, pp. 877–893. DOI: 10.1109/SP40000.2020.00059. URL: https://doi.ieeecomputersociety.org/10.1109/SP40000.2020.00059.

[27] Benjamin Wesolowski. *Efficient verifiable delay functions*. Cryptology ePrint Archive, Report 2018/623. https://ia.cr/2018/623. 2018.

A Technical Preliminaries

A.1 Notation

Ferveo cryptography uses the BLS12-381 pairing-friendly elliptic curve [5], which consists of groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ with order divisible by prime p, and $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is a nondegenerate bilinear map. Let G be a generator for $\mathbb{G}_1$ and H be a generator for $\mathbb{G}_2$. This curve provides an ideal security and performance level, although the abstract scheme can use other pairing-friendly elliptic curves.

A.2 Threshold Public Key Encryption Scheme

The starting point for our TPKE scheme begins with [2] and (TPKE without random oracles). The following amendments are desirable for our applications:

- Private keys are $\mathbb{G}_2$ elements instead of field elements
- Create a single decryption share for each validator, instead of each private key share
- The threshold decryption result should be publicly verifiable
- Show the scheme is re-keyable (sound with a biased key)

A.3 System Model

We assume that there are n parties who will share a "t-of-n" distributed key, such that $t+1$ parties are necessary and able to decrypt. The parties communicate over a synchronous protocol where every online party agrees on the sequence of messages exchanged.

B Proof sketches

B.1 Notation

Let $GroupGen(1^\lambda)$ be an efficient algorithm that outputs a tuple $bp = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, G, H)$ where $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ are groups with order divisible by p, G generates $\mathbb{G}_1$, H generates $\mathbb{G}_2$, and $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is a nondegenerate bilinear map.

Let $\mathrm{H}_{\mathbb{G}} : \{0,1\}^* \to \mathbb{G}$ be the hash to curve function into the group $\mathbb{G}$ as specified in RFC https://datatracker.ietf.org/doc/draft-irtf-cfrg-hash-to-curve/

B.2 Semantic security

The TPKE scheme is intended to be secure against adaptive chosen ciphertexts in the random oracle model (IND-CCA) when decrypted directly with uniformly random private keys and with threshold decryption (THD-IND-CCA [2]) when decrypted using private key shares.

Definition 1. For a probabilistic polynomial time (PPT) adversary A, their advantage is:

$$Adv(\lambda) = |Pr[gg \leftarrow GroupGen(1^\lambda); \alpha, \beta \leftarrow \mathbb{F} : A(gg, [\alpha]G, [\beta]G, [\alpha\beta]G) = 1] \\ - Pr[gg \leftarrow GroupGen(1^\lambda); \alpha, \beta, \gamma \leftarrow \mathbb{F} : A(gg, [\alpha]G, [\beta]G, [\gamma]G) = 1]|$$

Definition 2. For an adversary A,

$$Pr[gg \leftarrow GroupGen(1^\lambda); \alpha, \beta, \gamma \leftarrow \mathbb{F} : A(gg, [\alpha]G, [\beta]G, [\gamma]H, [\alpha\gamma]H) = e(G, H)^{\alpha\beta}] < negl(\lambda)$$

Theorem 1. *The TPKE is adaptive chosen ciphertext secure in the single decrypter model. Suppose for some $\epsilon > 0$ an adversary A has a non negligible advantage $Adv(\lambda) > \lambda^{-\epsilon}$.*

Then the Computational Bilinear Diffie-Hellman Assumption is false.

Proof Sketch. We will simulate the adversary that has advantage against the TPKE scheme to construct an adversary against CBDH.

On an instance of CBDH $(gg, [\alpha]G, [\beta]G, [\gamma]H, [\alpha\gamma]H)$, construct a ciphertext (U, W) where $U = [\alpha]G$ and $W = [\rho\alpha\gamma]H$ and public key $Y = [\beta]G$, where the simulation fixes $H_{\mathbb{G}_2}(U, aad) = [\rho\gamma]H$. (Whenever the simulation queries $H_{\mathbb{G}_2}$ on other values, the random oracle should return $[\rho]H$, for uniformly random ρ) As long as $[\rho\gamma]H \neq [\alpha\beta]H$ the simulation is indistinguishable to the adversary.

This ciphertext and public key are passed to the simulated adversary; since $e(U, [\rho\gamma]H) = e(G, W)$ the ciphertext validity check passes. Then by assumption, the adversary has advantage computing $e(G, H)^{\alpha\beta}$ from the ciphertext, which is now an advantage against the instance of CBDH.

In the unlikely case where $H_{\mathbb{G}_2}(U, aad) = [\alpha\beta]H$, then $e(G, H)^{\alpha\beta} = e(G, H_{\mathbb{G}_2}(U, aad))$. □

B.3 Rekeyability

The concept of rekeyability from [14] is necessary to prove that our DKG can be safely used with our TPKE scheme.

Theorem 2. *The threshold public key encryption scheme is re-keyable under the definition from [14]*

Proof Sketch. Suppose the public key Y_1 has associated private key Z_1. The shared secret S can be rekeyed with respect to the private key Z_1 to a new private key $\hat{Z} = [\alpha]Z_1 + Z_2$, as the new shared secret $\hat{S} = S^\alpha e(U, Z_2) = e(U, [\alpha]Z_2)e(U, Z_2) = e(U, [\alpha]Z_1 + Z_2)$.

The shared secret S can be rekeyed with respect to the public key Y_1 to a new public key $\hat{Y} = [\alpha]Y_1 + Y_2$ as the new shared secret $\hat{S} = S^\alpha e([r]Y_2, H) = e([r\alpha]Y_1, H)e([r]Y_2, H) = e([r]([\alpha]Y_1 + Y_2), H)$.

□

C Threshold Signature

Although Ferveo does not require or implement threshold signatures, we briefly note that the distributed key produced by our DKG can be used in a Schnorr-like [21] threshold signature scheme based on similar assumptions as our TPKE scheme. The details are omitted in the interest of time.

C.1 To sign

Let m be the message to sign.

1. Sample a random nonce k.
2. Let $R = e([k]G, H)$
3. Let $c = H_{\mathbb{F}}(R||Y||m)$
4. Let $z = [k]H + [c]Z$
5. Let the signature be $\sigma = (R, z)$

Note that $z = [k + xc]H$, but is computable without knowledge of x

As with other similar signature schemes, k may be derived in a deterministic way to avoid known issues with random nonces.

C.2 To verify

1. Let $\sigma = (R, z)$ be the signature of m
2. Let $c = H_{\mathbb{F}}(R||Y||m)$
3. Let $R' = e(G, z) * e([c]Y, H)^{-1}$
4. If $R = R'$ the signature is valid.

C.3 Threshold Signature

Threshold signature schemes have traditionally been based on Schnorr signatures and their techniques may be adapted [25], since although this scheme does not describe a standard Schnorr signature, it resembles it closely.

The technique of FROST [17] can be applied to this new scheme; the individual commitments R_i are computed similar to FROST as $R_i = e(D + [\rho_i]E_i, H)$, $R = \prod R_i$, and the value $z = [d_i + e_i\rho_i]H + [\lambda_i s_i c]Z$.

Although this signature scheme is not suitable as a random beacon, the key produced by our DKG can be used in the Verifiably Unpredictable Function defined in [14].