Fast batched asynchronous distributed key generation

Jens Groth and Victor Shoup

DFINITY

August 2, 2023

Abstract. We present new protocols for threshold Schnorr signatures that work in an *asynchronous* communication setting, providing *robustness* and *optimal resilience*. These protocols provide unprecedented performance in terms of communication and computational complexity. In terms of communication complexity, for each signature, a single party must transmit a few dozen group elements and scalars across the network (independent of the size of the signing committee). In terms of computational complexity, the amortized cost for one party to generate a signature is actually less than that of just running the standard Schnorr signing or verification algorithm (at least for moderately sized signing committees, say, up to 100).

For example, we estimate that with a signing committee of 49 parties, at most 16 of which are corrupt, we can generate *50,000 Schnorr signatures per second* (assuming each party can dedicate one standard CPU core and 500Mbs of network bandwidth to signing). Importantly, this estimate includes both the cost of an offline precomputation phase (which just churns out message independent "presignatures") and an online signature generation phase. Also, the online signing phase can generate a signature with very little network latency (just one to three rounds, depending on how throughput and latency are balanced).

To achieve this result, we provide two new innovations. One is a new secret sharing protocol (again, asynchronous, robust, optimally resilient) that allows the dealer to securely distribute shares of a large batch of ephemeral secret keys, and to publish the corresponding ephemeral public keys. To achieve better performance, our protocol minimizes public-key operations, and in particular, is based on a novel technique that does *not* use the traditional technique based on "polynomial commitments". The second innovation is a new algorithm to efficiently combine ephemeral public keys contributed by different parties (some possibly corrupt) into a smaller number of secure ephemeral public keys. This new algorithm is based on a novel construction of a so-called "super-invertible matrix" along with a corresponding highly-efficient algorithm for multiplying this matrix by a vector of group elements.

As protocols for verifiably sharing a secret key with an associated public key and the technology of super-invertible matrices both play a major role in threshold cryptography and multi-party computation, our two new innovations should have applicability well beyond that of threshold Schnorr signatures.

1 Introduction

The main motivation for our work is to design efficient protocols for threshold Schnorr signatures that work in an *asynchronous* communication setting, providing *robustness* and *optimal resilience*. As will be explained in detail below, our results and techniques result in threshold Schnorr protocols with extremely high throughput and low latency. These protocols follow the usual offline/online paradigm, where higher-latency, message-independent precomputations are performed in the offline phase, and lower-latency, message-dependent computations are performed in the online phase. The resulting protocol has linear communication complexity per signature in both phases — each party in the signing committee essentially transmits and receives a total number of 18 scalars and 9 group elements per signature in the offline phase, and 6 scalars per signature in the online phase.[1]

[1] We stress that we are not assuming any type of "broadcast channel". When we say P transmits a certain amount of data, we are counting the sum over all parties Q of the amount of data that P sends to Q over a point-to-point channel.

Moreover, for moderately sized signing committees (in the range 10–100), they enjoy extremely good computational complexity. In particular, over a group of order $q < 2^\lambda$, and with a signing committee of size n, the running time per signature of each party in the signing committee in the offline phase is dominated by the cost of performing $O(n + \lambda/n)$ group additions (we use additive notation and terminology throughout), and in the online phase is dominated by the cost of $O(n)$ arithmetic operations mod q. Note that these estimates assume some batching is done in the offline phase and a small amount of batching is done in the online phase. Somewhat surprisingly, this result says that for such moderately sized n, the running time per party per signature is less than the time used to just compute a Schnorr signature. Note that our results and techniques have much broader applicability. For example, they can also be applied to threshold ECDSA signatures and certainly other problems in threshold cryptography.

The big-O constants here are quite small. For example, with $n = 49$ and $\lambda = 256$, the number of group additions per party per signature in the offline phase is just *46*. Note that this group addition count does not presume an exorbitant amount of batching in the offline phase, nor does it assume any type of sophisticated multi-scalar/group multiplication algorithms. If the group is an elliptic curve such as `secp2565k1`, a single group addition can be performed by a reasonably good library in well under $0.3\mu s$.[2] This translates to a total of $13.8\mu s$ per signature, and we will conservatively round this up to $20\mu s$ to account for other overheads. This translates into to a throughput of 50,000 signatures per second. Now, as mentioned above each party transmits a total of 24 scalars and 9 group elements per signature, so roughly 10Kb (10,000 bits). So in order to sustain a throughput of 50,000 signatures per second, a network bandwidth of 500Mbs suffices, which is not unreasonable.

Let us recall the Schnorr signature scheme. Let E be a group of prime order q generated by $\mathcal{G} \in E$. As mentioned already, we use additive notation for the group operation of E, and denote the identity element of E by $\mathcal{O}$. The secret key is a random $x \in \mathbb{Z}_q$ an the public key is $\mathcal{X} \leftarrow x\mathcal{G} \in E$. To sign a message m, the signer chooses $r \in \mathbb{Z}_q$ at random, computes

$$\mathcal{R} \leftarrow r\mathcal{R} \in E, \quad h \leftarrow \mathsf{Hash}(\mathcal{X}, \mathcal{R}, m) \in \mathbb{Z}_q, \quad s \leftarrow r + xh \in E,$$

and outputs the signature $(\mathcal{R}, s) \in E \times \mathbb{Z}_q$.

1.1 An MPC engine geared towards Schnorr

The approach we take to designing a threshold Schnorr protocol in the asynchronous communication setting is to build it on top of a highly optimized **MPC (multi-party computation) engine**. That is, rather than designing and analyzing a monolithic protocol, we design an MPC engine that supports operations well suited to threshold Schnorr signatures (and other threshold cryptography tasks as well), and that can be efficiently implemented while providing robustness and optimal resilience in an asynchronous communication setting.

At a high level, we need an MPC engine that, as an ideal functionality $\mathfrak{F}_{\text{MPC}}$, supports the following operations:

[2] We timed this using `https://github.com/bitcoin-core/secp256k1` on a Macbook Pro with a 2.6 GHz 6-Core Intel Core i7 processor, and got results between $0.1\mu s$ and $0.3\mu s$, depending on the type of operation.

- $([r], \mathcal{R}) \leftarrow \mathsf{RandomKeyGen}()$:
  $\mathfrak{F}_{\text{MPC}}$ chooses $r \in \mathbb{Z}_q$ at random, computes $\mathcal{R} \leftarrow r\mathcal{G} \in E$, gives $\mathcal{R}$ immediately to the adversary, and gives $\mathcal{R}$ to each party as a delayed output (i.e., the adversary indicates when the output is given to any given party). The ideal functionality $\mathfrak{F}_{\text{MPC}}$ also stores r for future use.
- $([z] \leftarrow \mathsf{LinearOp}(a, [x], b, [y])$:
  For public inputs $a, b \in \mathbb{Z}_q$, and previously stored values $x, y \in \mathbb{Z}_q$, $\mathfrak{F}_{\text{MPC}}$ computes $z \leftarrow ax + by \in \mathbb{Z}_q$ and stores z for future use.
  This is typically implemented as a local computation.
- $z \leftarrow \mathsf{Open}([z])$:
  For a previously stored value $z \in \mathbb{Z}_q$, $\mathfrak{F}_{\text{MPC}}$ gives z to each party as a delayed output.

We assume we have n parties $P_1, \ldots, P_n$, at most $t < n/3$ of which may be corrupt. We also assume static corruptions — that is at the beginning of the attack and before the start of the protocol the adversary corrupts some subset of $t^* \leq t$ parties. We envision a signing protocol that is driven by a blockchain or any other BFT protocol that orders the signing requests and the execution of the $\mathfrak{F}_{\text{MPC}}$-operations, so that each party receives the same signing requests and initiates the same $\mathfrak{F}_{\text{MPC}}$-operations in the same order. It is not important here which method is used for the parties to agree on request and activation ordering.

Using the above MPC engine, we can easily implement threshold Schnorr signatures as follows. The protocol to generate the key is:

$([x], \mathcal{X}) \leftarrow \mathsf{RandomKeyGen}()$

The protocol to sign a message m is:

$([r], \mathcal{R}) \leftarrow \mathsf{RandomKeyGen}()$
$h \leftarrow \mathsf{Hash}(\mathcal{X}, \mathcal{R}, m) \in \mathbb{Z}_q$ *//local computation*
$[s] \leftarrow \mathsf{LinearOp}(1, [r], h, [x])$ *//local computation*
$s \leftarrow \mathsf{Open}([s])$
output $(\mathcal{R}, s)$

While the above is very simple, it is typically not very efficient. The problem is that in a typical implementation of $\mathfrak{F}_{\text{MPC}}$, the $\mathsf{RandomKeyGen}$ operation is fairly expensive. One way of improving this situation is to observe that the value r is independent of m, and so we might move the computation of $([r], \mathcal{R}) \leftarrow \mathsf{RandomKeyGen}()$ to an "offline" precomputation phase. In this case, we refer to the pair $([r], \mathcal{R})$ as a "presignature". Moreover, we might be able to exploit "batching" techniques to more efficiently produce such presignatures in batches, and then consume them in an "online" phase as signing requests are made.

The problem with this approach is that by computing these presignatures $([r], \mathcal{R})$ in advance and revealing the group element $\mathcal{R}$ to the adversary before the corresponding signing request is made, the signature scheme becomes insecure. This is a well-known problem and a number of good mitigation strategies have been devised. See [Sho23] for details on how this can be done efficiently and securely (see also Section 7.2).

So we can safely ignore these issues for now and focus on the remaining challenge: how to generate large batches of presignatures efficiently. In computing batches of presignatures, we are not so much concerned about the latency of producing a batch, as this occurs in the "offline"

phase. We are, however, concerned about the throughput, that is, the rate at which we can produce presignatures, as this will be the limiting factor on the signing throughput, that is, the rate at which we can process signing requests.

So the problem we focus on is this: *high-throughput generation of presignatures*. This problem may be simplified by the observation that for a presignature $([r], \mathcal{R})$, we do not require that the value r is perfectly random. In fact, as was first observed in a specific context in [GJKR07], and then again in an another specific context in [BHK+23], and then more recently in a much more general context in [Sho23], the threshold signing protocol will still be secure even if the adversary is allowed to *bias* the presignatures in a particular *benign* way, and moreover, it is much easier to generate such benignly biased presignatures than it is to generate unbiased presignatures.

One form of biased presignature that is relevant can be captured by following operation, which we can add to our MPC engine:

- $([r'], \mathcal{R}') \leftarrow \mathsf{BiasedKeyGen}()$:
  $\mathfrak{F}_{\mathrm{MPC}}$ chooses $r \in \mathbb{Z}_q$ at random, computes $\mathcal{R} \leftarrow r\mathcal{G} \in E$, gives $\mathcal{R}$ immediately to the adversary.
  The adversary later responds with a "bias" $(a, b) \in \mathbb{Z}_q^* \times \mathbb{Z}_q$.
  $\mathfrak{F}_{\mathrm{MPC}}$ then computes $r' \leftarrow ar + b \in \mathbb{Z}_q$ and $\mathcal{R}' \leftarrow r'\mathcal{G} \in E$, and gives $\mathcal{R}'$ to each party as a delayed output. $\mathfrak{F}_{\mathrm{MPC}}$ also stores r' for future use.

One can securely realize this functionality based on a simpler operation and a consensus protocol. The simpler operation is this:

- $([r_i], \mathcal{R}_i) \leftarrow \mathsf{InputKey}_i(r_i)$:
  Party P_i inputs $r_i \in \mathbb{Z}_q$ to $\mathfrak{F}_{\mathrm{MPC}}$, who computes $\mathcal{R}_i \leftarrow r_i\mathcal{G} \in E$, gives $\mathcal{R}_i$ immediately to the adversary, and gives $\mathcal{R}_i$ to each party as a delayed output. $\mathfrak{F}_{\mathrm{MPC}}$ also stores r_i for future use.

To implement $\mathsf{BiasedKeyGen}$ with this operation, each party P_i inputs a random secret key r_i to the ideal functionality via $\mathsf{InputKey}_i$ so that every party, including the adversary, learns the public key $\mathcal{R}_i$. Note that while honest parties input random secret keys, the corrupt parties may choose arbitrary secret keys in a way that depends on the public keys of the honest parties. The parties then use a consensus protocol to agree on a set $\mathcal{I}$ of $t+1$ indices, where for each $i \in \mathcal{I}$, the operation $\mathsf{InputKey}_i$ has successfully completed. The resulting biased key-pair is $([r'], \mathcal{R}')$, where $r' = \sum_{i \in \mathcal{I}} r_i$ and $\mathcal{R}' = \sum_{i \in \mathcal{I}} \mathcal{R}_i$. This is computed locally, using $\mathsf{LinearOp}$ to compute $[r']$ from $\{[r_i]\}_{i \in \mathcal{I}}$, and computing $\mathcal{R}'$ directly using the known values $\{\mathcal{R}_i\}_{i \in \mathcal{I}}$ as output by $\{\mathsf{InputKey}\}_{i \in \mathcal{I}}$. That this securely realizes $\mathsf{BiasedKeyGen}$ is fairly straightforward to see (this was observed implicitly in [Gro21] and explicitly in Section A.3.6 of [GS22]).

Yet better performance can be obtained by utilizing the well-known "batch randomness extraction" technique — an idea that goes back at least to [HN06], but first applied to Schnorr signatures in [BHK+23], and then analyzed more fully in the context of Schnorr signatures in [Sho23]. Here, we choose a certain $M \times N$ matrix W over $\mathbb{Z}_q$ (whose entries are public constants — see below), where $M = n - 2t$ and $N = n - t$. As above, each party P_i inputs a random secret key r_i to the ideal functionality via $\mathsf{InputKey}_i$, so that every party, including the adversary, learns the public key $\mathcal{R}_i$. As above, while honest parties input random secret keys, the corrupt parties may choose arbitrary secret keys in a way that depends on the public keys of the honest parties. The parties then use a

consensus protocol to agree on a set $\mathcal{I}$ of N indices, where for each $i \in \mathcal{I}$, the operation $\mathsf{InputKey}_i$ has successfully completed. Let us write

$$\mathcal{I} = \{i_1, \ldots, i_N\}.$$

The parties then locally compute M biased key-pairs

$$([r'_1], \mathcal{R}'_1), \ldots, ([r'_M], \mathcal{R}'_M)$$

where

$$\begin{pmatrix} r'_1 \\ \vdots \\ r'_M \end{pmatrix} = W \cdot \begin{pmatrix} r_{i_1} \\ \vdots \\ r_{i_N} \end{pmatrix} \qquad \text{and} \qquad \begin{pmatrix} \mathcal{R}'_1 \\ \vdots \\ \mathcal{R}'_M \end{pmatrix} = W \cdot \begin{pmatrix} \mathcal{R}_{i_1} \\ \vdots \\ \mathcal{R}_{i_N} \end{pmatrix}.$$

This is computed locally, using $\mathsf{LinearOp}$ to compute $[r'_1], \ldots, [r'_M]$ from $[r_1], \ldots, [r_N]$, and computing $\mathcal{R}'_1, \ldots, \mathcal{R}'_M$ directly using the known values $\{\mathcal{R}_i\}_{i \in \mathcal{I}}$.

The property that the matrix W must satisfy is called **super-invertibility**, which simply means that every subset of M columns of A is linearly independent. The definition of super-invertible matrices and their application to multi-party computation comes from [HN06].

The security property that the above protocol satisfies can be elegantly captured by adding the following operation to our MPC engine, where we define $M := n - 2t$ and $N^* := n - t^*$, where $t^* \leq t$ is the number of actual corrupted parties.

- $(([r'_1], \mathcal{R}'_1), \ldots, ([r'_M], \mathcal{R}'_M)) \leftarrow \mathsf{BatchedBiasedKeyGen}()$:
  $\mathfrak{F}_{\text{MPC}}$ chooses $r_1, \ldots, r_{N^*} \in \mathbb{Z}_q$ at random, computes

  $$\mathcal{R}_1 \leftarrow r_1\mathcal{G}, \ldots, \mathcal{R}_{N^*} \leftarrow r_{N^*}\mathcal{G}$$

  and gives these group elements immediately to the adversary.
  The adversary later responds with a "bias" $(A, \mathbf{b})$, where $A \in \mathbb{Z}_q^{M \times N^*}$ is a full rank matrix and $\mathbf{b} \in \mathbb{Z}_q^{M \times 1}$ is an arbitrary vector.
  $\mathfrak{F}_{\text{MPC}}$ then computes

  $$\begin{pmatrix} r'_1 \\ \vdots \\ r'_M \end{pmatrix} = A \cdot \begin{pmatrix} r_1 \\ \vdots \\ r_{N^*} \end{pmatrix} + \mathbf{b}$$

  and

  $$\mathcal{R}'_1 \leftarrow r'_1\mathcal{G}, \ldots, \mathcal{R}'_M \leftarrow r'_M\mathcal{G}$$

  and gives $(\mathcal{R}'_1, \ldots, \mathcal{R}'_M)$ to each party as a delayed output. $\mathfrak{F}_{\text{MPC}}$ also stores the values $r'_1, \ldots, r'_M$ for future use.

1.2 Two problems

So we now have reduced the problem of building threshold Schnorr to the following two problems:

Problem 1: Securely and efficiently implementing the $\mathfrak{F}_{\text{MPC}}$-operations $\mathsf{InputKey}$, $\mathsf{LinearOp}$, and Open.

Problem 2: Designing a super-invertible matrix equipped with an efficient algorithm for multiplication on the right by vector of group elements.

Indeed, with solutions to both of these problems, we can directly implement BiasedKeyGen and BatchedBiasKeyGen, as outlined above. Now, the papers [Sho23] and [BHK+23] do not analyze the usage of a biased keys as signing keys, but only as presignatures. However, the analysis in [Sho23] can be extended to allow for biased signing keys. That said, it is arguably not that important, since we presumably generate signing keys very occasionally, and so we could afford to use a more expensive implementation of an unbiased key. Moreover, for other applications, such as ECDSA, the use of an unbiased signing key is essential — see Section 3.6 of [GS22]. Note that to support ECDSA, we would also have to extend our MPC engine to include a multiplication operations. That is a issue we do not consider in the paper. Nevertheless, our techniques here can be extended to cover this as well, although with certain limitations. In any case, in this paper we do not consider any implementation of RandomKeyGen, as it is not needed to implement threshold Schnorr, but in any case, it is easy enough to implement in a way that is (a) not too horribly inefficient, and (b) is compatible with the other operations in our MPC engine.

2 Our contributions

We present new solutions to Problems 1 and 2 above.

2.1 Solution to Problem 1

Our new solution to Problem 1 is a new type of protocol, which we call a **GoAVSS protocol**. Here, GoAVSS stands for **group-oriented asynchronous verifiable secret sharing**. We assume parties $P_1, \ldots, P_n$, one of which is designated as the dealer, and at most $t < n/3$ of which may be (statically) corrupted. We also assume we have fixed evaluation points $e_1, \ldots, e_n$, which are distinct, nonzero elements of $\mathbb{Z}_q$. A GoAVSS protocol should securely realize the following ideal functionality.

- The dealer inputs polynomials $f_1, \ldots, f_L \in \mathbb{Z}_q[x]$ to the GoAVSS functionality, each of which must have degree at most t (this condition is enforced by the functionality).
- For each $\ell \in [L]$, the polynomial f_ℓ defines a secret key $s_\ell := f_\ell(0) \in \mathbb{Z}_q$ and a public key $\mathcal{S}_\ell := s_\ell \mathcal{G} \in E$, and the GoAVSS functionality immediately reveals the public keys $\mathcal{S}_1, \ldots, \mathcal{S}_L$ to the adversary.
- The GoAVSS functionality gives to each party P_j as a delayed output the public keys $\mathcal{S}_1, \ldots, \mathcal{S}_L$, as well as its shares of the secret keys $s_1, \ldots, s_L$, that is, the values $f_1(e_j), \ldots, f_L(e_j)$.

In addition, a GoAVSS protocol should satisfy a **completeness property**, which means that if the dealer is honest or any honest party outputs a value, then all honest parties eventually output a value. Here, "eventually" means if and when all honest parties initiate the protocol and all messages sent between honest parties are delivered.

Any GoAVSS protocol gives a solution to Problem 1. Indeed, such a protocol can be used directly to implement the InputKey operation of our MPC engine. In fact, one run of the GoAVSS protocol actually yields L instances of InputKey. For our application to threshold signatures, such batching is perfectly fine, as parties just input random secret keys. Moreover, such batching can be

used to get much more efficient protocols. Since a GoAVSS protocol distributes traditional Shamir shares of the secrets, the LinearOp is also easily and efficiently implemented.

As for the Open operation, the reader may notice that our GoAVSS functionality does not make any explicit mention of a "polynomial commitment" of any form that can be used to verify the secret shares revealed during the Open operation. This is intentional: our new GoAVSS protocol is very efficient precisely because it does not use a polynomial commitment at all. This is also very different from just about every other DKG protocol in the literature (going back to Feldman [Fel87] and Pedersen [Ped91]). Nevertheless, since we are assuming $n > 3t$ (which is necessary in the asynchronous setting), we do not need to use polynomial commitments to verify the secret shares revealed during the Open operation — we can instead just use error correcting codes. This is a standard technique in the field of information-theoretic asynchronous MPC. Below in Section 7.1, we review this technique, and discuss how this impacts the practical performance of the online phase of a threshold signing protocol. As we will see, to get the best throughput in the online phase, we have to increase the latency of the online phase just a bit (but this is not because we use error correcting codes). Note that the completeness property of our GoAVSS property is essential to make it possible to forgo polynomial commitments and rely on error correction.

As already mentioned, our GoAVSS protocol achieves very good performance by avoiding polynomial commitments. Instead, at a very high level, it works as follows:

- First, the dealer runs a "plain" AVSS protocol, which just works with scalars, rather than group elements, to distribute shares of the polynomials $f_1, \ldots, f_\ell \in \mathbb{Z}_q[x]$. The dealer then simply broadcasts the group elements $\mathcal{S}_1, \ldots, \mathcal{S}_L$ to the receivers.
  To get very high performance for this step, we may use the recently developed AVSS protocol from [SS23], which uses very lightweight cryptography (i.e., hash functions) and generally has very good communication and computational complexity.
- At this point, we may assume that the "plain" AVSS protocol ensures that receivers are holding shares of polynomials $f_1, \ldots, f_\ell$ of the right degree. However, there is no guarantee that $f_\ell(0)\mathcal{G} = \mathcal{S}_\ell$ for all $\ell \in [L]$. So we run another subprotocol that performs a simple statistical test. Moreover, this test is designed so that we can distribute the work among the receivers, so that each party performs a *very* small number of group additions — just $O(\lambda/n)$ additions per individual sharing, if $q < 2^\lambda$. The work actually decreases as n increases! This is, of course, too good to be true. Indeed, there is a trade-off, in that each party must perform $O(n)$ *very* simple scalar operations per individual sharing (each such operation is not much more expensive than the addition of two λ-bit numbers, and we estimate that such a scalar operation takes just a few percentage points of the time to perform a single group addition). For moderately sized n, this turns out to be an excellent trade-off.

We analyze in detail both the computational and communication complexity of our GoAVSS protocol, with certain subprotocols implemented as in [SS23]. These subprotocols have an "optimistic path", where no party misbehaves in a publicly provable way, and we only consider the cost on this optimistic path. Arguably, over a long run of the system where parties that provably misbehave are effectively removed from the system, this is the only cost that matters. Now, each run of our GoAVSS protocol produces L "raw sharings" created by a single dealer. In the intended usage as a subprotocol in our implementation of the BatchedBiasedKeyGen, every party will run the GoAVSS protocol n times: once playing role of both dealer and receiver, and $n-1$ times just as a receiver, and this will yield a total of $L \cdot (n - 2t) \geq L \cdot n/3$ "processed sharings". So we naturally measure the amortized cost per "processed sharing", which in the application to threshold

signatures, represents the amortized cost per presignature of the offline phase contributed by the GoAVSS protocol (but does not include the computational cost of applying the super-invertible matrix).

Communication complexity We define *communication complexity* as the sum, over all honest parties P and all parties Q, of the total number of bits that P sends to Q over a point-to-point channel.

The amortized communication complexity per processed sharing of our GoAVSS protocol is $O(n\lambda)$, where $q < 2^\lambda$ and we assume group elements are encoded as $O(\lambda)$-bit strings. Also, the communication complexity is well balanced: each party sends (and receives) $O(\lambda)$ bits per of data per processed sharing. In fact, each party essentially transmits (and receives) a total of 18 scalars and 9 group elements per processed sharing.

Computational complexity We define *computational complexity* to be the maximum running time of any one individual party.

The amortized computational complexity per processed sharing of our GoAVSS protocol is dominated by the cost of performing $O(n/\lambda + 1)$ additions in the group E, and $O(n)$ arithmetic operations in $\mathbb{Z}_q$.

To state the computational complexity more precisely, we introduce some terminology.

Full scalar/group multiplication: This is a scalar/group multiplication, where the scalar is a secret, full-sized element of $\mathbb{Z}_q$, and the group element is the generator $\mathcal{G}$. This means that (a) we can use precomputation on $\mathcal{G}$ to make the algorithm faster, but (b) we have to be careful to use a constant-time algorithm.

For example, one simple, standard algorithm is that of Lim and Lee [LL94] (which, as noted Section 7 of [Ber02], is a special case of Pippenger's algorithm [Pip76]). It is defined in terms of parameters h and v. It precomputes a table of $2^h \cdot v$ multiplies of $\mathcal{G}$, so that one full scalar/group multiplication costs at most $\lceil n/h \rceil + \lceil \lceil n/h \rceil / v \rceil - 2$.

Tiny scalar/group multiplication: Let ρ be a parameter (usually clear from context). This is a scalar/group multiplication where the scalar is a public, random ρ-bit number, and the group element is variable but public.

Moreover, the resulting products are only used as terms in a very long summation, so special optimized algorithms may be used. For example, for fairly small ρ, one quite good algorithm for this problem is a simple "on demand" version of Straus's algorithm (discussed, for example, at the end of Section 3 of [Ber02]). This algorithm has an amortized cost per product of α additions in E, where α is the average Hamming weight of random ρ-bit integer (or even better, the average Hamming weight of the NAF encoding of such a random number).

In this "on demand" version Straus's algorithm, to compute the sum $\mathcal{H} = \sum_i e_i \mathcal{G}_i$, we write $e_i = \sum_j b_{i,j} 2^j$, where each $b_{i,j} \in \{0, 1\}$, and then compute $\mathcal{H}$ as

$$\sum_j 2^j \left(\sum_i b_{i,j} \mathcal{G}_i \right),$$

first computing, for each j, the individual sum $\sum_i b_{i,j} \mathcal{G}_i$, and then combining these with a few doublings and additions. If the e_i's are small random numbers, and the number of

terms in the original sum is very large, then the amortized cost per term is with very high probability very near the average Hamming weight of the e_i's. In a NAF encoding (see `https://en.wikipedia.org/wiki/Non-adjacent_form`), each $b_{i,j} \in \{0, \pm 1\}$, and the average Hamming weight is usually smaller.

Tiny scalar/scalar multiplication: Again, let ρ be a parameter (usually clear from context). This is a scalar/scalar multiplication where one scalar input is a public, random ρ-bit number, and the other scalar input is a secret, full-sized element of $\mathbb{Z}_q$.
Moreover, the resulting products are only used as terms in a very long summation, so special optimized algorithms may be used. In particular, the cost of such an operation is essentially that of multiplying a ρ-bit integer by a λ-bit integer (without a modular reduction).

Let σ be a statistical security parameter (typically $\sigma = 80$). Set the parameter $\rho = \lceil 3\sigma/n \rceil + 2$, used in defining tiny scalar/group and scalar/scalar multiplications, as above. The amortized computational complexity per processed sharing of our GoAVSS protocol can be stated as follows.

- $3/n$ full scalar/group multiplications,
- 3 tiny scalar/group multiplications, and
- $4n$ tiny scalar/scalar multiplications.

In addition to the communication and computational complexity of our GoAVSS protocol, also of interest is the **round complexity**. This is a (reasonably small) constant. Because of the efficiency of our protocol, in terms of communication and computational complexity, the remaining challenge in achieving truly high-throughput generation of presignatures is to structure the protocol so as to keep the network and CPU bandwidth nearly fully utilized. In principle, this should be achievable via standard "pipelining" techniques, although making it work in practice could be a challenge.

2.2 Solution to Problem 2

Our solution to problem two can be stated very simply. For any $M \leq N \leq q$, we give an explicit construction of an $M \times N$ super-invertible matrix over $\mathbb{Z}_q$, along with an algorithm for multiplying this matrix on the right by an $N \times 1$ column vector over E that uses precisely $M \cdot (N-1)$ additions in the group E.

2.3 Combining the two solutions

Combining our solutions to Problems 1 an 2, we immediately get a protocol for computing presignatures for Schnorr threshold signing with the following properties:

- its amortized communication complexity per presignature is $O(n\lambda)$, i.e., $O(\lambda)$ bits per party and
- a party's amortized computational complexity per presignature is dominated by the cost of performing $O(n + \lambda/n)$ additions in the group E

The online complexity of the resulting protocol depends on a number of design choices with regard to how the Open protocol is implemented. If we insist on one round of communication in the online phase, we obtain

- an amortized communication complexity per signature of $O(n^2\lambda)$, with each party essentially transmitting n scalars per signature, and

– an amortized computational complexity per signature dominated by the cost of performing $O(n^2)$ arithmetic operations in $\mathbb{Z}_q$.

If we allows two rounds of communication in the online phase, and also allow for some batching of signing requests (which is reasonable in a heavily loaded system), we obtain

– an amortized communication complexity per signature of $O(n\lambda)$, with each party essentially transmitting 6 scalars per signature, and
– an amortized computational complexity per signature dominated by the cost of performing $O(n)$ arithmetic operations in $\mathbb{Z}_q$.

This is discussed in more detail in Section 7.1.

All of the above computational estimates assume only naive, quadratic-time polynomial arithmetic.

2.4 The rest of the paper

In Section 3, we present the precise security definitions of AVSS and GoAVSS that we will use throughout the paper. In Section 4, we review the subprotocols we make use of to build our new GoAVSS protocol. In Section 5, we present our new GoAVSS protocol, provide a careful security and complexity analysis, and look a couple of variations as well. We also present an analyze of simple variant in Section 5.4 that may be preferred in when n is very large. In Section 6, we present our new construction of a super-invertible matrix with a corresponding algorithm for applying the matrix to a vector of group elements. Finally, in Section 7, we discuss some details about the online signing phase of the threshold Schnorr signature derived from our protocols. In Section 8, we make an in-depth comparison (with numerical examples) with our closest competitor, SPRINT [BHK+23]; in Section 8.4, we look beyond SPRINT, and combine various ideas (from [GS22] as well as from this paper) to get the best possible "batch Feldman" GoAVSS protocol based on polynomial commitments, and compare its performance to that of our new GoAVSS protocol.

3 Preliminaries

3.1 Asynchronous verifiable secret sharing

We recall the notion of *asynchronous verifiable secret sharing (AVSS)*. We have n parties $P_1, \ldots, P_n$, of which at most $t < n/3$ may be corrupt. We assume *static* corruptions. Let $\mathcal{H}$ denote the indices of the honest parties, and let $\mathcal{C}$ denote the indices of the corrupt parties.

We assume the parties are connected by secure point-to-point channels, which provide both privacy and authentication. As we are working exclusively in the asynchronous communication model, there is no bound on the time required to deliver messages between honest parties.

Let $\mathbb{Z}_q$ be finite field with q elements. Let $\boldsymbol{e} = (e_1, \ldots, e_n) \in \mathbb{Z}_q^n$ be a sequence of distinct, nonzero elements of $\mathbb{Z}_q$. Let d be a positive integer, and let $\mathbb{Z}_q[x]_{<d}$ denote the $\mathbb{Z}_q$-subspace of $\mathbb{Z}_q[x]$ consisting of all polynomials of degree less than d. Let L be a positive integer.

An (n, d, L)**-AVSS protocol** over $\mathbb{Z}_q$ (with respect to $\boldsymbol{e}$) allows a dealer $D \in \{P_1, \ldots, P_n\}$ to share a polynomials $f_1, \ldots, f_L \in \mathbb{Z}_q[x]_{<d}$ in such a way that each party P_j learns only $f_1(e_j), \ldots, f_L(e_j)$. More precisely, it should satisfy a *security* property and a *completeness* property. The security property is captured by the ideal functionality $\mathfrak{F}_{\text{AVSS}}$ in Figure 1. The completeness

property is as follows: if the dealer is honest or any honest party outputs a value, then all honest parties eventually output a value. Here, "eventually" means if and when all honest parties initiate the protocol and all messages sent between honest parties are delivered.

One can show that like security, completeness is a composable property in the UC framework. That is, one can naturally define the notion of completeness for a hybrid protocol which makes use of ideal functionalities as subprotocols, and it is a simple exercise to show that if we replace these idealized subprotocols by complete, concrete protocols that securely realize these functionalities, the resulting protocol is a concrete protocol that is secure and complete. Completeness for such a hybrid protocol is the same as for a concrete protocol, except that in addition to the condition that all messages sent between honest parties have been delivered, we also add the condition that all RequestOutput messages for the ideal functionalities have been delivered — in fact, concrete protocols are really hybrid protocols with an appropriate secure messaging ideal functionality, and so this is actually the same condition.

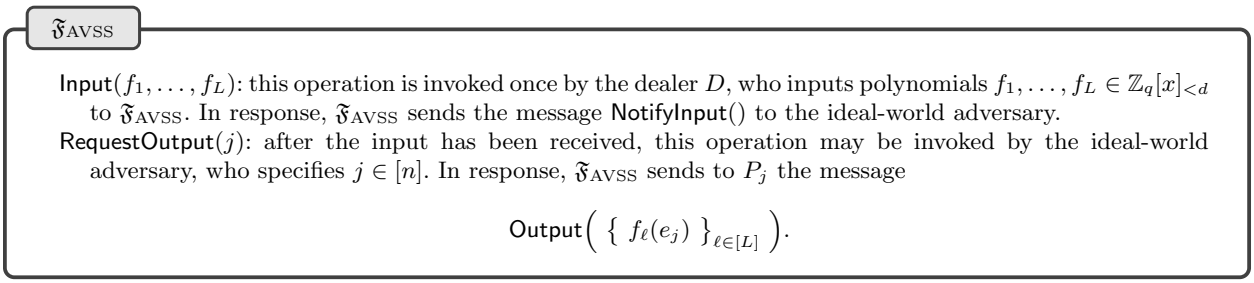

Fig. 1. The AVSS Ideal Functionality (parameterized by n, d, L, $\mathbb{Z}_q$, $\boldsymbol{e}$, and D)

3.2 Group-oriented AVSS

We now introduce the notion of a **group-oriented AVSS (GoAVSS)**. In this setting, q is a prime, and we are working over a group E of order q generated by $\mathcal{G} \in E$. We use additive notation for the group operation of E, and denote the identity element of E by $\mathcal{O}$.

Essentially, an (n, d, L)**-GoAVSS protocol** over E (with respect to $\boldsymbol{e}$) is the same as an (n, d, L)-GoAVSS protocol over $\mathbb{Z}_q$, and each party (including the adversary) also obtains $\mathcal{S}_\ell := f_\ell(0)\mathcal{G} \in E$ for each $\ell \in [L]$.

The security property of a GoAVSS protocol are captured by the ideal functionality $\mathfrak{F}_{\text{GoAVSS}}$ in Figure 2. We also require a completeness property, which is identical to that for an AVSS protocol.

4 Subprotocols

In this section, we review the subprotocols that our new GoAVSS protocol will need.

4.1 AVSS

Our GoAVSS protocol over E will be built using an ordinary AVSS protocol over $\mathbb{Z}_q$ (as defined above in Section 3.1). In principle, any such AVSS protocol could be used. However, the protocol in

$\mathfrak{F}_{\text{GoAVSS}}$

$\mathsf{Input}(f_1, \ldots, f_L)$: this operation is invoked once by the dealer D, who inputs polynomials $f_1, \ldots, f_L \in \mathbb{Z}_q[x]_{<d}$ to $\mathfrak{F}_{\text{GoAVSS}}$. In response, $\mathfrak{F}_{\text{GoAVSS}}$ sends the message $\mathsf{NotifyInput}(\{\mathcal{S}_\ell\}_{\ell\in[L]})$ to the ideal-world adversary, where $\mathcal{S}_\ell \coloneqq f_\ell(0)\mathcal{G}$, for $\ell \in [L]$.

$\mathsf{RequestOutput}(j)$: after the input has been received, this operation may be invoked by the ideal-world adversary, who specifies $j \in [n]$. In response, $\mathfrak{F}_{\text{GoAVSS}}$ sends to P_j the message

$$\mathsf{Output}\Big(\{ (\mathcal{S}_\ell, f_\ell(e_j)) \}_{\ell\in[L]} \Big).$$

Fig. 2. The GoAVSS Ideal Functionality (parameterized by n, d, L, E (which defines $\mathbb{Z}_q$), $\boldsymbol{e}$, and D)

[SS23] is well suited to the task, as it uses only "lightweight" cryptography (namely, hash functions) and is quite efficient, both in terms of communication and computational complexity, especially on the so-called "optimistic path", where no party provably misbehaves. More concretely, on the "optimistic path", if $q < 2^\lambda$, the communication complexity (total number of bits transmitted in aggregate by all honest parties)

$$6nL\lambda + O(\lambda \cdot n^2 \log n + n^3).$$

Also, the computational complexity (running time of any one individual honest party) is as follows, assuming $L = \Omega(n \log n)$:

- for a party acting in its role as a receiver, the cost of decoding, encoding, hashing, and decrypting $O(L\lambda)$ bits of data, performing $O(L(1 + n/\lambda))$ arithmetic operations in $\mathbb{Z}_q$, and generating $O(\lambda L(1 + n/\lambda))$ pseudorandom bits.
- for a party acting in its role as a dealer, the cost of encrypting, encoding, and hashing $O(nL\lambda)$ bits of data, performing $O(dL(1+n/\lambda))$ arithmetic operations in $\mathbb{Z}_q$, and generating $O(d\lambda L(1+ n/\lambda))$ pseudorandom bits.

Note that the dealer plays a role both as a dealer and as a receiver. Here, encoding and decoding refers to encoding and decoding data using an $(n, n - 2t)$-erasure code.

Note that this protocol makes use of a collision resistant hash function with a κ-bit output and a statistical security parameter σ, and we assume $\max\{\kappa, \sigma\} \leq \lambda$. This protocol uses a statistical test that has error bound of $2^n/q$, For large n, this test must be repeated several times, and this is what gives rise to the additive term n^3 appearing in the communication complexity bound and the additive term n/λ appearing in the computational complexity bounds.

Note that when the protocol falls off the "optimistic path", the communication and computational complexity may increase by a factor of $O(n)$. However, at least one misbehaving party will be identified and can be effectively removed from participating any further in the protocol.

In the above computational complexity estimates, we have not included the cost for the dealer of actually computing the shares $f_\ell(e_j)$ for $\ell \in [L]$, $j \in [n]$, which is a part of any AVSS protocol. For very large n, an asymptotically fast multi-point evaluation algorithm may be useful. However, for small to moderate sized n, it is more practical to use a simple Horner's rule evaluation, which adds to the dealer's cost $\approx dnL$ multiplications and additions mod q. Moreover, if the evaluation points are $1, \ldots, n$ (which is typical), we can run many steps of Horner without any reductions mod q, making these steps relatively inexpensive.

4.2 Reliable broadcast

A **reliable broadcast** protocol allows a sender S to broadcast a single message m to $P_1, \ldots, P_n$ in such a way that all parties are guaranteed to receive the same message. More precisely, it should satisfy a *security* property and a *completeness* property. The security property is captured by the ideal functionality $\mathfrak{F}_{\text{ReliableBroadcast}}$ in Figure 3. The completeness property says that: if one honest party outputs a message, then every honest party eventually outputs a message; moreover, if S is honest, then every honest party eventually outputs a message.

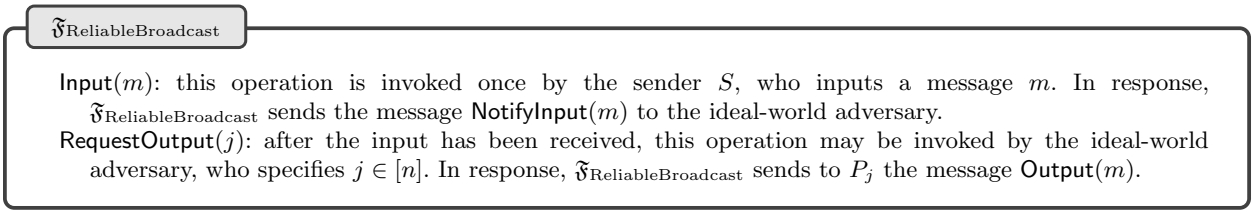

Fig. 3. The Reliable Broadcast Ideal Functionality (parameterized by S)

A well-known Reliable Broadcast protocol is due to Bracha [Bra87]. However, its communication complexity (total number of bits transmitted in aggregate by all honest parties) is $O(n^2 \cdot |m|)$. Better communication complexity can be obtained using an protocol based on erasure codes. This approach was initially considered in [CT05], who give a protocol with communication complexity

$$3n|m| + O(\kappa \cdot n^2 \log n),$$

where κ is the output length of a collision-resistant hash. Also, the computational complexity (running time of any one individual honest party) is as follows, assuming $|m| = \Omega(\kappa \cdot n \log n)$:

- for a party acting in its role as a receiver, the cost of decoding, encoding, and hashing $O(|m|)$ bits of data;
- for a party acting in its role as a dealer, the cost of encoding and hashing $O(n|m|)$ bits of data operations in $\mathbb{Z}_q$.

Note that the dealer plays a role both as a dealer and as a receiver. Here, encoding and decoding refers to encoding and decoding data using an $(n, n-2t)$-erasure code.

While there have been various subsequent improvements on reliable broadcast, [CT05] is efficient enough for our purposes.

4.3 Random Beacon

This is a protocol that reveals a value ω chosen at random from an **output space** Ω, in such a way that satisfies a *security* property and a *completeness* property. The security property is captured by the ideal functionality $\mathfrak{F}_{\text{Beacon}}$ in Figure 4. The main idea is that the adversary learns nothing about ω until after at least one honest party initiates the protocol. The completeness property says that if at least $t+1$ honest parties initiate the protocol, every honest party eventually outputs a value.

$\mathfrak{F}_{\text{Beacon}}$

Input(): This operation may be invoked once by each party P_j. If this is the first time this is invoked by any honest party, $\mathfrak{F}_{\text{Beacon}}$ chooses $\omega \in \Omega$ at random and sends NotifyInput(j, ω) to the ideal-world adversary; otherwise, $\mathfrak{F}_{\text{Beacon}}$ sends NotifyInput(j) to the ideal-world adversary.

RequestOutput(j): after the value ω has been generated, this operation may be invoked by the ideal-world adversary, who specifies $j \in [n]$. In response, $\mathfrak{F}_{\text{AVSS}}$ sends to P_j the message Output(ω).

Fig. 4. The Random Beacon Functionality $\mathfrak{F}_{\text{Beacon}}$ (parameterized by output space Ω)

A random beacon may be efficiently implemented using any AVSS protocol and any consensus protocol, using the standard technique of agreeing on a set of $t+1$ secret sharings, and then opening all of them and adding them up. This will yield a beacon output that is an element of a finite field. If the output of the random beacon needs to be longer, it can be passed through a PRG or a hash function.

5 Our new GoAVSS protocol

Our new GoAVSS protocol is presented in Figure 5.

- The protocol is parameterized by a subset $R \subseteq \mathbb{Z}_q$.
- The protocol makes use of
  - an instance of an AVSS subprotocol, which is invoked as an ideal functionality $\mathfrak{F}_{\text{AVSS}}$;
  - two instances of a reliable broadcast subprotocol, both of which are invoked as the ideal functionality $\mathfrak{F}_{\text{ReliableBroadcast}}$ (but there are really two separate instances here of this ideal functionality);
  - an instance of a random beacon, which is invoked as an ideal functionality $\mathfrak{F}_{\text{beacon}}$; this beacon outputs
$$\{\gamma_\ell^{(k)}\}_{\ell \in [L], k \in [n]},$$
where each $\gamma_\ell^{(k)} \in R \subseteq \mathbb{Z}_q$; an implementation may choose to instead use a beacon that outputs a short seed to a PRG, and then use the PRG to derive these values.

5.1 Security analysis of protocol Π_{GoAVSS1}

In the security analysis of this protocol, we make use of the following version of the Chernoff bound. Let $S_{N,p}$ be the number of successes among N independent Bernoulli trials, each with success probability p. Let M be a nonnegative integer. We define

$$\text{Tail}(N, M, p) \coloneqq \Pr[S_{N,p} \geq M]. \tag{1}$$

Then assuming $0 < p < M/N < 1$, we have

$$\text{Tail}(N, M, p) \leq 2^{-N \cdot H(M/N, p)}, \tag{2}$$

where

$$H(a, p) \coloneqq a \log_2 \left(\frac{a}{p}\right) + (1-a) \log_2 \left(\frac{1-a}{1-p}\right).$$

Π_{GoAVSS1}

The dealer $D \in \{P_1, \ldots, P_n\}$ has input $f_1, \ldots, f_L \in \mathbb{Z}_q[x]_{<d}$.

1. Dealer D:
   (a) Choose $g^{(k)} \in \mathbb{Z}_q[x]_{<d}$ at random for $k \in [n]$
   (b) Invoke the operation $\mathsf{Input}(\ \{g^{(k)}\}_{k\in[n]},\ \{f_\ell\}_{\ell\in[L]}\ )$ on $\mathfrak{F}_{\text{AVSS}}$.
   (c) Compute $\mathcal{T}^{(k)} \leftarrow g^{(k)}(0)\mathcal{G} \in E$ for $k \in [n]$ and $\mathcal{S}_\ell \leftarrow f_\ell(0)\mathcal{G} \in E$ for $\ell \in [L]$.
   (d) Invoke the operation
   $$\mathsf{Input}(\ \{\mathcal{T}^{(k)}\}_{k\in[n]},\ \{\mathcal{S}_\ell\}_{\ell\in[L]}\ )$$
   on $\mathfrak{F}_{\text{ReliableBroadcast}}$.
2. Each P_j:
   (a) Wait for $\mathfrak{F}_{\text{AVSS}}$ to deliver the message
   $$\mathsf{Output}(\ \{w_j^{(k)}\}_{k\in[n]},\ \{v_{\ell,j}\}_{\ell\in[L]}\ )$$
   and for $\mathfrak{F}_{\text{ReliableBroadcast}}$ to deliver the message
   $$\mathsf{Output}(\ \{\mathcal{T}^{(k)}\}_{k\in[n]},\ \{\mathcal{S}_\ell\}_{\ell\in[L]}\ ).$$
   (b) Invoke the operation $\mathsf{Input}()$ on $\mathfrak{F}_{\text{Beacon}}$.
3. Each P_j: Wait for $\mathfrak{F}_{\text{Beacon}}$ to deliver the message
   $$\mathsf{Output}(\ \{\gamma_\ell^{(k)}\}_{\ell\in[L],k\in[n]}\ ),$$
   with each $\gamma_\ell^{(k)} \in R \subseteq \mathbb{Z}_q$.
4. Dealer D:
   (a) For each $k \in [n]$, compute
   $$h^{(k)} \leftarrow g^{(k)} + \sum_{\ell\in[L]} \gamma_\ell^{(k)} \cdot f_\ell \in \mathbb{Z}_q[x]_{<d}.$$
   (b) Invoke the operation $\mathsf{Input}(\ \{h^{(k)}\}_{k\in[n]}\ )$ on $\mathfrak{F}_{\text{ReliableBroadcast}}$.
5. Each P_j:
   (a) Wait for $\mathfrak{F}_{\text{ReliableBroadcast}}$ to deliver the message $\mathsf{Output}(\ \{h^{(k)}\}_{k\in[n]}\ )$.
   (b) For each $k \in [n]$, check that $h^{(k)} \in \mathbb{Z}_q[x]_{<d}$ and that
   $$h^{(k)}(e_j) = w_j^{(k)} + \sum_{\ell\in[L]} \gamma_\ell^{(k)} \cdot v_{\ell,j}.$$
   (c) Check that
   $$h^{(j)}(0)\mathcal{G} = \mathcal{T}^{(j)} + \sum_{\ell\in[L]} \gamma_\ell^{(j)} \mathcal{S}_\ell.$$
   (d) If these checks do not pass then *abort*. Otherwise:
      i. send a "vote" message to all parties;
      ii. wait for "vote" messages from $n - t$ distinct parties;
      iii. output $\{(\mathcal{S}_\ell, v_{\ell,j})\}_{\ell\in[L]}$.

Fig. 5. A GoAVSS protocol

Theorem 5.1 (Security of Π_{GoAVSS1}). *Suppose that $t < d \leq n-2t$ and $|R| \geq 2^{\rho}$. Then assuming that*

$$\text{Tail}(n-t, n-2t, 1/|R|) \leq 2^{-n\cdot(\rho-2)/3} \tag{3}$$

is negligible, protocol Π_{GoAVSS1} securely and completely realizes $\mathfrak{F}_{\text{GoAVSS}}$ in the $(\mathfrak{F}_{\text{AVSS}}, \mathfrak{F}_{\text{Beacon}}, \mathfrak{F}_{\text{ReliableBroadcast}})$-hybrid model.

Proof. If the dealer is honest, we just have to show how to simulate the information that the protocol leaks to the adversary, given the information leaked by $\mathfrak{F}_{\text{GoAVSS}}$, which is $\{v_{\ell,j}\}_{\ell\in[L],j\in\mathcal{C}}$ and $\{\mathcal{S}_\ell\}_{\ell\in[L]}$, and given the fact that the simulator is also allowed to generate the output $\{\gamma_\ell^{(k)}\}_{\ell\in[L],k\in[n]}$ of the random beacon in advance. This is straightforward. For each $k \in [n]$, the simulator can simply generate the polynomial $h^{(k)}$ at random, and then compute the adversary's shares of $g^{(k)}$ as

$$w_j^{(k)} \leftarrow h^{(k)}(e_j) - \sum_{\ell\in[L]} \gamma_\ell^{(k)} v_{\ell,j}$$

for $j \in \mathcal{C}$, and the group element $\mathcal{T}^{(k)}$ as

$$\mathcal{T}^{(k)} \leftarrow h^{(k)}(0)\mathcal{G} - \sum_{\ell\in[L]} \gamma_\ell^{(k)} \mathcal{S}_\ell.$$

Now assume the dealer is corrupt. Consider the point in time where some honest party collects "vote" messages from $n-t$ distinct parties in Step 5(d) for the very first time. Let $\mathfrak{E}$ be the event that $\mathcal{S}_\ell \neq f_\ell(0)\mathcal{G}$ for some $\ell \in [L]$. So long as $\mathfrak{E}$ does not occur, correctness and completeness will hold. We argue that $\Pr[\mathfrak{E}]$ is at most (3), from which the theorem will follow.

By the logic of the protocol, at this point in time, at least $n-2t$ out of a set of $n-t$ honest parties must have found that all checks passed in Step 5. Call these parties "accepting" parties. Since $n-2t \geq d$, and all polynomials here have degree less than d, this means all of the polynomials $h^{(k)}$ for $k \in [n]$ were correctly computed, that is,

$$h^{(k)} = g^{(k)} + \sum_{\ell\in[L]} \gamma_\ell^{(k)} f_\ell$$

for all $k \in [n]$. Therefore, for each "accepting" party P_j, we have

$$\mathcal{T}^{(j)} + \sum_{\ell\in[L]} \gamma_\ell^{(j)} \mathcal{S}_\ell = h^{(j)}(0)\mathcal{G} = g^{(j)}(0)\mathcal{G} + \sum_{\ell\in[L]} \gamma_\ell^{(j)} f_\ell(0)\mathcal{G}.$$

Here, the first equality holds by the logic of test in Step 5(c), and the second holds because of the fact that the polynomials $h^{(k)}$ for $k \in [n]$ were correctly computed.

For each $k \in [n]$, if the values

$$\{f_\ell\}_{\ell\in[L]},\ \{\mathcal{S}_\ell\}_{\ell\in[L]},\ \mathcal{T}^{(k)},\ \text{and},\ g^{(k)}$$

are fixed, and $\mathcal{S}_\ell \neq f_\ell(0)\mathcal{G}$ for some $\ell \in [L]$, and if we choose the values $\gamma_\ell^{(k)} \in R$ at random for $\ell \in [L]$, then the probability that

$$\mathcal{T}^{(k)} + \sum_{\ell\in[L]} \gamma_\ell^{(k)} \mathcal{S}_\ell = g^{(k)}(0)\mathcal{G} + \sum_{\ell\in[L]} \gamma_\ell^{(k)} f_\ell(0)\mathcal{G}$$

is clearly at most $2^{-\rho}$.

Thus, if $\mathfrak{E}$ is to occur, at least $n-2t$ parties out of a set of $n-t$ honest parties must each find that a certain event occurs, where these events are independent of one another and each occurs with probability $\leq 2^{-\rho}$. Therefore, $\Pr[\mathfrak{E}] \leq \text{Tail}(n-t, n-2t, 2^{-\rho})$, and the bound (3) follows from the Chernoff bound (2). □

Remark 5.1. The error bound (3) is the probability that a corrupt dealer breaks either the correctness or completeness property of the protocol. So to achieve an error bound of $2^{-\sigma}$, where σ is a statistical security parameter, we can set $\rho := \lceil 3\sigma/n \rceil + 2$. In a typical implementation, R will be chosen to be the set of all ρ-bit numbers. For a typical setting such as $\sigma = 80$, and with a network of size, say, $n = 49 = 3 \cdot 16 + 1$, we can set $\rho = 7$. By using such small numbers, we get a protocol that is very attractive from a computational complexity perspective. We analyze the computational complexity of this protocol in detail below in Section 5.2. □

Remark 5.2. Note that the analysis here is in the static corruption model, and the error bound (3) reflects that. If we carried out the analysis in an adaptive corruption model, it would be

$$\text{Tail}(n, n-2t, 1/|R|) \leq 2^{-n\cdot(\rho-3)/3}.$$

Indeed, in a adaptive corruption model, before corrupting any parties, the adversary could then hope that $n-2t$ of these parties cast a positive vote, and then corrupt t of the remaining parties and make them cast positive votes as well. While our recommended implementation of the AVSS subprotocol in [SS23] is only analyzed in the static corruption model, it is suggested there that it may be secure against adaptive corruptions in the random oracle model. If this adaptive error bound is used, we would have to add 1 to the value of ρ to achieve the same level of security, so this would generally have a mild impact on the performance of the protocol. □

Remark 5.3. The random beacon may output a short seed, which is passed to a PRG to derive all of the challenge values $\gamma_\ell^{(k)}$. When we do this, we need to add to the error probability in Theorem 5.1 a term that measures advantage of a certain adversary (whose running time is essentially the same as that of the original adversary) in distinguishing the PRG output from random. □

Remark 5.4. We could replace the voting step in Step 5(d) by a simple Bracha-like agreement protocol (see, for example, Section 3.3 of [SS23]). This would make the completeness property of the protocol unconditional, at the cost of an extra round of communication. □

5.2 Complexity analysis of protocol Π_{GoAVSS1}

We analyze both computational and communication complexity. For each complexity metric, we consider the cost contributed by the body of the protocol, as well the cost contributed by the recommended implementation of the subprotocols. For subprotocols that have an "optimistic path", where no party provably misbehaves, we consider only the cost on this optimistic path. Arguably, over a long run of the system where parties that provably misbehave are effectively removed from the system, this is the only cost that matters. To simplify matters, we assume the $d = t+1$ and $n = 3t+1$.

We will also consider the "amortized" cost, in two different senses.

– One is the amortized cost per "raw sharing" in one run of Π_{GoAVSS1}, where one party is the designated dealer, and the other parties are receivers. As there are L sharings generated in one run of Π_{GoAVSS1}, and it is assumed that L is very large, we can effectively ignore the cost of any protocol steps and subprotocols whose cost does not grow with L (such as the random beacon).
– The other is the amortized cost per "processed sharing". This is the sum of the amortized cost per "raw sharing" over n runs of the protocol, with different designated dealer in each run, divided by the number of sharings output by the batch randomness extraction procedure (which is at least $t+1$). This is ultimately the number of interest in applications.

Throughout this section, $\lambda := \lceil \log_2(q) \rceil$ and σ is a statistical security parameter. We assume $\sigma \leq \lambda$. In typical settings, we will have $\lambda = 256$ and $\sigma = 80$.

Communication complexity As usual, we measure communication complexity as the sum, over all honest parties P and all parties Q, of the total number of bits that P sends to Q over a point-to-point channel. With AVSS and reliable broadcast implemented as described in Section 4, it is easily seen that the amortized communication complexity per processed sharing is $O(n\lambda)$. This assumes elements of E can be encoded in $O(\lambda)$ bits. Also, the communication complexity is well balanced: each party transmits (and receives) $O(\lambda)$ bits of data per processed sharing. In fact, it is easy to see from the estimates given in Section 4 that each party essentially transmits (and receives) a total of 18 scalars and 9 group elements per processed sharing.

Computational complexity To state computational costs, we use the terms "full scalar/group multiplication", "tiny scalar/group multiplication", and "tiny scalar/scalar multiplication" introduced in Section 2.1.

Amortized cost per raw sharing.

– For a party acting in its role as a receiver, the amortized cost per raw sharing in the body of our protocol is:
  • 1 tiny scalar/group multiplication and addition.
  • n tiny scalar/scalar multiplications.
– For a party acting in its role as a dealer, the amortized cost per raw sharing in the body of our protocol is:
  • 1 full scalar/group multiplication.
  • $n(t+1)$ tiny scalar/scalar multiplications.

Note that the dealer plays a role both as a dealer and as a receiver.

Amortized cost per processed sharing. The amortized cost per processed sharing is equal to $n/(t+1)$ times the amortized receiver cost per raw sharing plus $1/(t+1)$ times the amortized dealer cost per raw sharing. This gives us an amortized cost per processed sharing of:

– $1/(t+1)$ full scalar/group multiplications.
– 3 tiny scalar/group multiplications.
– $4n$ tiny scalar/scalar multiplications.

Example 5.1. Consider $n = 49 = 3 \cdot 16 + 1$ and we set the statistical security parameter $\sigma := 80$. So we set $\rho := 7$. Let us also assume that $\lambda = 256$. We compute the amortized cost per processed sharing. This is:

- 1/17 full scalar/group multiplications.
  Suppose we use Lim and Lee's algorithm as suggested in Section 2.1, with parameters $h = 5$ and $v = 2$. This results in a table of size $2^5 \cdot 2 = 64$, which is reasonable small (and so more ammenable to constant-time lookup techniques). With these patrameters, we can compute a full scalar/group multiplication $\lceil 256/5 \rceil + \lceil \lceil 256/5 \rceil / 2 \rceil - 2 = 76$ group additions.[3] Dividing by 17, we get an amortized cost of less than 5 group additions.
- 3 tiny scalar/group multiplications.
  Suppose we use the on-demand Straus algorithm as suggested in Section 2.1, and NAF encode the 7-bit scalar (assuming group negation is free as it is for an elliptic curve). The expected Hamming weight of such a NAF-encoded 7-bit scalar is ≈ 2.77 (obtained by direct computation). this leads to an amortized cost per processed sharing of $\approx 8.3 \approx 2.77 \cdot 3$ group additions. While this estimate is based on an average-case analysis, we are only performing these computations on very large batches pseudorandom scalars, and so the actual cost will be very close to the average-case cost with overwhelming probability. Note that other algorithms here may give slightly better performance (see Remark 8.2).
- 196 tiny scalar/scalar multiplications.

So we get a total of $\approx 13.3 = 5 + 8.3$ group additions plus 196 tiny scalar/scalar multiplications.

Now let us add in the amortized cost of applying our new construction of a super-invertible matrix (see Section 6). This is $2t = 32$ group additions and $2t = 32$ additions in $\mathbb{Z}_q$ per processed sharing. So this gives us:

- $45.3 = 13.3 + 32$ group additions;
- 196 tiny scalar/scalar multiplications;
- 32 additions in $\mathbb{Z}_q$.

Note that we perform no more than about 5 times as many tiny scalar/scalar multiplications as we do group additions. However, each of these is tiny scalar/scalar multiplications relatively cheap (essentially, the cost of multiplying a ρ-bit integer by a λ-bit integer, with no modular reduction), so that in any reasonably optimized implementation, the cost of these should be significantly less than the cost of the group additions.

Indeed, we conservatively estimate that each tiny scalar/scalar multiplication is about 1/40th the cost of a group addition over an elliptic curve such as `secp256k1`. In such a group addition, we have to perform several multiplications mod p, where p is also a 256-bit prime. We estimate the number of such multiplications mod p to be about 10 (although this depends on many factors). On a 64-bit machine, assuming 256 bit numbers are represented as four 64-bit limbs (i.e., four base-2^{64} digits), one multiplication mod p costs four 1-limb-by-4-limb multiplications. While reduction mod p is not free, we ignore that cost here. In contrast, a tiny scalar/scalar multiplication should take about the same cost as one 1-limb-by-4-limb multiplication. This is where we get the factor of $40 = 10 \cdot 4$.[4]

[3] This estimate is also roughly consistent with timing measurements based on `https://github.com/bitcoin-core/secp256k1`.

[4] Also note that on CPUs that support Intel's AVX512 instruction set with integer fused multiply-and-add (IFMA) instructions, a single tiny scalar/scalar multiplication can take as little as 3 cycles, with all instruction latencies masked. This is done using a "redundant representation", so that basically the cost is to unpack the 7-bit scalar from a word (shift and mask), broadcast it to all 64-bit-slots of a 512-bit SIMD register, and accumulate the product with one IFMA.

Note that if we use the adaptive error bound as discussed in Remark 5.2, we have to use $\rho = 8$ rather than $\rho = 7$. The expected NAF Hamming weight increases from ≈ 2.77 to ≈ 3.11, so an increase of ≈ 0.34. Hence, we have to add $0.34 \cdot 3 \approx 1$ to our amortized addition cost. Thus, the impact of this is very small. □

Other costs. The above cost analysis left out some costs that we shall now discuss. As we shall see, these other costs should not have any significant impact on the overall performance.

Cost of subprotocols. In terms of amortized computational complexity per sharing (raw or processed), the only subprotocols that matter are AVSS and reliable broadcast (specifically, the first reliable broadcast invoked by the dealer in Step 1(d)). We assume all subprotocols are implemented as in [SS23]. We focus here on the AVSS subprotocol, as this is themost expensive. This protocol uses a statistical test that has error bound of $2^n/q$. For n of the size we are mostly interested in here, this error probability will be sufficiently small, and the test only needs to be repeated once. If we count the number of arithmetic operations in $\mathbb{Z}_q$ performed by this AVSS protocol, the amortized number of such operations per processed sharing is easily seen to be 4. Thus, this will not have any measurable impact on the overall GoAVSS protocol.

For larger values of n, this test has to be repeated several times. However, for $\lambda = 256$ and $\sigma = 80$, and for n up to 1000, we have to repeat the test at most 5 times, giving an amortized cost per processed sharing of 20 arithmetic operations in $\mathbb{Z}_q$, which still has no measurable impact on the overall GoAVSS protocol.

The above does not take into account the cost for the dealer of actually computing the shares $f_\ell(e_j)$ for $\ell \in [L]$, $j \in [n]$, which is a part of any AVSS protocol. As discussed in Section 4.1, for small to moderate size n, we can use use Horner's rule, which will add the cost of performing $\approx n$ multiplications and additions mod q to the amortized cost per processed sharing. Moreover, if the evaluation points are $1, \ldots, n$ (which is typical), we can run many steps of Horner without any reductions mod q, making these steps relatively inexpensive (similar in cost to a tiny scalar/scalar multiplication).

Cost of erasure codes. We also consider the cost of performing the encoding and decoding operations for the erasure codes used in the reliable broadcast protocol and similar subprotocols used in the AVSS protocol in [SS23]. Note that all of these encoding and decoding operations work only on public data, so there are no restrictions on the type of codes and algorithms that may be used. The computational cost of these protocols is very implementation dependent. But even if we use naive, quadratic-time algorithms, the amortized computational cost per processed sharing will be essentially $O(nw)$ word operations per processed sharing, where w is the number of machine words needed to represent q. Here, the implied big-O constant is quite reasonable. Note that such erasure codes generally will not perform arithmetic operations in $\mathbb{Z}_q$ — rather, they work over a domain that is conveniently suited to the machine instruction set. For example, chip manufacturers have recently added special-purpose instructions to more efficiently support the type of operations needed to implement erasure codes [DGK18a]. For larger n, asymptotically fast algorithms may be used. In any case, it seems likely that in any good implementation of erasure codes, this cost will not greatly impact the overall cost of GoAVSS.

Cost of PRGs and hashing. We also must consider the cost of generating pseudorandom bits as a the output for the random beacon, both in the GoAVSS protocol and in the implementation of the AVSS subprotocol. In typical settings, where $n = O(\lambda)$, the amortized number of bits per processed sharing that need to generated is $O(\lambda)$. It seems likely that with any good implemen-

tation of a pseudorandom bit generator, especially on a CPU with good cryptographic hardware support [FLdO18,DGK18b], this cost will not greatly impact the overall cost of GoAVSS.
Similarly, in the underlying AVSS and reliable broadcast subprotocols, much of the data sent and received needs to be hashed. The amortized number of bits per processed sharing that need to be hashed is $O(\lambda)$. It seems likely that with any good implementation of a hash function, especially on a CPU with good cryptographic hardware support [FLdO18], this cost will not greatly impact the overall cost of GoAVSS.

5.3 A variation with packing

As discussed above in Section 8.3, there are situations where we may want to do "packed" secret sharing. The idea is that instead of having a polynomial $f \in \mathbb{Z}_q[x]$ encode a single secret as the value of f at the point 0, we instead have it encode several secrets, as the value of f at several points, say $e'_1, \ldots, e'_p$. In this setting, we can easily generalize the notion of a "packed" GoAVSS protocol. It is an easy exercise to generalize our GoAVSS protocol to this case as well. Instead of broadcasting $\{g^{(k)}(0)\mathcal{G}\}_k$ and $\{f_\ell(0)\}_\ell$ in Step 1(d), the dealer would instead broadcast $\{(g^{(k)}(e'_1)\mathcal{G}, \ldots, g^{(k)}(e'_p))\}_k$ and $\{(f_\ell(e'_1), \ldots, f_\ell(e'_p))\}_\ell$. The test in Step 5(c) would then be performed p times, once for each evaluation point $e'_1, \ldots, e'_p$ (but we use the same values $\gamma_\ell^{(k)}$ for each test). The error probability (3) would have to be replaced with

$$\mathrm{Tail}(n-t, n-2t, p/|R|).$$

If we now amortize over the number p of packed secrets, the amortized number of group operations does not change at all (but the fact that the error bound increases means that we may have to increase slightly the size of ρ to get the same error bound, so it will actually increase if we want to maintain the same overall error bound). However, the amortized cost of all other operations will decrease by a factor of p.

5.4 A variation for large n

For large n, especially $n \gg \log_2(q)$, the following variation of Π_{GoAVSS1} may be preferred.

Suppose that instead of having k range over $[n]$ in Π_{GoAVSS1}, it instead ranges over $[K]$ for some parameter $K \geq 1$. Moreover, in Step 5(c) of the protocol, Party P_j checks that

$$h^{(k)}(0)\mathcal{G} = \mathcal{T}^{(k)} + \sum_{\ell \in [L]} \gamma_\ell^{(k)} \mathcal{S}_\ell, \tag{4}$$

where $k := (j \bmod K) + 1 \in [K]$. Let us call this variation Π_{GoAVSS2}.

Theorem 5.2 (Security of Π_{GoAVSS2}). *Suppose that $t < d \leq n - 2t$ and $|R| \geq 2^\rho$. Then assuming that*

$$\mathrm{Tail}\left(\ K,\ \left\lceil\ (n-2t)\ /\ \lceil n/K \rceil\ \right\rceil,\ 1/|R|\ \right) \tag{5}$$

is negligible, protocol Π_{GoAVSS2} securely and completely realizes $\mathfrak{F}_{\mathrm{GoAVSS}}$ in the $(\mathfrak{F}_{\mathrm{AVSS}}, \mathfrak{F}_{\mathrm{Beacon}}, \mathfrak{F}_{\mathrm{ReliableBroadcast}})$-hybrid model.

Proof. The main thing we have to do is to bound the probability that a corrupt dealer "wins" by breaking either the correctness or completeness property of the protocol. As it simplifies the proof, we will assume the adversary adaptively corrupts parties. For an index $k \in [K]$, let us say that party P_j is "associated with" k if $k = (j \bmod K) + 1$, and let us say that k is "good" if the equality (4) holds. To win, there must be at least $n - 2t$ parties associated with good indices, and the adversary can corrupt t other parties. Since there are at most $\lceil n/K \rceil$ parties associated with any one index, this means there must be at least $\lceil (n-2t)/\lceil n/K \rceil \rceil$ good indices. □

Remark 5.5. The error bound (5) is the probability that a corrupt dealer breaks either the correctness or completeness property of the protocol.

Remark 5.6. The point of this protocol is that when $n = 3t + 1$ amortized computational cost of this protocol per processed sharing becomes

- $1/(t+1)$ full scalar/group multiplications,
- 3 tiny scalar/group multiplications,
- $4K$ tiny scalar/scalar multiplications,

instead of

- $1/(t+1)$ full scalar/group multiplications,
- 3 tiny scalar/group multiplications,
- $4n$ tiny scalar/scalar multiplications.

However, the value of ρ defining the size of the "tiny" scalars may be smaller in Π_{GoAVSS1} than in Π_{GoAVSS2}.

Remark 5.7. Note that for $K = 1$, the error bound (5) is $2^{-\rho}$. This protocol is already interesting, as we only need to compute scalar/group multiplications with ρ-bit scalars to obtain an error probability of $2^{-\rho}$.

Example 5.2. Suppose $n = 1000 = 3 \cdot 333 + 1$ and $q < 2^{256}$. Suppose we want an error probability of 2^{-80}. Let us set $K = 50$. As in Example 5.1, we assume one full scalar/group multiplication costs 76 group additions. In this case, the error bound (5) is

$$\text{Tail}(50, 17, 1/2^{\rho}) \leq 2^{-50(0.34\rho - 1)}.$$

To bound this by 2^{-80}, we can set $\rho := 8$. As we saw in Example 5.1, for this value of ρ, we can bound the cost of a tiny scalar/group multiplication by ≈ 3.11 group operations. This leads to an amortized cost per processed sharing of about

$$76/334 + 3 \cdot 3.11 \approx 9.6$$

group operations and plus 200 tiny scalar/scalar multiplications, which is actually less than the cost that we calculated in Example 5.1 for $n = 49$.

If we extend the above example, keeping the group size and the statistical security parameter the same, and let n increase, essentially the same computational complexity bound holds. That is, the computational cost does not increase at all as n increases. Of course, in the context of an application such as threshold Schnorr, one has to include the cost applying a super-invertible matrix, which will dominate the overall cost, even using our new construction in Section 6.

6 Super-invertible matrices from Pascal

Let $A \in \mathbb{Z}_q^{M\times N}$ be a matrix with M rows and $N \geq M$ columns. The matrix A is called **super-invertible** if every collection of M of its columns is linearly independent.

The well-known symmetric Pascal matrix $S_N \in \mathbb{Z}^{N\times N}$ is an $N \times N$ matrix whose i, j entry is defined to be

$$S_{i,j} \coloneqq C_{i+j,i} = C_{i+j,j} = \frac{(i+j)!}{i!j!},$$

where the indices i, j start at 0 and $C_{n,k}$ is the binomial coefficient

$$C_{n,k} = \binom{n}{k}.$$

For example,

$$S_4 = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 \\ 1 & 3 & 6 & 10 \\ 1 & 4 & 10 & 20 \end{pmatrix}.$$

Define the matrix $S_{M,N,q} \in \mathbb{Z}_q^{M\times N}$ to be the matrix consisting of the first M rows of S_N with entries mapped from $\mathbb{Z}$ to $\mathbb{Z}_q$.

Theorem 6.1 (The Pascal matrix is super-invertible). *Assuming $q \geq N$, the matrix $S_{M,N,q}$ is super-invertible.*

Proof. First, for $i = 0, 1, \ldots, M-1$ define the polynomial

$$p_i(x) \coloneqq \frac{(x+1)(x+2)\cdots(x+i)}{1 \cdot 2 \cdot \cdots \cdot i} \in \mathbb{Z}_q[x].$$

Note that since $M \leq N \leq q$, the denominator in the definition of $p_i(x)$, which is the image of $i!$ in $\mathbb{Z}_q$, is nonzero and hence invertible. So we see that $p_i(x)$ is a polynomial of degree i.

Second, for $j = 0, 1, \ldots, N-1$, observe that the i, j entry of $S_{M,N,q}$ is equal to $p_i(j)$. This follows from the definition, as the i, j entry of $S_{M,N,q}$ is

$$\frac{(i+j)!}{i!j!} = \frac{(j+1)(j+2)\cdots(j+i)}{1 \cdot 2 \cdot \cdots \cdot i} = p_i(j).$$

So now consider any subset $\mathcal{J} \subseteq \{0, \ldots, N-1\}$ of cardinality M and form the matrix $R \in \mathbb{Z}_q^{M\times M}$ consisting of the columns of $S_{M,N,q}$ indexed by $j \in \mathcal{J}$. By the above observations, we can write

$$R = P \cdot V,$$

where $P \in \mathbb{Z}_q^{M\times M}$ is the matrix whose ith row, for $i = 0, 1, \ldots, M-1$, is the coefficient vector of the polynomial $p_i(x)$, and

$$V = \begin{pmatrix} 1 & 1 & \ldots & 1 \\ j_1 & j_2 & \cdots & j_M \\ \vdots & \vdots & & \vdots \\ j_1^{M-1} & j_2^{M-1} & \cdots & j_M^{M-1} \end{pmatrix} \in \mathbb{Z}_q^{M\times M}$$

is a Vandermonde matrix. Since $N \leq q$, the entries $j_1, \ldots, j_M \in \mathcal{J}$ are distinct when mapped to $\mathbb{Z}_q$ and hence V is nonsingular. Since P is a lower diagonal matrix with nonzero entries along the diagonal, P is also nonsingular. Since R is a product of nonsingular matrices, it follows that R is nonsingular, which proves the theorem. □

Now suppose we are given as input a column vector of group elements $\mathbf{x} = (\mathcal{X}_0, \ldots, \mathcal{X}_{N-1})^\intercal \in E^{N\times 1}$ and want to compute a column vector of group elements $\mathbf{y} = (\mathcal{Y}_0, \ldots, \mathcal{Y}_{M-1})^\intercal \in E^{M\times 1}$, where $\mathbf{y} = S_{M,N,q}\mathbf{x}$. We next show how this can be done using $M \cdot (N-1)$ group additions.

Let

$$Q \coloneqq \begin{pmatrix} 1 & 1 & 1 & \ldots \\ 0 & 1 & 1 & \ldots \\ 0 & 0 & 1 & \ldots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix} \in \mathbb{Z}_q^{N\times N}.$$

Given a column vector $\mathbf{x} \in E^{N\times 1}$, we can compute $\mathbf{z} = Q\mathbf{x} \in E^{N-1}$ using just $N-1$ group additions. Indeed, if $\mathbf{x} = (\mathcal{X}_0, \ldots, \mathcal{X}_{N-1})^\intercal$ and $\mathbf{z} = (\mathcal{Z}_0, \ldots, \mathcal{Z}_{N-1})^\intercal$, we can compute $\mathcal{Z}_{N-1} \leftarrow \mathcal{X}_{N-1}$ and $\mathcal{Z}_j \leftarrow \mathcal{X}_j + \mathcal{Z}_{j+1}$ for $j = N-2$ down to 0.

Our algorithm for computing $\mathbf{y} = S_{M,N,q}\mathbf{x}$ is based on the following fact:

Lemma 6.1. *For $i = 0, 1, 2, \ldots$, we have*

$$Q^{i+1} = \begin{pmatrix} C_{i,i} & C_{i+1,i} & C_{i+2,i} & C_{i+3,i} & \cdots \\ 0 & C_{i,i} & C_{i+1,i} & C_{i+2,i} & \cdots \\ 0 & 0 & C_{i,i} & C_{i+1,i} & \cdots \\ \vdots & \vdots & & \ddots & \end{pmatrix}.$$

Proof. This follows by induction on i. The base case $i = 0$ is clear from the definition of Q. Assuming the statement holds for $i \geq 0$, so that Q^{i+1} is of the prescribed form, and writing $Q^{i+2} = Q \cdot Q^{i+1}$, the result is immediate from the so-called "hockey-stick identity", which says that

$$\sum_{\ell=k}^{n} C_{\ell,k} = C_{n+1,k+1}$$

for all $0 \leq k \leq n$. □

Note that for $i = 0, 1, 2, \ldots$, the top row of Q^{i+1} is equal to the ith row of S_N. Based on this, we get the following algorithm. The input is a column vector of group elements $\mathbf{x} \in E^{N\times 1}$. The output is the column vector of group elements $\mathbf{y} \in E^{M\times 1}$, where $\mathbf{y} = S_{M,N,q}\mathbf{x}$. For any such column vector $\mathbf{z}$ of group elements, we write $\mathbf{z}[i]$ for the ith entry of $\mathbf{z}$, where the index i starts at 0.

$\mathbf{z} \leftarrow \mathbf{x}$
for $i = 0$ to $M-1$ do
    $\mathbf{z} \leftarrow Q\mathbf{z}$
    $\mathbf{y}[i] \leftarrow \mathbf{z}[0]$

Correctness follows from the above discussion. As we already observed, each execution of the step $\mathbf{z} \leftarrow Q\mathbf{z}$ takes $N-1$ group additions, and this step is executed M times, for a total of $M \cdot (N-1)$ group additions.

Remark 6.1 (Parallel computation). As written, the above algorithm appears inherently sequential. However, it can be parallelized. Define $\mathbf{z}^{(i)}$ to be the value of $\mathbf{z}$ at the beginning of loop iteration i. So we have

$$\mathbf{z}^{(i+1)}[N-1] = \mathbf{z}^{(i)}[N-1]$$

and

$$\mathbf{z}^{(i+1)}[j] = \mathbf{z}^{(i)}[j] + \mathbf{z}^{(i+1)}[j+1]$$

for $j = N-2$ down to 1.

It follows that the algorithm can be programmed to run in $(N-1)+(M-1)$ parallel rounds:

- In the first round, we compute the group element $\mathbf{z}^{(1)}[N-2]$.
- In the second round, we compute the group elements $\mathbf{z}^{(1)}[N-3], \mathbf{z}^{(2)}[N-2]$ in parallel.
- In the third round, we compute the group elements $\mathbf{z}^{(1)}[N-4], \mathbf{z}^{(2)}[N-3], \mathbf{z}^{(3)}[N-2]$ in parallel.
- And so on.

□

7 Some details on the online phase of the signing protocol

7.1 Opening a shared secret: error correction and batching

Since our construction in the offline phase produces shared secrets without corresponding polynomial commitments, to the protocol to open a shared secret must rely on error correcting codes — specifically, Reed-Solomon codes.

The fact that $n > 3t$ means that we can use the well-known technique of **online error correction**. The paper [CP17] contains a very nice description of this technique. We recall here a few of the details. Assume, for simplicity, that $n = 3t+1$ and that the secret shares are the values of a polynomial of degree at most t over $\mathbb{Z}_q$.

Suppose every party sends its share of a secret to a party P. Party P can wait for $2t+1$ shares. Upon receiving these, he can interpolate through the first $t+1$ shares to get a polynomial f of degree at most t, and then check that f is consistent with the remaining shares. If this check passes, P can be sure that f is correct and so the secret is $f(0)$. If this check fails, P can be sure that one of the $2t+1$ shares it has received is incorrect. On the one hand, P cannot be sure which share is incorrect; on the other hand, P can certainly afford to wait for another share from an honest party. The details of the protocol from this stage forward do not matter too much (it involves running a Reed-Solomon decoder a number of times). Eventually, not only will P recover the secret, it will also find out who was lying in the initial stage. Thus, while a corrupt party may get away with lying to P once, and forcing him to do extra work, he will never bother P again — P may safely ignore him from that point on. Thus, in a long run of the system, the cost of the initial stage, which is not much more expensive than a standard polynomial interpolation, is all that matters — the cost of the Reed-Solomon decoders does not matter at all.

Now suppose we want to publicly open a shared secret, so that every party broadcasts its share, and every party recovers the secret as above. The communication cost is $O(n^2\lambda)$, assuming $q < 2^\lambda$. Also, the computation costs $O(n^2)$ arithmetic operations in $\mathbb{Z}_q$, assuming a naive algorithm for polynomial interpolation. Both of these costs can be reduced by a factor of $O(n)$ by using a well-know "batching" technique, also described in [CP17]. To exploit this technique, we need to process batches of at least $t+1$ openings at a time. There is a penalty for this, however — opening such a batch of shared secrets now costs two rounds of communication, instead of 1.

Consider the online signing phase for the threshold Schnorr protocol such as the one obtained from our constructions here. In a heavily loaded system, it may make sense to exploit this batching technique. Indeed, in some settings, the extra communication latency may not matter too much. For example, for a distributed signing service driven by a blockchain, where both signing requests and results are placed on the blockchain, the extra latency incurred by this batched opening technique may be easily masked by the latency inherent in the blockchain.

7.2 Protecting against presignature attacks

As already mentioned, if presignatures are used directly, an adversary can carry out a subexponential time attack. See [Sho23] for details.

One mitigation strategy analyzed in [Sho23] (building on ideas from [BHK+23]) is to obtain a random $\delta \in \mathbb{Z}_q$ from a Random Beacon after a signing request has been made. If the presignature to be used for that signing request $([r], \mathcal{R})$ (which was visible to the adversary before making the signing request), we "tweak" it by replacing it by the presignature $([r + \delta], \mathcal{R} + \delta\mathcal{G})$, and use this "tweaked" presignature to carry out the signing request.

As already mentioned above in Section 7.1, we may wish to anyway batch signing requests in some way. If we do this, we can actually use the same value $\delta \in \mathbb{Z}_q$ and the same group element $\delta\mathcal{G}$ to tweak all the presignatures in the batch (this type of batched tweaking was not analyzed explicitly in [Sho23], but it easy to do). This allows us to amortize the cost of generating δ and computing $\delta\mathcal{G}$ over the size of the batch, We can easily implement the Random Beacon using a shared random value generated earlier by an AVSS protocol. So the cost of generating δ is just one opening of a shared secret (see above in Section 7.1 for some details on this). The cost of computing the group element $\delta\mathcal{G}$ is just one scalar/group multiplication. As the communication and computational complexity of these steps is amortized over the size of the batch, these costs can be safely ignored. However, one cost that cannot be amortized away is latency: the Random Beacon costs one round of latency. If we also implement the batching techniques in Section 7.1, this results in a latency of 3 rounds of communication to satisfy a signing request. As mentioned in Section 7.1, in some settings, such as a blockchain setting, the extra latency can be masked by other latencies.

Alternatively, if we want to minimize latency of signing requests at all costs, we can use the FROST-like mitigation analyzed in a general setting in [Sho23]. With this mitigation, we can derive δ from a hash function applied to (among other things) the message to be signed. This eliminates the extra latency incurred by the Random Beacon, but it comes at a cost. To carry out this mitigation, for one signature, we actually need two presignatures $([r], \mathcal{R})$ and $([s], \mathcal{S})$, and the "tweaked" signature used to sign the message is $([r + \delta s], \mathcal{R} + \delta\mathcal{S})$. This means that in the offline phase, the amortized cost per signature will increase by a factor of 2, as we now need two presignatures for every signature. In addition, for every signature, we now have to do a scalar/group multiplication — a rather expensive one, as now the group element is not fixed. Indeed, our offline protocol is so efficient that now the computational cost of this one scalar/group multiplication may well be more than the more the amortized computational cost of the offline phase.

Perhaps a good compromise is a "hybrid" strategy, where we produce in the offline phase a large number of presignatures to be used with the Random Beacon strategy, and a smaller number to be used with the FROST strategy. When the system is lightly loaded, we can use the FROST strategy to generate signatures, and when heavily loaded, the Random Beacon strategy. This would guarantee that under light loads (or for select users willing to pay a higher price), the system

has minimal latency, while under heavy loads, we increase latency to ensure good throughput is maintained. Although we have not carried it out in detail, it should be possible to extend the security analysis in [Sho23] to cover this hybrid strategy.

8 Comparison to other work

Up until now, the state of the art in threshold Schnorr is the SPRINT protocol [BHK+23].

We first review the SPRINT protocol at a very high level. The SPRINT protocol has a number of parameters that can be tuned. One of these parameters is an "efficiency parameter" that allows one to trade off between efficiency and resilience. For now, we focus on the setting of this parameter that (like our protocol) maintains optimal resilience, that is, any number of parties up to $t < n/3$ may be corrupt. For simplicity, we assume $n = 3t+1$ here. We also assume $q < 2^\lambda$ and that elements of E can be encoded as 2λ-bit strings (which corresponds to the standard affine representation for a point on an elliptic curve).

Like the threshold ECDSA protocol in [GS22], SPRINT makes use of the standard technique of implementing a AVSS protocol using "polynomial commitments". Unlike [GS22], SPRINT uses so-called Feldman commitments, which are perfectly binding but not perfectly hiding. For a polynomial $f = \sum_k f_k x^k \in \mathbb{Z}_q[x]$, its commitment is $\boldsymbol{\mathcal{F}} \coloneqq \sum_k \mathcal{F}_k x^k \in E[x]$, where $\mathcal{F}_k \coloneqq f_k\mathcal{G} \in E$. We also define the notation $f\mathcal{G} \coloneqq \boldsymbol{\mathcal{F}}$. The evaluation of such a polynomial commitment at a point $\alpha \in \mathbb{Z}_q$ is defined as $\boldsymbol{\mathcal{F}}(\alpha) \coloneqq \sum_k \alpha^k \mathcal{F}_k \in E$. Clearly, if $\boldsymbol{\mathcal{F}} = f\mathcal{G}$, then $\boldsymbol{\mathcal{F}}(\alpha) = f(\alpha)\mathcal{G}$. Thus, $\boldsymbol{\mathcal{F}}(\alpha)$ is a commitment to $f(\alpha)$. For the AVSS, we assume evaluation points $e_1, \ldots, e_n$, which should be distinct nonzero elements of $\mathbb{Z}_q$. In practice, one would likely use $e_j = j$ for $j \in [n]$.

In their protocol, each party P_i runs an AVSS protocol acting as a dealer where all parties $P_1, \ldots, P_n$, act as receivers. The dealer P_i chooses a random polynomial $f_i \in \mathbb{Z}_q[x]$ of degree at most t, computes $\boldsymbol{\mathcal{F}}_i \leftarrow f_i\mathcal{G} \in E[x]$, and broadcasts $\boldsymbol{\mathcal{F}}_i$ to all parties, and sends each party P_j its secret share $v_{i,j} \coloneqq f_i(e_j) \in \mathbb{Z}_q$ over a private channel. Each party P_j can verify its share $v_{i,j} \in \mathbb{Z}_q$ by checking that $v_{i,j}\mathcal{G} = \boldsymbol{\mathcal{F}}_i(e_j)$.

Then, the protocol agrees on a set of $\mathcal{I} \subseteq \{1, \ldots, n\}$ of $n - t = 2t + 1$ dealers who have had their dealings sufficiently well-validated. [BHK+23] introduces a specific agreement protocol for this purpose The details of this protocol are not significant here, except to point out that at the end of the protocol, we have a set $\mathcal{J}$ of $2t + 1$ parties, each of which is either a corrupt party or an honest party that holds shares of all sharings produced by the dealers in $\mathcal{I}$. This is a consequence of that fact that their AVSS protocol does satisfy a "completeness" property like ours.

These dealings are then combined using a super-invertible matrix W. To this end, the matrix W must be multiplied on the right by the column vector $\{\boldsymbol{\mathcal{F}}_i(0)\}_{i\in\mathcal{I}}$ to get $t+1$ ephemeral public keys needed to produce or verify a signature. To generate a signature, each party can easily generate signature share by a simple local computation and broadcast this to other parties. Parties can then use this data to compute signature shares, which are just linear combinations of their secret shares, and which are then "opened" to produce a signature. As a consequence of the fact that their AVSS protocol is incomplete, these signature cannot be "opened" using error-correcting codes, as in Section 7.1. Nevertheless, each party can collect $t + 1$ such shares and optimistically interpolate and test if the resulting signature share is valid. On the pessimistic path, where this fails, a party must compute a polynomial commitment as a linear combination of the polynomial commitments $\{\boldsymbol{\mathcal{F}}\}_{i\in\mathcal{I}}$, and use this to validate individual signature shares. We shall analyze only the optimistic path.

8.1 Communication complexity

To make an apples-to-apples comparison of the communication complexity, we estimate the amortized communication complexity per signature on the optimistic path, where communication complexity is defined as the sum, over all honest parties P and all parties Q, of the total number of bits that P sends to Q over a point-to-point channel. While SPRINT was designed and analyzed on top of a specific type of broadcast channel, no matter how this channel is implemented, broadcasting 1 bit to n parties will require 1 bit to be delivered to each on n parties, and so will contribute a term of n to the communication complexity as we have defined it, which ultimately corresponds to how communication bandwidth must be modeled as a finite resource in the real world.

In the offline phase, each party broadcasts $t + 1$ group elements, and sends n scalars over point-to-point channels, leading to a communication complexity of

$$\approx 2n^2(t + 1)\lambda + n^2\lambda$$

per dealer. This excludes many overheads which we assume can be ignored by appropriate additional batching.[5] To get the amortized cost per signature, we multiply by n (the number of dealers) and divide by $t + 1 \approx n/3$ (the size of the batch in the batch randomness extraction protocol). Thus, the amortized communication complexity per signature is $\approx (2n^2 + 3n)\lambda$. Their protocol is very symmetric, so in fact, each party transmits $\approx (2n + 3)\lambda$ bits per signature.

In the online phase, each party broadcasts a signature share, which means communication complexity per signature of $\approx n^2\lambda$. Again, the protocol is very symmetric, so each party transmits $\approx n\lambda$ bits per signature. For security reasons, signing requests are processed in batches of size $t+1$.

Thus, both the offline and online phases of SPRINT has a communication complexity per signature of $O(n^2\lambda)$ in both the offline and online phases. In contrast, our protocol has a communication complexity per signature of $O(n\lambda)$ in both the offline and online phases, assuming signing requests are also processed in batches of size $t+1$, as discussed in Section 7.1. Perhaps one could reduce the communication complexity of SPRINT's offline phase by using the "compact polynomial commitment" scheme of [KZG10], but that would restrict the use of the protocol to pairing-friendly elliptic curves E. Perhaps one could reduce the communication complexity of SPRINT's online phase by using the same batching technique we discussed in Section 7.1. However, it is not at all clear how to do this in an "optimistic" way without losing the ability to correctly identify a misbehaving party (this stems from the fact that their AVSS protocol is incomplete, which precludes the use of error correction).

For a more concrete comparison, recall that in our protocol, each party transmits 18 scalars and 9 group elements per signature in the offline phase. This translates to 36λ bits. Compare this to $(2n+3)\lambda$ bits for SPRINT. Thus, already for $n > 16$, our protocol will exhibit better communication complexity in the offline phase. As for the online phase, in our protocol, each party translates 6 scalars per signature, which translates to 6λ bits. Compare this to $n\lambda$ for SPRINT. Thus, already for $n > 6$, our protocol will exhibit better communication complexity in the online phase.

Example 8.1. As we did above in Example 5.1. Consider the case $n = 49 = 3\cdot 16+1$, considering the communication complexity for both the offline and online phase per signature, the communication complexity of SPRINT is $150/42 \approx 4.7$ times that of our protocol.

[5] As presented, the SPRINT protocol becomes insecure if additional batching is allowed. This is discussed in [Sho23], as well as other approaches that could be used instead to avoid this problem.

8.2 Computational complexity

Here, we focus on the running time of an individual party in the offline phase. We break the analysis up into two stages: Stage 1, which includes just the AVSS portion, and Stage 2, which includes just the batch randomness extraction. The offline phase of both the SPRINT protocol and ours can be decomposed into these two stages.

Stage 1: AVSS In this stage, we are just considering the AVSS protocol, but not the computation of the super-invertible-matrix/vector product. To state computational costs, in addition to the terms "full scalar/group multiplication", "tiny scalar/group multiplication", and "tiny scalar/scalar multiplication" introduced in Section 2.1, we introduce the following term:

Short scalar/group multiplication: This is a scalar/group multiplication, where the scalars are small (at most n) and public, and the group element is variable and public.
Typically, the best algorithm is this case will be a simple repeated doubling algorithm (possibly with NAF encoding of the scalar).

For SPRINT, each party as a dealer does one polynomial commitment computation and n local verifications of a share. One polynomial commitment computation costs $t+1$ full scalar/group multiplications. For a local verification of a share, a party P_j must evaluate a polynomial commitment $\boldsymbol{\mathcal{F}}$ at a point e_j, and must compute one full scalar/group multiplication. For the evaluation of a polynomial commitment, we may exploit the fact that the evaluation points e_i are small integers. In this setting, the best way to do this is to use Horner's rule, which takes t short scalar/group multiplications and t group additions. Thus, the total running time of each party during this stage is dominated by the cost of $(t+1)+n$ full scalar/group multiplications, and nt short scalar/group multiplications, and nt group additions. We divide these numbers by $t+1 \approx n/3$ to get the amortized cost per signature:

- 4 full scalar/group multiplications,
- n short scalar/group multiplications, and
- n group additions.

Compare these to the corresponding numbers for our protocol, stated in Section 2.1, which we repeat here for convenience. Let $\rho = \lceil 3\sigma/n \rceil + 2$, where σ is a statistical security parameter (typically $\sigma = 80$). This is the parameter used to define tiny scalar/group multiplications and tiny scalar/scalar multiplications.

- $1/(t+1)$ full scalar/group multiplications,
- 3 tiny scalar/group multiplications, and
- $4n$ tiny scalar/scalar multiplications.

Example 8.2. Let us compare using $n = 49 = 3 \cdot 16 + 1$ and $\lambda = 256$. To implement full scalar/group multiplications, we use Lim and Lee's algorithm as suggested in Section 2.1, with parameters $h = 5$ and $v = 2$. This results in a table of size $2^5 \cdot 2 = 64$, which is reasonable small (and so more amenable to constant-time lookup techniques). With these parameters, we can compute a full scalar/group multiplication $\lceil 256/5 \rceil + \lceil \lceil 256/5 \rceil / 2 \rceil - 2 = 76$ group additions.[6] For short

[6] This estimate is also roughly consistent with timing measurements based on `https://github.com/bitcoin-core/secp256k1`.

scalar/group multiplications, let $A(m)$ denote the length of the shortest addition-subtraction chain for m (which is the right metric when E is an elliptic curve, as subtraction is free). See `https://oeis.org/A128998` for a list of values of $A(m)$ for $m = 1, \ldots, 87$. In the range $m = 1, \ldots, 49$, the maximum value for $A(m)$ is 7, and so we use this as our estimate for the cost of a short scalar/group multiplication.

So the amortized cost per signature is

$$\approx 4 \cdot 76 + 49 \cdot 7 + 49 \approx 696$$

group additions per signature in this stage.

For our protocol, we set the security parameter $\sigma := 80$, and compute $\rho = \lceil 3 \cdot 80/49 \rceil + 2 = 7$. Let us first estimate the cost of a tiny scalar/group multiplication with this parameter ρ. Suppose we use the on-demand Straus algorithm as suggested in Section 2.1, and NAF encode the 7-bit scalar (assuming group negation is free as it is for an elliptic curve). The expected Hamming weight of such a NAF-encoded 7-bit scalar is ≈ 2.77 (obtained by direct computation). This leads to a cost per tiny scalar/group multiplications of amortized cost of 2.77. While this estimate is based on an average-case analysis, we are only performing these computations on very large batches of random scalars, and so the actual cost will be very close to the average-case cost with overwhelming probability.

So the amortized cost per signature for this stage of our protocol is

$$\approx (1/17) \cdot 76 + 3 \cdot 2.77 \approx 13$$

group additions and

$$\approx 4 \cdot 49 = 196$$

tiny scalar/scalar multiplications. As discussed in Example 5.1, we may conservatively estimate that each tiny scalar/scalar multiplication is about 1/40th the cost of a group addition. This means that the total cost of all of the tiny scalar/scalar multiplications is roughly equivalent to that of 5 group additions, which would give us an estimate of about $13 + 5 = 18$ group additions altogether.

Thus, we conclude that our new protocol in this stage is about $696/18 \approx 39$ times faster than SPRINT. Of course, this estimate must be taken with a grain of salt, as there are a number of built-in assumptions, and a number of algorithm overheads (both for SPRINT and our new protocol) which we have not fully taken into account. That said, we believe that to a first approximation, this is a fair comparison that indicates that our new protocol potentially provides a significant performance improvement. □

Stage 2: Batch randomness extraction In this stage, the only thing we are interested in is the computational cost of multiplying an $M \times N$ super-invertible matrix on the right by a column vector of group elements, where $N = t + 1$ and $N = 2t + 1$. Both the SPRINT protocol and our protocol behave identically in this stage.

While [BHK+23] does consider some constructions of super-invertible matrices, these constructions will not yield very efficient algorithms for moderately sized n (up to a few hundred). Prior to this work, for such moderately sized n, the best approach known was to use a Vandermonde matrix, where each column consists of powers of a small evaluation. We could, for example, use evaluation points $0, 1, \ldots, 2t$; however, in the setting where group negation is free, a better approach would be to use the evaluation points $0, \pm 1, \ldots, \pm t$. The cost then would be $2t^2$ short scalar/group

multiplications (with all scalars now in the range $1, \ldots, t$) and $t(2t-1) + 2t = t(2t+1)$ group additions.

With our new construction for a super-invertible matrix, that cost becomes $(t+1) \cdot 2t$ group additions.

Example 8.3. Let us continue with our example with $n = 49 = 3 \cdot 16 + 1$ and $\lambda = 256$. For the Vandermonde construction, we estimate the cost of computing $1 \cdot \mathcal{G}_1, 2 \cdot \mathcal{G}_2, \ldots, 16\mathcal{G}_{16}$ to be $\sum_{m=1}^{16} A(m) = 54$, where $A(m)$ is defined as in Example 8.2. Multiplying this by $2t$, the total cost of computing the $2t^2$ short scalar/group multiplications is $32 \cdot 54 = 1728$ additions. We perform an additional $16 \cdot (2 \cdot 16 + 1) = 528$ group additions for a total of $1728 + 528 = 2256$ group additions. We divide by 17 to get the amortized cost per signature, so about 133 group additions per signature.

With our new construction, the cost is $17 \cdot 32 = 544$ group additions. We divide by 17 to get the amortized cost per signature, so about 32 group additions per signature. So our new construction is $133/32 \approx 4$ times faster.

Putting together the amortized cost per signature in both stages of the offline phase, SPRINT with the Vandermonde matrix construction costs $696 + 133 \approx 829$ group additions, while our new protocol with the new matrix construction costs $18 + 32 \approx 50$ group operations.[7]

So considering the entirety of the offline phase, our new protocol with the new matrix construction is $829/50 \approx 17$ times faster than SPRINT with the old matrix construction. □

8.3 Packing secrets

One of SPRINT's innovations is to exploit "packed" secret sharing. This is an idea that goes back at least to [FY92]. The idea is that instead of having a polynomial $f \in \mathbb{Z}_q[x]$ encode a single secret as the value of f at the point 0, we instead have it encode several secrets, as the value of f at several points. Let p be a "packing" parameter. Recall that we are assuming evaluation points $1, \ldots, n$ for the secret shares. Then we can use the evaluation points $-(p-1), \ldots, -1, 0$ as the evaluation points for the packed secrets. Setting $p > 1$ can lead to better amortized performance, as we discuss. However, this comes at a cost: the protocol is no longer optimally resistant. Indeed, if t is a bound on the number of corrupt parties, instead of requiring $n > 3t$, the SPRINT protocol requires that $n > 3t + 2(p-1)$.

The only change to Stage 1 of the offline phase of SPRINT is that now, in the AVSS protocol, instead of sharing polynomials of degree less than $t+1$, we need to share polynomials of degree less than

$$d \coloneqq t + 1 + 2(p-1).$$

The requirement that $n > 3t + 2(p-1)$ is equivalent to $n \geq d + 2t$. For simplicity, let us assume that

$$n = d + 2t = 3t + 1 + 2(p-1)$$

going forward.

We calculate the following cost per signature for Stage 1:

- $(1 + n/d)/p$ full scalar/group multiplications,
- n/p short scalar/group multiplications, and

[7] This is a little bit higher than the cost reported in the Introduction, as we include here the cost of the tiny scalar/scalar multiplications, translated to an equivalent number of group operations.

– n/p group additions.

When Stage 1 completes, as before, we agree on a set $\mathcal{I} \subseteq \{1, \ldots, n\}$ of $n-t$ dealers with corresponding polynomial commitments $\{\boldsymbol{\mathcal{F}}_i\}_{i\in\mathcal{I}}$. Before proceeding to Stage 2, in need to additionally compute

$$\boldsymbol{\mathcal{F}}_i(j) \quad \text{for all } i \in \mathcal{I} \text{ and } j \in \{-(p-1), \ldots, -1, 0\}.$$

Let us call this Stage 1.5.

We note that in [BHK+23], the authors also suggest to use a nonstandard polynomial commitment scheme, which instead of being $f\mathcal{G}$ as we defined above, is the vector of group elements

$$f(-(p-1))\mathcal{G}, \ldots, f(-1)\mathcal{G}, f(0)\mathcal{G}, f(1)\mathcal{G}, \ldots, f(d-p)\mathcal{G}.$$

The point of this "optimization" is to ensure that the polynomial commitments themselves contain the group elements we need to compute in Stage 1.5, so that this step is free. However, this slows down Stage 1 significantly. Indeed, with this nonstandard polynomial commitment, for parties P_j with $j > d-p$, the cost of computing the share commitment $f(j)\mathcal{G}$ from this nonstandard polynomial commitment will be d scalar/group multiplications, where the scalars are now full-sized elements of $\mathbb{Z}_q$ (Lagrange interpolation coefficients). This is much greater than the cost of computing $d-1$ short scalar/group multiplications and additions using standard polynomial commitments. In particular, any savings by obtained by packing will almost surely be more than offset by this extra cost.

A better approach is to use a standard polynomial commitment, and Stage 1.5 now costs $(p-1)(n-t)(d-1)$ short scalar/group multiplications and group additions, where the scalars are now integers less than p in absolute value. To get the amortized cost per signature of doing this, we divide by $p(n-2t)$, to get an amortized cost of at most about $2(d-1)$ short scalar/group multiplications and additions per signature.

For Stage 2, we now have to compute p matrix-vector products, where that matrix is an $(n-2t) \times (n-t)$ super-invertible matrix, and the vectors are $\{\boldsymbol{\mathcal{F}}_i(j)\}_{i\in\mathcal{I}}$ for $j = -(p-1), \ldots, -1, 0$, as calculated in Stage 1.5. Let us be generous and assume that we use our new matrix construction for batch randomness extraction, which contributes an amortized cost of $n-t-1$ group additions per signature (for each packed secret, we have to do one matrix-vector multiply, so the amortized cost of this operation is independent of p).

Example 8.4. Let us continue our example with $n = 49 = 3 \cdot 16 + 1$ and $\lambda = 256$.

Let us set the packing parameter $p \coloneqq 7$. This means we can set $t \coloneqq 12$ so that $n = 3t+1+2(p-1)$ as required, and $d = 25$. So compared to before, with no packing, the protocol can withstand only 12 corruptions, rather than 16. In Stage 1, the amortized cost per signature is about

$$(3 \cdot 76 + 49 \cdot 7 + 49)/7 \approx 89$$

group additions.

For Stage 1.5, the amortized cost per signature is at most $2(d-1)$ short scalar/group multiplications and additions. With $A(m)$ defined as in Example 8.2, we calculate this amortized cost as

$$\frac{(n-t)(d-1)\sum_{m=1}^{p-1} A(m)}{p(n-2t)} = \frac{37 \cdot 24 \cdot 11}{7 \cdot 25} \approx 56$$

group operations.

For Stage 2, the amortized cost per signature is now

$$n - t - 1 = 36.$$

So the total cost per signature in the offline phase is about $89 + 56 + 36 = 181$ group additions.

Recall that we estimated the amortized cost per signature for our protocol in the offline phase at 50. So SPRINT with these parameters allows only 12 corruptions, while our allows 16, but SPRINT is still $181/50 \approx 3.6$ times slower than ours (even after we also "gifted" SPRINT a better polynomial commitment strategy and a better super-invertible matrix construction). □

The above discussion focused on the impact of packing on the computational complexity of SPRINT in the offline phase. As for communication complexity, it reduces the amortized communication complexity in both the offline phase and online phases by a factor of p.

Example 8.5. Let us continue our example with $n = 49 = 3 \cdot 16 + 1$ and $\lambda = 256$, but now with $p = 7$. We estimated that without packing SPRINT's communication complexity was about 4.7 times that our ours. With packing, it is therefore $\approx 4.7/7 \approx 0.67$ times that of ours.

Remark 8.1. While we do not think it will be very useful in the context of threshold Schnorr, our GoAVSS protocol also generalizes to support packed secret sharing (see Section 5.3).

8.4 Beyond SPRINT

As designed, SPRINT's threshold Schnorr protocol does not use a batch GoAVSS protocol like ours (where each dealer shares many polynomials at a time). In fact, it cannot. The reason is that, as pointed out in [Sho23], SPRINT's strategy for mitigating presignature attacks does not allow for this type of batch production of presignatures — there is a subexponential attack if this is done. However, as also pointed out in [Sho23], there are better mitigation strategies that could be used instead (see also Section 7.2 above). Given that fact, we might consider improving the performance of SPRINT's GoAVSS protocol by making it batched.

To that end, in this section, we focus on the general problem of optimizing a "batched Feldman" GoAVSS protocol based on polynomial commitments. We focus exclusively on the computational cost for the to generate and validate dealings, and do not concern ourselves with issues of communication,[8] or the exact procedure used to ensure that sufficiently many receivers have validated a dealing (one could use techniques such as [BHK+23] or also [GS22]). However, we will assume that $n - 2t$ honest parties must successfully validate a dealing for it to be accepted. The basic ideas we present here were already in sketched in Section 8.9 of [GS22], but we flesh them out here a bit more, and add some extra ideas.

Assume a dealer P_i is going to share a batch of polynomials $f_{i,1}, \ldots, f_{i,L} \in \mathbb{Z}_q[x]$, each of degree at most t. As pointed out in [GS22], one standard trick that we can apply is that of "vector" polynomial commitments. Suppose we have group elements $\mathcal{G}_1, \ldots, \mathcal{G}_L$ (and assume that finding nontrivial linear relations between them is computationally as hard as computing discrete logarithms in E). Then the vector polynomial commitment for these polynomials is just

$$\boldsymbol{\mathcal{F}}_i = \sum_{\ell} f_{i,\ell} \mathcal{G}_\ell \in E[x].$$

[8] That said, the use of vector polynomial commitments, discussed below, can substantially reduce the communication complexity of the protocol.

A party P_j that is given corresponding putative shares $v_{i,j,1}, \ldots, v_{i,j,L}$ from party P_i, along with a vector commitment $\boldsymbol{\mathcal{F}}_i$, can validate these shares by testing if

$$\sum_{\ell} v_{i,j,\ell} \mathcal{G}_\ell = \boldsymbol{\mathcal{F}}_i(j). \tag{6}$$

However, in order for this protocol to be useful to us (to actually be a GoAVSS protocol), we have to augment the protocol so that the dealer P_i also publishes group elements

$$\mathcal{S}_{i,1} \coloneqq f_{i,1}(0)\mathcal{G}, \ldots, \mathcal{S}_{i,L} \coloneqq f_{i,L}(0)\mathcal{G},$$

along with a non-interactive zero-knowledge proof that these group elements are consistent with $\boldsymbol{\mathcal{F}}_i$. That is, if $\mathcal{C}_i \coloneqq \boldsymbol{\mathcal{F}}_i(0)$, the dealer must prove knowledge of $s_{i,1}, \ldots, s_{i,L} \in \mathbb{Z}_q$ such that $\mathcal{C}_i = \sum_\ell s_{i,\ell}\mathcal{G}_\ell$ and

$$\mathcal{S}_{i,1} = s_{i,1}\mathcal{G}, \ldots, \mathcal{S}_{i,L} = s_{i,L}\mathcal{G}.$$

A standard Sigma protocol for this runs as follows:

- Prover chooses $s'_{i,\ell} \in \mathbb{Z}_q$ at random for $\ell \in [L]$, and sends

$$\mathcal{C}'_i \coloneqq \sum_\ell s'_{i,\ell}\mathcal{G}_\ell \quad \text{and} \quad \left\{ \mathcal{S}'_{i,\ell} \coloneqq s'_{i,\ell}\mathcal{G} \right\}_\ell$$

  to the verifier.
- The verifier chooses $e \in \mathbb{Z}_q$ at random and sends e to the prover.
- The prover responds to the verifier with

$$\left\{ z_{i,\ell} \coloneqq s'_{i,\ell} + e \cdot s_{i,\ell} \right\}_\ell.$$

- The verifier then checks that

$$\sum_\ell z_{i,\ell}\mathcal{G}_\ell = \mathcal{C}'_i + e \cdot \mathcal{C}_i \tag{7}$$

  and

$$z_{i,\ell}\mathcal{G} = \mathcal{S}'_{i,\ell} + e \cdot \mathcal{S}_{i,\ell} \quad (\text{for } \ell \in [L]). \tag{8}$$

Of course, we turn this into a non-interactive zero-knowledge proof using the Fiat-Shamir heuristic.

The verifier can combine the checks (6) and (7) into a single check by just checking a random linear combination of the two. So the cost now of this check is essentially one multi-scalar/group multiplication of length L, with full-sized private scalars.

Party P_j can easily reduce its cost of checking (8) by choosing $r_1, \ldots, r_L \in \mathbb{Z}_q$ at random, and then checking if

$$\big(\sum_\ell r_\ell z_{i,\ell}\big)\mathcal{G} = \sum_\ell r_\ell \mathcal{S}'_{i,\ell} + e \cdot \big(\sum_\ell r_\ell \mathcal{S}_{i,\ell}\big). \tag{9}$$

Moreover, the r_ℓ's may be small (and do not need to remain private). In fact, if we choose the r_ℓ's to be of length ρ, then we obtain the same security bound as in (3) in Theorem 5.1. This is because we are assuming that $n-2t$ honest parties must successfully validate a dealing for it to be accepted, so the same "distributed verification" argument made in Theorem 5.1 applies here as well.

With this, we can calculate the amortized cost per signature (or "processed sharing") of this "batched Feldman" protocol as follows:

F1) $2/(t+1)$ full scalar/group multiplications (as a dealer, once for each $\mathcal{S}_\ell$ and $\mathcal{S}'_\ell$),
F2) 6 tiny scalar/group multiplications (as a receiver, for the check (9)), and
F3) 4 full* scalar/group multiplications (1 as a dealer to compute the vector polynomial commitment, and 3 as a receiver to check the combination of (6) and (7)).

Here, we define a *full* scalar/group multiplication* to be one where the scalars are full sized and private, but the resulting products are used as terms in a very long summation (with fixed and public group elements).

We can compare these numbers to our new GoAVSS protocol:

N1) $1/(t+1)$ full scalar/group multiplications,
N2) 3 tiny scalar/group multiplications, and
N3) $4n$ tiny scalar/scalar multiplications.

We see that the optimized batched Feldman protocol is twice as slow as our new GoAVSS protocol on the first two counts. How it ultimately compares depends on the relative cost of (F3) vs (N3).

Example 8.6. Let us compare using $n = 49 = 3 \cdot 16 + 1$ and $\lambda = 256$. For large L, the Bos-Coster algorithm is a very good algorithm for multi-scalar/group multiplication. See discussion and references in [BDL+11], as well as empirical evidence of the practicality of the method. One nice feature of Bos-Coster is that it is easy to experimentally determine the average group addition count for a given set of parameters. Unfortunately, Bos-Coster is not constant time (as required here since the scalars are private), and has certain overheads that must be accounted for as well. Nevertheless, we can us the addition count for Bos-Coster as a reasonable estimate. If Bos-Coster is used to process a batch of 1K products at a time, the amortized cost per product is about 30 group additions. If we increase the batch size to 8K, the cost is about 20 group additions. Let us be generous and assume the right answer is about 20.[9]

This leads to an estimate for (F3) above of $4 \cdot 20 = 80$ group additions. Compare to the cost of (N3) above, which we estimated in Example 8.2 to be equivalent to about 5 group additions.

Putting everything together, we find that this optimized batch Feldman is still

$$\approx (13 \cdot 2 + 80)/(13 + 5) \approx 5.9$$

slower than our new GoAVSS protocol. This is assuming a very highly-optimized, constant-time multi-scalar/group multiplication algorithm.

We note that it may be possible to reduce the cost (F3) above even further, if each party P_i optimistically batches the corresponding checks from different dealers P_i. With luck, P_j can wait to collect data from all n dealers and perform all of its checks on those dealers in one batch. This would be difficult to implement easily in an asynchronous environment where some dealers may indeed be temporarily offline, if not fully malicious. But even if it did, the cost of (F3) would still be 1 full* scalar/group multiplication (arising from the fact that P_j, itself acting as dealer, has to generate one vector polynomial commitment), and this optimized optimistic batch Feldman is still

$$\approx (13 \cdot 2 + 20)/(13 + 5) \approx 2.6$$

[9] This estimate is also roughly consistent with timing measurements based on `https://github.com/bitcoin-core/secp256k1`.

times slower than our new GoAVSS protocol.

But this is all really beside the point. All of this effort spent on optimizing the Feldman approach is aimed at reducing the cost of things that we can just simply avoid doing to begin with. □

Remark 8.2. In the last example, we mentioned the use of the Bos-Coster algorithm. We could, in fact, use Bos-Coster to implement the tiny scalar/group multiplications, rather than on-demand Straus. If we use Bos-Coster with a relatively small batch size of 128, this would reduce the number of group additions per product from 2.77 (using on-demand Straus) to 1.62, and would shave off $\approx 3 \cdot (2.77 - 1.62) = 3.45$ from the amortized group addition count per signature for our protocol. The reason we did not do this is that the on-demand Straus algorithm is extremely simple to program and adds essentially no overhead, while Bos-Coster requires a priority queue. That said, because the numbers in the priority queue are very small integers (less than $2^7 = 128$), and because in Bos-Coster the size of the largest value never increases, it is possible to implement the priority in an extremely simple fashion as a "bucket queue", with an amortized cost per queue operation that is a very small constant (and with these parameters, everything should easily fit in the L1 cache). So perhaps this approach would actually give a small performance improvement to our protocol, and would not be too challenging to implement either.

Acknowledgements

The first author thanks Melissa Chase for discussions about cryptographic applications of the symmetric Pascal matrix.

References

BDL+11. D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang. High-speed high-security signatures. Cryptology ePrint Archive, Paper 2011/368, 2011. `https://eprint.iacr.org/2011/368`.

Ber02. D. J. Bernstein. Pippenger's exponentiation algorithm, 2002. Manuscript, `https://cr.yp.to/papers/pippenger.pdf`.

BHK+23. F. Benhamouda, S. Halevi, H. Krawczyk, Y. Ma, and T. Rabin. SPRINT: High-throughput robust distributed schnorr signatures. Cryptology ePrint Archive, Paper 2023/427, 2023. URL `https://eprint.iacr.org/2023/427`. `https://eprint.iacr.org/2023/427`.

Bra87. G. Bracha. Asynchronous byzantine agreement protocols. *Inf. Comput.*, 75(2):130–143, 1987.

CP17. A. Choudhury and A. Patra. An efficient framework for unconditionally secure multiparty computation. *IEEE Trans. Inf. Theory*, 63(1):428–468, 2017.

CT05. C. Cachin and S. Tessaro. Asynchronous verifiable information dispersal. In P. Fraigniaud, editor, *Distributed Computing, 19th International Conference, DISC 2005, Cracow, Poland, September 26-29, 2005, Proceedings*, volume 3724 of *Lecture Notes in Computer Science*, pages 503–504. Springer, 2005.

DGK18a. N. Drucker, S. Gueron, and V. Krasnov. The comeback of Reed Solomon codes. In *2018 IEEE 25th Symposium on Computer Arithmetic (ARITH)*, pages 125–129, 2018.

DGK18b. N. Drucker, S. Gueron, and V. Krasnov. Making AES great again: the forthcoming vectorized aes instruction. Cryptology ePrint Archive, Paper 2018/392, 2018. `https://eprint.iacr.org/2018/392`.

Fel87. P. Feldman. A practical scheme for non-interactive verifiable secret sharing. In *28th Annual Symposium on Foundations of Computer Science, Los Angeles, California, USA, 27-29 October 1987*, pages 427–437. IEEE Computer Society, 1987.

FLdO18. A. Faz-Hernández, J. C. López-Hernández, and A. K. D. S. de Oliveira. SoK: A performance evaluation of cryptographic instruction sets on modern architectures. In K. Emura, J. H. Seo, and Y. Watanabe, editors, *Proceedings of the 5th ACM on ASIA Public-Key Cryptography Workshop, APKC@AsiaCCS, Incheon, Republic of Korea, June 4, 2018*, pages 9–18. ACM, 2018.

FY92. M. K. Franklin and M. Yung. Communication complexity of secure computation (extended abstract). In S. R. Kosaraju, M. Fellows, A. Wigderson, and J. A. Ellis, editors, *Proceedings of the 24th Annual ACM Symposium on Theory of Computing, May 4-6, 1992, Victoria, British Columbia, Canada*, pages 699–710. ACM, 1992.

GJKR07. R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Secure distributed key generation for discrete-log based cryptosystems. *J. Cryptol.*, 20(1):51–83, 2007.

Gro21. J. Groth. Non-interactive distributed key generation and key resharing. Cryptology ePrint Archive, Report 2021/339, 2021. `https://ia.cr/2021/339`.

GS22. J. Groth and V. Shoup. Design and analysis of a distributed ECDSA signing service. Cryptology ePrint Archive, Report 2022/506, 2022. `https://ia.cr/2022/506`.

HN06. M. Hirt and J. B. Nielsen. Robust multiparty computation with linear communication complexity. In C. Dwork, editor, *Advances in Cryptology - CRYPTO 2006, 26th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 2006, Proceedings*, volume 4117 of *Lecture Notes in Computer Science*, pages 463–482. Springer, 2006.

KZG10. A. Kate, G. M. Zaverucha, and I. Goldberg. Constant-size commitments to polynomials and their applications. In M. Abe, editor, *Advances in Cryptology - ASIACRYPT 2010 - 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings*, volume 6477 of *Lecture Notes in Computer Science*, pages 177–194. Springer, 2010.

LL94. C. H. Lim and P. J. Lee. More flexible exponentiation with precomputation. In Y. Desmedt, editor, *Advances in Cryptology - CRYPTO '94, 14th Annual International Cryptology Conference, Santa Barbara, California, USA, August 21-25, 1994, Proceedings*, volume 839 of *Lecture Notes in Computer Science*, pages 95–107. Springer, 1994.

Ped91. T. P. Pedersen. A threshold cryptosystem without a trusted party (extended abstract). In D. W. Davies, editor, *Advances in Cryptology - EUROCRYPT '91, Workshop on the Theory and Application of of Cryptographic Techniques, Brighton, UK, April 8-11, 1991, Proceedings*, volume 547 of *Lecture Notes in Computer Science*, pages 522–526. Springer, 1991.

Pip76. N. Pippenger. On the evaluation of powers and related problems (preliminary version). In *17th Annual Symposium on Foundations of Computer Science, Houston, Texas, USA, 25-27 October 1976*, pages 258–263. IEEE Computer Society, 1976.

Sho23. V. Shoup. The many faces of Schnorr. Cryptology ePrint Archive, Paper 2023/1019, 2023. `https://eprint.iacr.org/2023/1019`.

SS23. V. Shoup and N. P. Smart. Lightweight asynchronous verifiable secret sharing with optimal resilience. Cryptology ePrint Archive, Paper 2023/536, 2023. `https://eprint.iacr.org/2023/536`.