Proximity Signatures

Guillermo Angeris
gangeris@baincapital.com

Kobi Gurkan
kgurkan@baincapital.com

April 2026

Abstract

In this note we introduce the concept of *proximity signatures*, where verifiers who can only access a small part of some data would like a guarantee that (a) this data is "close" to a uniquely decodable message (so the message can be decoded from the data via error decoding) and (b) the uniquely decodable message is signed by an associated secret key. This is useful in situations where the message is very large but the verifiers are small devices who only need the guarantee that the message was signed by some specific secret key or set of keys. As a motivating example, we consider the data availability problem. There, users submit large signed pieces of data that together form a larger data matrix. The signatures and integrity of this data must then be checked by nodes who can only download a small proportion of this matrix. We present a construction inspired by linear subspace signatures.

1 Introduction

An important part of blockchain security is to ensure that the data approved by validators to be included in the chain's history is publicly available for download to any clients who'd like to use it — light clients, full nodes, and validators alike. The problem of ensuring that this data is indeed available for download is called the *data availability problem*, first defined in [Al-19]. Solving this problem has become an important part of the scaling roadmaps for many blockchains, including Ethereum [Eth26], Solana [Sol26], Celestia [Cel26], Commonware [Com26] and others, since much of the cost associated with verifying a given blockchain comes from the cost of downloading the data.

Data availability. Works such as [Al-19, EMA24, Fei22, HSW23] show how to efficiently ensure data availability without requiring each participant download a complete block, under a variety of trust assumptions. Many of these protocols do not address an important practical point of block verification in the data availability setting: how to ensure that the data included in a given block was paid for, and, conversely, that the data which was paid for was indeed included in the block. We call this particular guarantee the "payment for inclusion" guarantee. In general, if payments for inclusion is mentioned at all, it is generally

in the context of verifying arbitrary computation over a given block [Al-19, EA25], which we argue next can be very expensive in practice.

Payment for inclusion. In the context of blockchains, a *payment* is, roughly speaking, a short message of some pre-specified form that is signed by some associated secret key. Similarly, a *data blob* is a (generally large) piece of abstract data also signed by some secret key. A *block* is a collection of payments and data blobs. The *payment for inclusion* problem seeks to verify that in a given block, every data blob, signed by some secret key, has an associated payment signed by the same secret key, and vice versa. In what follows, it is worth keeping in mind the following numbers, which are approximate near-term targets for many chains implementing data availability protocols: payments are on the order of 128 bytes, blobs are on the order of 1 MiB, and blocks are on the order of 128 MiB. (This means that we might expect roughly 128 blobs per block or so.) We assume light nodes, who seek to verify payment for inclusion, have no more than 1 MiB/s of bandwidth, which roughly corresponds to a 3G internet connection in many parts of the world. Finally, we assume that a block is produced every 10 or so seconds.

Signature verification. In many cases, the signature verification of blobs and payments has the following form [Ame99, BLS01, Sch91]. First, most standard signature verification algorithms assume that the message (or blob) is known to the verifier, along with a public key and purported signature of the message. Then, the verifier must hash the complete message (or blob) to some small string (or element of a given group). Finally, some operations are performed on this small string (or group element) along with the signature to verify that the signature is valid. That is, that the signature indeed must have come from the known secret key and matches the message (or blob) that the verifier has.

Signatures in the data availability setting. There are two important difficulties associated with this signature verification in the data availability setting. First, only a small number of blobs can be downloaded by a light node before the next block is produced, so signatures cannot be directly verified by light nodes. One possible approach to solving this problem is to construct a succinct proof of the following two facts: (1) the blob was included in the block and (2) the signature of the blob indeed corresponds to the provided secret key. While (1) is reasonably fast to prove (see, *e.g.*, [EA25]), (2) requires constructing a succinct proof that the blob's signature verifies. Unfortunately, as mentioned in the previous paragraph, verifying the signature requires hashing each blob, which, in turn is extremely expensive in practice, with the best-known overhead being around ten thousand or so times the computational cost of doing the hashing itself for SHA256 [Nat15]. This overhead means that proving the validity of message signatures is much more expensive than other protocol computations, so powerful hardware is often needed to generate these proofs within the block production time limits.

Encoded messages. In many data availability schemes, each blob is encoded using some linear error correcting code before being placed in a block. This enables light nodes to

download only small parts of the block and allows them to collectively reconstruct the original block if necessary. Since these encodings are so structured, they also allow a number of other applications to exist. For example, it is possible for light nodes to use these encodings to verify that the received parts of a block are indeed correctly computed [EMA24] and, in some cases, verify any arbitrary computation over the given block [EA25]. A natural question is: can we similarly exploit this structure to allow light nodes to verify payment for inclusion in a given block, while only having to download a very small amount of data?

Proximity signatures. To this end, we introduce the notion of "proximity signatures" which we define more carefully below. Roughly speaking, a proximity signature guarantees that some provided encoded message, such as a blob, (1) decodes to a unique message, and, (2) that this message is signed by the provided secret key and could only have been signed by this key. The main idea is to use the fact that these messages are encoded using some error correcting code, which allows a verifier to only sample a very small number of entries of this message, to be sure that this encoding can indeed be uniquely decoded to a specific, signed message.

We similarly define "batched proximity signatures" which are more directly applicable to the data availability setting. Here, a number of encoded messages (each corresponding to a blob, say) are each signed by different secret keys, and we would like to verify that these encodings are each uniquely decodable and signed by the corresponding secret keys, while only reading a very small amount of the data.

Data availability sampling and proximity signatures. One particularly interesting and important interplay between proximity signatures and data availability sampling is that work done for data availability sampling (that is, light nodes downloading some parts of the block) can be reused for the verification of proximity signatures. This cuts down on the light node communication costs substantially, enabling much smaller nodes to be able to verify payment for inclusion without requiring the large proving costs of current approaches. We will show how this interplay can be used in practice to enable practical schemes that allow light nodes to verify the inclusion of signed data in a block.

1.1 Technical overview

Our design focuses on integrating signatures with [EMA24] and [EA25]. Specifically, we model the chain's blocks as a matrix over a finite field. In this matrix, as in the introduction, each column corresponds to a data blob, and we assume that each data blob is uniquely produced by a single provider. (That is, no signer signs two different data blobs, or columns, in one block.)

Main construction. The main construction we present adapts [Bon+09] to the case when a random linear combination of the vectors is the only thing the verifier receives. This construction, while incurring a high computational cost for the verifier, is appealing, since it uses elements that are naturally produced in ZODA, and is amenable to further compression

of verifier work by other techniques. This approach does not require additional work by full nodes beyond the data overhead that is required to store and share the signatures.

Other results. Of independent interest, as part of our main construction, we introduce *many-slot signatures*, where the signature scheme is unforgeable when a signer presents only one signature in an agreed upon slot. The starting point for our construction is the observation that the signatures of the form $\sigma = \alpha m g_1$, where α is the private key, are insecure since you can transform σ to a signature on any other message m' by computing $\sigma' = m' m^{-1} \sigma$. We observe that if you add a fixed component h to the signature, as in $\sigma = \alpha(g_0 + m g_1)$, you obtain a signature scheme where a signature can't be transformed into another, unless there's already one signature for a set of base points g_0 and g_1. Note that these can be extended to n messages per slot by including n powers of m, as follows: $\sigma = \alpha(g_0 + m g_1 + ... + m^{n-1} g_{n-1})$. Intuitively, this works since each signature is an evaluation of a polynomial in random coefficients over the scalar field of the curve, which can only be recovered if you have $n + 1$ evaluations. OOPS [GJN25] generalizes these signatures for multiple messages per slot, for the single-signer scalar message case.

Additional constructions. We present two additional constructions. One, presented in section 3, is a basic single-signer version built from a proof of proximity together with any standard signature scheme, and the other is a design based on BLS signatures, presented in appendix A, where encoded elements are signed individually. A full node, when sampled, aggregates these signatures over the sampled positions. A similar construction can utilize any signature scheme, without the bandwidth reduction that is achieved by aggregation.

Related work. The closest piece of related work is the definition of beholder signatures in [Dzi+25], released during the preparation of this note. A beholder signature guarantees that signers, when signing a commitment to a message, know the complete message. In proximity signatures, on the other hand, we have a different guarantee: the data is available to be reconstructed in a network of nodes and this data, once reconstructed, decodes to what the original signed data must (uniquely) decode to.

This note. In what follows, we will define proximity signatures, many-slot proximity signatures, and batched proximity signatures more carefully, present the main construction inspired by linear subspace signatures, and briefly sketch the other two constructions. To keep this note reasonably short, we will not spell out all of the associated security games in full formal detail, though any reader familiar with standard signature security definitions and error correcting codes should be able to fill in the gaps.

2 Definitions

We revisit the definition of existential unforgeability from [BSW06], and extend it to the case of proximity signatures.

2.1 Signature systems

A *signature system* is a collection of three algorithms: `KeyGen`, `Sign`, and `Verify`. These work as follows.

- `KeyGen` takes no input and outputs a public key `PK` and a secret key κ
- `Sign` takes a secret key κ and a message $\tilde{m}$ and outputs a signature σ
- `Verify` takes a public key `PK`, a message $\tilde{m}$, and a signature σ and outputs a Boolean indicating whether σ is a valid signature of $\tilde{m}$

We will briefly review the definition of the existential unforgeability of signature systems, which uses the three algorithms below.

2.1.1 Existential unforgeability

We define *existential unforgeability* under an adaptive chosen-message attack via the following game, which proceeds in three phases: set up, queries, and output. The game is between two parties, the *challenger* and the *adversary*. We will say that the signature system is *secure* (*i.e.*, it satisfies existential unforgeability) if no reasonable computationally-bounded adversary can win this game with non-negligible probability. This is a standard definition; see, *e.g.*, [BSW06] for more.

Setup. In the setup phase, the challenger runs `KeyGen`. The challenger then sends the adversary the resulting public key `PK` and keeps the secret key κ to itself.

Queries. The adversary issues signature queries $\tilde{m}_1, ..., \tilde{m}_q$. To each query $\tilde{m}_i$ the challenger responds by running `Sign` to generate a signature σ_i of $\tilde{m}_i$ and sending σ_i to the adversary. These queries may be asked adaptively so that each query $\tilde{m}_i$ may depend on the replies to $\tilde{m}_1, ..., \tilde{m}_{i-1}$.

Output. Finally, the adversary outputs a pair $(\tilde{m}_{q+1}, \sigma_{q+1})$. The adversary wins if σ_{q+1} is a valid signature of $\tilde{m}_{q+1}$ according to `Verify` and $\tilde{m}_{q+1}$ is not one of the messages $\tilde{m}_1, ..., \tilde{m}_q$ generated during the query phase.

2.1.2 Discussion

The intuition behind this game and definition of security is that an adversary who is given access to signatures of messages of its choice should not be able to forge a signature of any new message. This captures the idea that only someone who knows the secret key should be able to sign new messages, even if they can query and observe signatures of other messages.

2.2 Proximity signature systems

A *proximity signature system* takes the same form as a signature system along with an additional encoding step. As part of the associated system, we assume that all parties have agreed upon an error correcting code (for example, a Reed–Solomon code) that is used to encode messages before signing.

- `KeyGen` takes no input and outputs a public key `PK` and a secret key κ
- `Sign` takes a secret key κ and a message $\tilde{m}$ and outputs a signature σ and an encoding of the message m
- `Verify` takes a public key `PK`, queries parts of the encoding m and parts of the signature σ, and outputs a boolean whether the encoding indeed corresponds to an encoding of some unique message $\tilde{m}$ and, in turn, whether σ is a valid signature of $\tilde{m}$

2.2.1 Proximate existential unforgeability

We define proximate existential unforgeability under an adaptive chosen-message attack via the following game.

Setup. The challenger runs `KeyGen`. It gives the adversary the resulting public key `PK` and keeps the secret key κ to itself.

Signature Queries. The adversary requests that messages $\tilde{m}_1, ..., \tilde{m}_q$ be signed. To each query $\tilde{m}_i$, the challenger responds by running `Sign` to generate a signature σ_i and sends σ_i to the adversary. (The adversary can compute the encoding m_i directly or modify it as she wishes.) These queries may be adaptive so that each query $\tilde{m}_i$ may depend on the replies to $\tilde{m}_1, ..., \tilde{m}_{i-1}$.

Output. Finally the adversary outputs a pair (m_{q+1}, σ_{q+1}). The adversary wins if `Verify` accepts and one of the two following conditions is true: (a) m_{q+1} is not uniquely decodable or (b) m_{q+1} is uniquely decodable to some message $\tilde{m}_{q+1}$, not equal to any query $\tilde{m}_1, ..., \tilde{m}_q$, and σ_{q+1} is a valid signature of the encoded m_{q+1} according to `Verify`.

2.2.2 Generalization of signature schemes

Proximity signatures generalize signature schemes. A standard signature scheme corresponds exactly to the case where the code is the identity code (*i.e.*, the encoding is just the message, so $m = \tilde{m}$) and the verifier queries the entire codeword m, outputting the same result as the signature scheme verifier. We can also see that proximity signatures augment traditional signature schemes by allowing for the verification of signatures on parts of a purportedly encoded message, rather than requiring access to the complete original message. The guarantee a verifier gets is that there is a unique message corresponding to the encoding, that the encoding is close to the encoding of this unique message, and that the signature indeed corresponds to this unique message.

Guarantees. Note that this generalization does not prevent the adversary from doing certain silly attacks that are not possible in standard signature schemes. For example, the adversary could corrupt a small number of entries in an encoding of a message and, so long as the number of corruptions is small enough, the verifier would accept the encoding as valid. Of course, this is not a problem in our case, since the corrupted encoding would still decode

to the same unique message, and hence the signature would still be valid. But we note that this guarantee is slightly different from that of a standard signature scheme.

2.3 Many-slot proximity signature systems

We extend the notion of proximity signature systems to the case where the signature scheme we use accepts an additional *slot* parameter. We call this a *many-slot signature scheme* since the public key of the signer is the same across slots. On the other hand, if the signer signs any one slot twice, then the signature is malleable, and allows the adversary to forge a signature for any message. Note that this is not a strict generalization of a proximity signature system, since we add the restriction on the amount of signatures the adversary can query for the slot they wish to forge a signature for. In our case, it's at most one message per slot. OOPS [GJN25] addresses the case for multiple queries per slot for scalar messages.

The methods are adapted from above as follows.

- `KeyGen` takes no input and outputs a public key `PK` and a secret key κ
- `Sign` takes a secret key κ, a message $\tilde{m}$ and a slot s, and outputs a signature σ and an encoding of the message m
- `Verify` takes a public key `PK`, a slot s, queries parts of the encoding m and parts of the signature σ, and outputs a boolean whether the encoding indeed corresponds to an encoding of some unique message $\tilde{m}$ and, in turn, whether σ is a valid signature of $\tilde{m}$

2.3.1 Many-slot proximate existential unforgeability

In this security game, the adversary is required to forge a proof for any slot s^* of their choosing, and is allowed to query the signature oracle for this slot at most once. This models the realistic scenario where the adversary can observe a signature produced by another party and attempts to forge using this observed signature. We allow the adversary to request signatures on (message, slot) pairs for other slots (*i.e.*, slots not equal to s^*) freely, including more than once. The adversary then generates an encoding and succeeds if the proof verifies against the public key and the encoding.

More explicitly, we define many-slot proximate existential unforgeability under an adaptive chosen-message attack via the following game.

Setup. The challenger runs `KeyGen`. It gives the adversary the resulting public key `PK` and keeps the secret key κ to itself.

Signature Queries. The adversary issues signature queries $(\tilde{m}_1, s_1), ..., (\tilde{m}_q, s_q)$. Note that s_j does not need to be unique. To each query $\tilde{m}_j$ the challenger responds by running `Sign` to generate a signature σ_j of $\tilde{m}_j$ and sending σ_j to the adversary. These queries may be asked adaptively so that each query $\tilde{m}_j$ may depend on the replies to $\tilde{m}_1, ..., \tilde{m}_{j-1}$.

Output. Finally the adversary outputs σ^*, s^*, and m, where $s^* = s_j$ for at most one slot index j. The adversary wins if `Verify` accepts and one of the following conditions is true: (a)

m is not uniquely decodable to a message $\tilde{m}^*$ or (b) m is uniquely decodable to a message $\tilde{m}^*$, s^* was queried at most once, and σ^* is a valid signature for the encoded m and slot s^* with the public key `PK` according to `Verify`.

We say that the many-slot proximity signature system is secure if no reasonable adversary (under certain security assumptions) can win this game with non-negligible probability.

2.3.2 Discussion

In our security definition the adversary can query more than one signature for slots other than the challenge slot s^*. This allows the adversary to forge signatures for previous slots. Even with that, the adversary should not be able to forge a signature for the challenge slot, which has at most one observed signature associated with it.

2.3.3 Batching

We can also define the *batched* many-slot proximity signature system and corresponding security game, in which multiple signers sign messages for the same slot, and the signatures and encodings can be aggregated by a batching algorithm. This is the version we will use in our practical construction—the security proof and game is a direct adaptation of the one above to the multiple signer setting, where there is an additional batching algorithm.

3 Constructions

In this section, we will first discuss a single-signer proximity signature construction, which is what one might call a "black box" construction, in that it uses any standard signature scheme as a subroutine. We then present a second construction for many-slot signatures, inspired by [Bon+09], which is strengthened to a final batched many-slot proximity signature construction that is suitable to the data availability setting.

Basic notation. Throughout the section, we use the following notation. Let $\mathbf{G}_1$ and $\mathbf{G}_2$ be finite Abelian groups of prime order p and let $\mathbf{F}$ be a finite field also of order p. Since both groups are Abelian, we use additive notation; *i.e.*, for $x, y \in \mathbf{G}_1$ then $x + y$ is the group operation and 0 is the identity element. This lets us write, for $\alpha \in \mathbf{F}$ and $x \in \mathbf{G}_1$, the scalar product $\alpha x \in \mathbf{G}_1$ (and similarly for $\mathbf{G}_2$). Additionally, for convenience, we set $g_1 \in \mathbf{G}_1$ and $g_2 \in \mathbf{G}_2$ to be fixed, publicly known generators of the groups. Finally, for elements $x \in \mathbf{G}_1$ and $z \in \mathbf{G}_2$, we define a *pairing* $\langle x, z \rangle \in \mathbf{G}_T$, where $\mathbf{G}_T$ is another finite Abelian group of prime order p. Though important for implementation, for the purposes of this note, we will use no other mathematical facts about the pairing except for the fact that it is bilinear and nondegenerate. That is, the pairing is *bilinear* if for any $\alpha, \beta \in \mathbf{F}$ we have that $\langle \alpha x, \beta z \rangle = \alpha\beta\langle x, z \rangle$ and we also have $\langle x + y, z \rangle = \langle x, z \rangle + \langle y, z \rangle$ for any $x, y \in \mathbf{G}_1$ and $z \in \mathbf{G}_2$, with a similar condition holding for the second argument. The pairing is *nondegenerate* if, for any $x \in \mathbf{G}_1$ and $z \in \mathbf{G}_2$ we have $\langle x, z \rangle = 0$ only when $x = 0$ or $z = 0$.

3.1 Proximity signature

For the basic proximity signature, there are a number of direct constructions. The simplest case is to use any "proof of proximity" (such as Ligero [Ame+17], FRI [Ben+18], WHIR [Arn+25], or Ligerito [NA25]) to prove that the encoding m is close to a unique message $\tilde{m}$, and then use any standard signature scheme to sign the proof. Another property we might want would be the ability to guarantee that, after querying a message and verifying proximity (along with verifying a signature), further queries of the message can be checked against some part of the original proof to guarantee their integrity, such as in [HSW23], FRIDA [HSW24], or ZODA [EMA24].

3.2 An efficient batched many-slot proximity signature

The construction presented next is inspired by [Bon+09]. We assume that there exists some code generator matrix $G \in \mathbf{F}^{\ell' \times \ell}$ for some linear error correcting code (usually a Reed–Solomon code) that encodes messages of length ℓ to codewords of length ℓ'. We will also assume the existence of a hash function $H : \mathbf{F} \times \mathbf{F} \to \mathbf{G}_1$, which we will use as a random oracle. (Roughly speaking, we will assume that H behaves like a uniformly random function, taken over the sets of functions $\mathbf{F} \times \mathbf{F} \to \mathbf{G}_1$ over each instantiation of the protocol.) Finally, it will be extremely convenient to allow 'ordered lists of elements' of the group $\mathbf{G}_1$ to inherit scalar operations over $\mathbf{F}$. That is, for a list of n elements $z \in \mathbf{G}_1^n$, we can define $\alpha z = (\alpha z_1, ..., \alpha z_n)$ for any $\alpha \in \mathbf{F}$, much like we can for vectors. Similar operations hold for addition, "inner products" with a vector over $\mathbf{F}$ and, finally, matrix-vector products where the matrix is over $\mathbf{F}$. (Indeed, mathematically-inclined readers will note that since $\mathbf{G}_1$ is isomorphic to $\mathbf{F}$ then $\mathbf{G}_1^n$ is isomorphic to $\mathbf{F}^n$ with a map that is linear over $\mathbf{F}$, so $\mathbf{G}_1^n$ is a vector space with identical structure to $\mathbf{F}^n$, and the operations above are the obvious ones.) As usual, we assume that the order of $\mathbf{G}_1$ and $\mathbf{G}_2$ (and therefore the size of $\mathbf{F}$) is large enough such that $1/|\mathbf{F}|$ is negligibly small.

Notation. Unless stated otherwise, ℓ is the message length, ℓ' is the codeword length, and n is the number of signers. We will use $i = 1, ..., n$ to index signers, $j = 1, ..., q$ as a slot index with s_j as the corresponding slot number, and $k = 1, ..., \ell$ to index elements inside a message.

Set up. In this set up, we have multiple signers and each signer $i = 1, ..., n$ has a secret key $\kappa_i \in \mathbf{F}$. Every signer produces a message over each slot $s_1, ..., s_q$. For each signer $i = 1, ..., n$ and slot index $j = 1, ..., q$, their message $\tilde{m}_{ij} \in \mathbf{F}^\ell$ will be a vector of length ℓ over $\mathbf{F}$. For convenience, we will define the "batched" message for slot index $j = 1, ..., q$ to be the matrix $\tilde{X}_j \in \mathbf{F}^{\ell \times n}$—that is, the n columns of the matrix $\tilde{X}_j$ correspond to the messages $\tilde{m}_{1j}, ..., \tilde{m}_{nj}$. We can then define the encoding $X_j \in \mathbf{F}^{\ell' \times n}$ of the batched message $\tilde{X}_j$ to be

$$X_j = G\tilde{X}_j,$$

for each slot index $j = 1, ..., q$. That is, the columns of X_j are the encoded messages $G\tilde{m}_{1j}, ..., G\tilde{m}_{nj}$. Finally, we will define a *batching algorithm* that will batch the signatures for

all signers, for each slot j, along with encoding the batched message X_j and provide a proof that the encoding X_j indeed corresponds to a unique $\tilde{X}_j$, and, therefore, that each message $\tilde{m}_{ij}$ is uniquely decodable from X_j.

KeyGen. Each signer $i = 1, ..., n$ generates a uniformly random $\kappa_i \in \mathbf{F}$ and constructs the public key:

$$\mathsf{PK}_i = \kappa_i^{-1} g_2,$$

where the inverse is over the field $\mathbf{F}$.

Sign. Each signer $i = 1, ..., n$ produces a signature for each message in each slot index $j = 1, ..., q$, where the message is indexed by the pair (i, j). The signature Σ_{ij} corresponding to signer i for slot index j (and therefore to message $\tilde{m}_{ij}$) is defined as

$$\Sigma_{ij} = \kappa_i \left(H(0, s_j) + \sum_{k=1}^{\ell} \left(\tilde{m}_{ij}\right)_k H(k, s_j) \right).$$

Note that Σ_{ij} is an element of $\mathbf{G}_1$.

Batching algorithm. The proof of proximity and batched signatures are produced by the following batching algorithm. The proof of proximity is the same as that of the ZODA Hadamard variation, found in appendix E of [EMA24]. To construct this proof, the batching algorithm receives all messages $\tilde{X}_j$ for slot index j, and computes the encoding

$$X_j = G\tilde{X}_j,$$

for slot index j. It then sends to the verifier the encoding X_j (or, in practice, an oracle to the rows of the encoding X_j) and receives a random vector $r_j \in \mathbf{F}^n$. (In practice, this randomness comes from Fiat–Shamir, but it can also be provided by the verifier in an interactive version.) The batching algorithm then computes the random linear combination

$$y_j = \tilde{X}_j r_j.$$

Finally, it forwards y_j to the verifier, along with the signatures $\{\Sigma_{ij}\}$. Note that the batching algorithm does not need to know the private keys of the signers with the signers to produce the batched signature. It also does not require any further interaction with the signers after receiving the messages and signatures. Both properties are important in practice. Additionally, note that the total additional proof size for slot j (that is, in addition to theoriginal ZODA Hadamard proof) comes from the n signatures. This additional size is linear in the number of signers and independent of the message length.

Verify. For each slot $j = 1, ..., q$, the verify algorithm will have already chosen a random vector $r_j \in \mathbf{F}^n$ and will receive the encoding X_j, proof of proximity y_j, and individual message signatures Σ_{ij} for each signer $i = 1, ..., n$. The algorithm then uniformly randomly samples

a set $S \subseteq \{1, ..., \ell'\}$ of some fixed size $|S|$ (chosen based on security parameters) and then verifies that

$$\big(X_j\big)_S r_j = G_S y_j, \tag{1}$$

where G_S is the submatrix of G consisting of the rows indexed by S and similarly for $(X_j)_S$. (Note that this means that the verifier only needs access to $|S|$ rows of the matrix X_j.) The verifier then checks that for slot j, the following condition holds:

$$\sum_{i=1}^{n} \big(r_j\big)_i \langle \Sigma_{ij}, \mathsf{PK}_i \rangle = \left\langle \big(r_j^T\, \mathbf{1}\big) H\big(0, s_j\big) + \sum_{k=1}^{\ell} \big(y_j\big)_k H\big(k, s_j\big),\, g_2 \right\rangle. \tag{2}$$

If both checks pass, the verifier accepts.

3.2.1 Explanation of the construction

At a high level, the construction is relatively simple. Each signer signs their messages using a BLS-like signature scheme, with some additional randomness provided by the hash function H, much in the same way as [Bon+09]. The signatures are then batched together using the randomness r_j. The main idea is that it is possible to reuse the random linear combination y_j produced in the ZODA proximity proof to also verify the batched signatures, since the random linear combination y_j is known to satisfy

$$\tilde{X}_j r_j = y_j \tag{3}$$

with extremely high probability once (1) is verified, where $\tilde{X}_j$ is the unique matrix that the columns of X_j decode to.

A helpful exercise is to see that the batched signature verification equation (2) holds when the signatures are all valid. For convenience, define the vector of hash outputs

$$h_j = \big(H\big(0, s_j\big), ..., H\big(\ell, s_j\big)\big) \in \mathbf{G}_1^{\ell+1}$$

and set

$$\big(w_j\big)_i = H\big(0, s_j\big) + \sum_{k=1}^{\ell} \big(\tilde{m}_{ij}\big)_k H\big(k, s_j\big)$$

for each signer $i = 1, ..., n$, and slot index j. Equivalently, using matrix notation,

$$w_j = \begin{bmatrix} \mathbf{1} & \tilde{X}_j^T \end{bmatrix} h_j \in \mathbf{G}_1^n,$$

where $\mathbf{1}$ is the all-ones vector.

Now, let's start by expanding the inner terms on the left hand side of the verification equation (2), using the definition of the signature:

$$\langle \Sigma_{ij}, \mathsf{PK}_i \rangle = \kappa_i^{-1}\kappa_i \left\langle H(0, s_j) + \sum_{k=1}^{\ell} (\tilde{m}_{ij})_k H(k, s_j), g_2 \right\rangle = \left\langle (w_j)_i, g_2 \right\rangle.$$

So, by the bilinearity of the pairing and the above, we have that

$$\sum_{i=1}^{n} (r_j)_i \langle \Sigma_{ij}, \mathsf{PK}_i \rangle = \sum_{i=1}^{n} (r_j)_i \left\langle (w_j)_i, g_2 \right\rangle = \langle r_j^T w_j, g_2 \rangle.$$

Finally, using the ZODA Hadamard guarantee (3), we know the random linear combination $\tilde{X}_j r_j = y_j$, so

$$r_j^T w_j = r_j^T \begin{bmatrix} \mathbf{1} & \tilde{X}_j^T \end{bmatrix} h_j = (r_j^T \mathbf{1}) h_{0j} + (r_j^T \tilde{X}_j^T) \overline{h}_j = (r_j^T \mathbf{1}) h_{0j} + y_j^T \overline{h}_j,$$

where we have set $h_j = (h_{0j}, \overline{h}_j) \in \mathbf{G}_1 \times \mathbf{G}_1^{\ell}$; *i.e.*, $\overline{h}_j \in \mathbf{G}_1^{\ell}$ is the vector h_j without its first element. Putting this all together,

$$\begin{aligned} \sum_{i=1}^{n} (r_j)_i \langle \Sigma_{ij}, \mathsf{PK}_i \rangle &= \langle r_j^T w_j, g_2 \rangle \\ &= \left\langle (r_j^T \mathbf{1}) h_{0j} + y_j^T \overline{h}_j, g_2 \right\rangle \\ &= \left\langle (r_j^T \mathbf{1}) H(0, s_j) + \sum_{k=1}^{\ell} (y_j)_k H(k, s_j), g_2 \right\rangle, \end{aligned}$$

which is exactly the right hand side of the verification equation (2).

3.2.2 Discussion

The main difference from [Bon+09] is that we add the offset factor $H(0, s_j)$ which prevents the adversary from being able to scale any signature Σ_{ij} by a constant factor in $\mathbf{F}$ and receive another valid signature. In turn, if the adversary were to receive multiple signatures for the same slot s_j — that is, if $s_j = s_{j'}$ for any two slot indices $j \neq j'$ — then any affine combination of the two signatures $(1-\alpha)\Sigma_{ij} + \alpha\Sigma_{ij'}$ for any $\alpha \in \mathbf{F}$ and signer i would also be a valid signature for the message $(1-\alpha)\tilde{m}_{ij} + \alpha\tilde{m}_{ij'}$, which would mean that the signature scheme is malleable. We point out some interesting observations of this scheme next.

Verify single. Note that, if verification succeeds, then for each signer $i = 1, ..., n$ and slot index $j = 1, ..., q$, the signature Σ_{ij} is a valid signature of the message $\tilde{m}_{ij}$ for slot s_j with public key PK_i, that is

$$\langle \Sigma_{ij}, \mathsf{PK}_i \rangle = \left\langle H(0, s_j) + \sum_{k=1}^{\ell} (\tilde{m}_{ij})_k H(k, s_j), g_2 \right\rangle. \tag{4}$$

This follows from the ZODA proximity check guarantees (3) combined with (2) and the fact that r_j is uniformly random. This means that, with high probability, the individual claim

holds over each signer i and slot index j. We will make use of the fact that the verification of the batched signatures implies that the individual signatures are valid in the security proof below.

Runtime. The verify algorithm computes a number of pairings proportional to the number of signers, which is roughly the square root of the amount of data in $\tilde{X}$. These pairings depend only on the signing step. This, in turn, makes it possible to use techniques such as inner pairing products proofs [Bün+21] and general SNARKs to reduce the verification cost non-interactively. The usual tricks of performing a single final exponentiation and bringing the random coefficients into one of the source groups also apply here. Note that the randomness that is originally derived using Fiat–Shamir in ZODA now also has to include the signatures and public keys, since we're also using it to aggregate the pairings.

Numerics. Assuming we use BLS12-381, each signature is 48 bytes. For a data matrix of dimensions $\ell \times n$, this means that we add $48n$ bytes overall to be stored by full nodes, since we have a signature for each column. The communication overhead in each sampling instance is $48n$ bytes. The heaviest verifier work is $n + 1$ pairings and a multisum of size $\ell + 1$.

3.3 Security proof

By relying on the algebraic group model (AGM) [FKL18] and the random oracle model [BR93], used here implicitly for the ZODA guarantees, the proof of security is straightforward. In the algebraic group model, the adversary is restricted to outputting group elements that are linear combinations of the group elements it has seen so far. We will show that any adversary that wins the many-slot proximate existential unforgeability game in the AGM with non-negligible probability can be used to solve the $(1, 1)$-dlog problem as defined in [BFL20], assuming the security of the ZODA protocol, which is proven to be secure in the random oracle model in [EMA24].

Problem. The $(1, 1)$-dlog problem is defined as follows. An adversary A is given a challenge consisting of inputs $(g, \alpha g) \in \mathbf{G}_1 \times \mathbf{G}_1$ and $(h, \alpha h) \in \mathbf{G}_2 \times \mathbf{G}_2$. The adversary must then output $\alpha \in \mathbf{F}$ with non-negligible probability. This is assumed to be a hard problem for any reasonable adversary, so reducing the security of our scheme to this problem is sufficient to prove security. Using the observation in [Mae15], and the proofs of [FKL18], we build an adversary for the $(1, 1)$-dlog problem.

Adversary. As a reminder, the adversary for the many-slot batched proximate existential unforgeability is allowed to make a limited amount q_S of queries to a signature oracle for a public key PK on some messages of its choosing, $\tilde{m}_1, ..., \tilde{m}_{q_S} \in \mathbf{F}^\ell$, receiving signatures $\sigma_1, ..., \sigma_{q_S}$. This will correspond to the one honest signer in the game, who we will indicate as the signer index $i = 1$, while the rest of the signers are controlled by the adversary. Eventually, after seeing answers to the signature oracle and the hash oracle, the adversary must produce an encoding $X_{s^*} \in \mathbf{F}^{\ell' \times n}$ for some target slot s^* (where the adversary queried the signing oracle on slot s^* at most once) such that X_{s^*} is 'close to' an encoding of some data matrix

$\tilde{X}_{s^*} \in \mathbf{F}^{\ell \times n}$, along with a signature σ_{s^*} corresponding to the honest signer's signature. Since the rest of the signers are controlled by the adversary, we can assume that the adversary also produces valid signatures for the other signers, which we will not need in the security proof.

Verifying a single signature. From equation (4), we know that, if verification succeeds, then, for each slot index j, verifying a batched signature implies that the individual signatures (and in particular, the signatures corresponding to the honest signer $i = 1$) are valid for the corresponding messages. Specializing to the honest signer, we will write $\sigma_j = \Sigma_{1j}$, $\mathsf{PK} = \mathsf{PK}_1$, and $\tilde{m}_j = \tilde{m}_{1j}$. That is, for each slot index j,

$$\langle \sigma_j,\ \mathsf{PK} \rangle = \left\langle H(0, s_j) + \sum_{k=1}^{\ell} (\tilde{m}_j)_k H(k, s_j),\, g_2 \right\rangle. \tag{5}$$

Since this is true for any slot index j with slot s_j, then, if the adversary wins on some target slot s^*, there exists a uniquely decodable message $\tilde{m}^* \in \mathbf{F}^{\ell}$ such that

$$\langle \sigma^*,\ \mathsf{PK} \rangle = \left\langle H(0, s^*) + \sum_{k=1}^{\ell} (\tilde{m}^*)_k H(k, s^*),\, g_2 \right\rangle. \tag{6}$$

This is exactly the single-signature verification equation for the honest signer on the forged slot. The rest of the reduction shows that any adversary able to produce such a signature can be turned into an algorithm for recovering α.

Reduction. Assume that B_{alg} is an adversary that wins the many-slot proximate existential unforgeability game with non-negligible probability. We build an adversary A that uses B_{alg} to solve the $(1, 1)$-dlog problem.

Oracle simulation. Adversary A receives a challenge $(g, \alpha g) \in \mathbf{G}_1 \times \mathbf{G}_1$ and $(h, \alpha h) \in \mathbf{G}_2 \times \mathbf{G}_2$. It gives B_{alg} the descriptions of $\mathbf{G}_1$ and $\mathbf{G}_2$, sets $g_1 = g$, $g_2 = \alpha h$, and publishes the honest public key as $\mathsf{PK} = h$. With these choices, the honest secret key is $\kappa = \alpha$ since

$$\mathsf{PK} = h = \alpha^{-1}(\alpha h) = \kappa^{-1} g_2.$$

For each random oracle query $H(k, s)$ for index k and slot s, the adversary A samples a fresh $\rho_{ks} \in \mathbf{F}$ uniformly at random and returns

$$H(k, s) = \rho_{ks} g.$$

When B_{alg} asks for a signature on a message $v_j = ((v_j)_1, ..., (v_j)_\ell) \in \mathbf{F}^{\ell}$ in slot s_j, adversary A returns

$$\sigma_j = \left(\rho_{0 s_j} + \sum_{k=1}^{\ell} (v_j)_k \rho_{k s_j} \right) (\alpha g). \tag{7}$$

This is a correct simulation of the signing oracle, since

$$\begin{aligned}\langle \sigma_j,\ \mathsf{PK} \rangle &= \left\langle \left(\rho_{0 s_j} + \sum_{k=1}^{\ell} \left(v_j\right)_k \rho_{k s_j} \right) (\alpha g),\, h \right\rangle \\ &= \left\langle \left(\rho_{0 s_j} + \sum_{k=1}^{\ell} \left(v_j\right)_k \rho_{k s_j} \right) g,\, \alpha h \right\rangle \\ &= \left\langle H(0, s_j) + \sum_{k=1}^{\ell} \left(v_j\right)_k H(k, s_j),\, g_2 \right\rangle.\end{aligned}$$

Algebraic form of the forgery. Let T_H denote the set of slots on which B_{alg} queried the random oracle. Since B_{alg} is algebraic, any forged group element σ^* must be a (known) linear combination of previously-observed group elements. In particular, there exist coefficients β, $\{\gamma_{ks}\}$, and $\{\xi_j\}$, in the field $\mathbf{F}$ and known to the adversary, such that

$$\sigma^* = \beta g + \sum_{s \in T_H} \sum_{k=0}^{\ell} \gamma_{ks} H(k, s) + \sum_{j=1}^{q_S} \xi_j \sigma_j. \tag{8}$$

Now, for convenience, we will write μ_j to be the random linear combination of the oracle outputs and the queried messages:

$$\mu_j = \rho_{0 s_j} + \sum_{k=1}^{\ell} \left(v_j\right)_k \rho_{k s_j},$$

for each signature query j. Similarly for the forged signature, we will write

$$\mu^* = \rho_{0 s^*} + \sum_{k=1}^{\ell} \left(\tilde{m}^*\right)_k \rho_{k s^*}.$$

By construction of the simulated signatures, $\sigma_j = \alpha \mu_j g$ for every signature query j. Additionally, from (6) and the random oracle,

$$\langle \sigma^*, h \rangle = \langle \mu^* g, \alpha h \rangle = \langle \alpha \mu^* g, h \rangle.$$

Finally, using (again) the bilinearity and the fact that the pairing is nondegenerate, note that forged signature must have the same form as the simulated signatures:

$$\sigma^* = \alpha \mu^* g. \tag{9}$$

Extracting the secret key. Substituting $H(k, s) = \rho_{ks} g$ along with $\sigma_j = \alpha \mu_j g$ and (9) into (8) gives

$$\mu^* \alpha = \beta + \sum_{s \in T_H} \sum_{k=0}^{\ell} \gamma_{ks} \rho_{ks} + \alpha \sum_{j=1}^{q_S} \xi_j \mu_j.$$

Rearranging,

$$\alpha\left(\mu^* - \sum_{j=1}^{q_S} \xi_j \mu_j\right) = \beta + \sum_{s \in T_H} \sum_{k=0}^{\ell} \gamma_{ks} \rho_{ks}. \tag{10}$$

Therefore, if the coefficient, which we will write as

$$\eta = \mu^* - \sum_{j=1}^{q_S} \xi_j \mu_j$$

is nonzero, then the adversary A can recover α by taking a simple inverse over $\mathbf{F}$. The only remaining question is to show that the coefficient η is nonzero with very high probability.

Nonzero coefficient. To show this, we consider two cases. First, if no signature query was ever made on slot s^*, then none of the values μ_j depend on ρ_{0s^*}. This means that the coefficient is of the form

$$\eta = \rho_{0s^*} + C$$

for some constant $C \in \mathbf{F}$ that is independent of ρ_{0s^*}. Since ρ_{0s^*} is uniform in $\mathbf{F}$ by construction, then η is therefore zero with probability at most $1/|\mathbf{F}|$, which is assumed to be negligibly small.

Second, suppose there was exactly one signature query on slot s^*, say with slot index j^* (such that $s_{j^*} = s^*$). Then

$$\eta = (1 - \xi_{j^*})\rho_{0s^*} + \sum_{k=1}^{\ell} \Big((\tilde{m}^*)_k - \xi_{j^*} (v_{j^*})_k \Big) \rho_{ks^*} - \sum_{j \neq j^*} \xi_j \mu_j.$$

If $1 - \xi_{j^*} \neq 0$, the same argument as above applies, since $(1 - \xi_{j^*})\rho_{0s^*}$ is also uniform. On the other hand, if $1 - \xi_{j^*} = 0$, then $\xi_{j^*} = 1$. In other words, it must be the case that

$$\eta = \sum_{k=1}^{\ell} \Big((\tilde{m}^*)_k - (v_{j^*})_k \Big) \rho_{ks^*} - \sum_{j \neq j^*} \xi_j \mu_j$$

is zero. But, note that, since, by definition of the security game, the forged message $\tilde{m}^*$ is not the same as any previously signed message, so $(\tilde{m}^*)_k \neq (v_{j^*})_k$ for some k. This means that the corresponding term in the sum is nonzero, and, since ρ_{ks^*} is uniform in $\mathbf{F}$, the probability that η is zero is at most $1/|\mathbf{F}|$, which is assumed to be negligible. Therefore, for any successful forgery, η is zero with negligible probability. This, in turn, means that A can recover α with non-negligible probability as well.

Putting this all together, we get that, whenever B_{alg} wins the many-slot proximate existential unforgeability game with non-negligible probability, the adversary A can solve the $(1, 1)$-dlog problem with non-negligible probability as well, up to the (negligible) soundness error of the ZODA proof and the additional $1/|\mathbf{F}|$ terms above.

Discussion. In the proof above, the challenger decodes the forged message for the honest signer directly. This is only to make the reduction concrete. In the actual verification algorithm, the verifier need not decode each message individually. Once the ZODA proof passes, the verifier already knows that the random linear combination $y_j = \tilde{X}_j r_j$ is correct, and can use the same randomness r_j to batch the pairing checks, as in (2). In other words, the ZODA proof already provides the linear combination of message coordinates needed for batched signature verification. The remaining cost to the verifier is the multi-pairing over the signatures and public keys.

4 Acknowledgements

The authors would like to thank Andrija Novakovic for helpful discussions. We also thank different AIs for assisting in exploration along with helping edit and polish the final draft.

Bibliography

[Al-19] M. Al-Bassam, "Lazyledger: A distributed data availability ledger with client-side smart contracts," *arXiv preprint arXiv:1905.09274*, 2019.

[Eth26] Ethereum Foundation, "Ethereum." 2026.

[Sol26] Solana Foundation, "Solana." 2026.

[Cel26] Celestia Labs, "Celestia." 2026.

[Com26] Commonware, "Commonware." 2026.

[HSW23] M. Hall-Andersen, M. Simkin, and B. Wagner, "Foundations of data availability sampling," *Cryptology ePrint Archive*, 2023.

[EMA24] A. Evans, N. Mohnblatt, and G. Angeris, "ZODA: Zero-Overhead Data Availability." [Online]. Available: https://eprint.iacr.org/2025/034

[Fei22] D. Feist, "New Sharding Design with Tight Beacon and Shard Block Integration." July 2022.

[EA25] A. Evans and G. Angeris, "The Accidental Computer: Polynomial Commitments from Data Availability," *Cryptology ePrint Archive*, 2025.

[Ame99] "Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)," technical report ANSI X9.62, 1999.

[BLS01] D. Boneh, B. Lynn, and H. Shacham, "Short Signatures from the Weil Pairing," in *Advances in Cryptology — ASIACRYPT 2001*, Springer, 2001, pp. 514–532.

[Sch91] C.-P. Schnorr, "Efficient Signature Generation by Smart Cards," in *Journal of Cryptology*, Springer, 1991, pp. 161–174.

[Nat15] "Secure Hash Standard (SHS)," technical report FIPS PUB 180-4, Aug. 2015. [Online]. Available: https://csrc.nist.gov/publications/detail/fips/180/4/final

[Bon+09] D. Boneh, D. Freeman, J. Katz, and B. Waters, "Signing a linear subspace: Signature schemes for network coding," in *International workshop on public key cryptography*, 2009, pp. 68–87.

[GJN25] K. Gurkan, P. Jovanovic, and A. Novakovic, "OOPS: One-time Oblivious Polynomial Signatures," *Cryptology ePrint Archive*, 2025.

[Dzi+25] S. Dziembowski, S. Faust, P. Kędzior, M. Mielniczuk, S. K. Mohanty, and K. Pietrzak, "Beholder Signatures," *Cryptology ePrint Archive*, 2025.

[BSW06] D. Boneh, E. Shen, and B. Waters, "Strongly unforgeable signatures based on computational Diffie-Hellman," in *International Workshop on Public Key Cryptography*, 2006, pp. 229–240.

[Ame+17] S. Ames, C. Hazay, Y. Ishai, and M. Venkitasubramaniam, "Ligero: Lightweight sublinear arguments without a trusted setup," in *Proceedings of the 2017 acm sigsac conference on computer and communications security*, 2017, pp. 2087–2104.

[Ben+18] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev, "Fast reed-solomon interactive oracle proofs of proximity," in *45th international colloquium on automata, languages, and programming (icalp 2018)*, 2018, pp. 14–11.

[Arn+25] G. Arnon, A. Chiesa, G. Fenzi, and E. Yogev, "WHIR: Reed–Solomon proximity testing with super-fast verification," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, 2025, pp. 214–243.

[NA25] A. Novakovic and G. Angeris, "Ligerito: A Small and Concretely Fast Polynomial Commitment Scheme," *Cryptology ePrint Archive*, 2025.

[HSW24] M. Hall-Andersen, M. Simkin, and B. Wagner, "FRIDA: Data Availability Sampling from FRI," in *Advances in Cryptology - CRYPTO 2024 - 44th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2024, Proceedings, Part VI*, L. Reyzin and D. Stebila, Eds., in Lecture Notes in Computer Science, vol. 14925. Springer, 2024, pp. 289–324. doi: 10.1007/978-3-031-68391-6_9.

[Bün+21] B. Bünz, M. Maller, P. Mishra, N. Tyagi, and P. Vesely, "Proofs for inner pairing products and applications," in *International conference on the theory and application of cryptology and information security*, 2021, pp. 65–97.

[FKL18] G. Fuchsbauer, E. Kiltz, and J. Loss, "The algebraic group model and its applications," in *Annual International Cryptology Conference*, 2018, pp. 33–62.

[BR93] M. Bellare and P. Rogaway, "Random oracles are practical: A paradigm for designing efficient protocols," in *Proceedings of the 1st ACM Conference on Computer and Communications Security*, 1993, pp. 62–73.

[BFL20] B. Bauer, G. Fuchsbauer, and J. Loss, "A classification of computational assumptions in the algebraic group model," in *Annual International Cryptology Conference*, 2020, pp. 121–151.

[Mae15] Maeher, "Re: Discrete Logarithm problem with inverse." [Online]. Available: https://crypto.stackexchange.com/a/29961

[Wal] G. Walker, "Schnorr Signatures."

A A BLS-based construction

We briefly describe a simpler variant based directly on BLS signatures. Here signatures lie in $\mathbf{G}_1$ and public keys lie in $\mathbf{G}_2$. For a secret key $\alpha \in \mathbf{F}$, the signature of an element z is $\alpha H(z) \in \mathbf{G}_1$, where H hashes to $\mathbf{G}_1$, and the corresponding public key is $\mathsf{PK} = \alpha g_2 \in \mathbf{G}_2$. As in the main construction, the proximity guarantee comes from an associated ZODA proof; the difference here is that the sampled encoded elements are signed directly.

Signing a message. Signer j encodes the message $\tilde{m}_j$ to get m_j. (Here, as before, each column corresponds to a signer.) The signer then individually signs each element $(m_j)_i$ in the encoded column with its private key α_i. These signatures are auxiliary data attached to the column and are not part of the encoding itself. It then sends the encoded column and signatures to a *full node*, who will assemble these columns into an encoded matrix X (where $X_{ij} = (m_j)_i$) and later aggregate the corresponding signatures during sampling, as we will see next.

Aggregation during sampling. The verifier samples a set of rows S, as in the ZODA proof. The full node serving the data then returns the sampled entries of the encoded columns in these positions. For each column j, it also aggregates the corresponding signatures by adding the signatures on the sampled entries X_{ij} for $i \in S$. That is, for each column j, the full node sends an aggregate signature

$$\sigma_j = \alpha_j \sum_{i \in S} H(X_{ij}).$$

Verification. The verifier checks each signature σ_j by verifying that

$$\langle \sigma_j, g_2 \rangle = \left\langle \sum_{i \in S} H(X_{ij}), \mathsf{PK}_j \right\rangle.$$

It is possible to optimize this slightly by checking a random linear combination instead (which reduces the number of pairings from $2n$ to $n + 1$) and batching the final exponentiation as follows.

First, write each pairing equation as:

$$-\langle \sigma_j, g_2 \rangle + \left\langle \sum_{i \in S} H(X_{ij}), \mathsf{PK}_j \right\rangle = 0.$$

Second, use a random linear combination to batch the checks:

$$\sum_{j=1}^{n} \rho^j \left(-\langle \sigma_j, g_2 \rangle + \left\langle \sum_{i \in S} H(X_{ij}), \mathsf{PK}_j \right\rangle \right) = 0.$$

Finally, use the bilinearity and properties of pairings to achieve the following:

$$\text{FinalExp}\left(\text{MillerLoop}\left(\sum_{j=1}^{n}\rho^j\sigma_j, -g_2\right) + \sum_{j=1}^{n}\text{MillerLoop}\left(\rho^j\sum_{i\in S}H(X_{ij}), \mathsf{PK}_j\right)\right)$$

As above, the ZODA proof is what certifies that the sampled data comes from a uniquely decodable encoding; the pairing checks here are only for the signatures on the sampled encoded entries.

Numerics. Assuming we use BLS12-381, each signature is 48 bytes. For a data matrix of dimensions $m \times n$, this means that we add $48mn$ bytes overall to be stored by full nodes. The computational overhead for full nodes in each sampling instance is $n|S|$ group additions, with $|S|$ being the sample size. The communication overhead in each sampling instance is $48n$ bytes. The heaviest verifier work is $n + 1$ pairings, a multisum of size n, and $n|S|$ hashes to the group.

Numerics for a Schnorr alternative. We could also look at a construction that, similar to the construction above, requires full nodes to store all the signatures, without the benefit of per-column aggregation. One reason we would consider it would be the possibility to use smaller curves, such as when using Ed25519, and to hash to a field rather than a group. In this case, the numerics become the following: $32mn$ bytes storage overhead but no computational overhead for full nodes. The communication overhead in each sampling instance is $32n|S|$. The heaviest verifier work is two multisums of size $n|S|$ when using batch verification [Wal].

Guarantees. The guarantee is the same as in the main construction. The ZODA proof shows that the sampled data is close to a uniquely decodable encoding, while the BLS checks show that the sampled encoded elements were signed by the claimed signers. Taken together, these imply that the uniquely decodable messages corresponding to the accepted encoding are the ones signed by the signers.

Discussion. The main challenge of the BLS variant of this construction is that it requires storing a signature for each element in the data matrix, and full nodes must perform aggregation for each sampling instance. The Schnorr variant avoids this aggregation step, but requires much more communication and many more signature checks.