FRIDA: Data Availability Sampling from FRI

Mathias Hall-Andersen*[1] Mark Simkin [2] Benedikt Wagner† [3,4]

February 15, 2024

[1] Aarhus University
ma@cs.au.dk
[2] Ethereum Foundation
mark.simkin@ethereum.org
[3] CISPA Helmholtz Center for Information Security
benedikt.wagner@cispa.de
[4] Saarland University

Abstract

As blockchains like Ethereum continue to grow, clients with limited resources can no longer store the entire chain. Light nodes that want to use the blockchain, without verifying that it is in a good state overall, can just download the block headers without the corresponding block contents. As those light nodes may eventually need some of the block contents, they would like to ensure that they are in principle available.

Data availability sampling, introduced by Bassam et al., is a process that allows light nodes to check the availability of data without download it. In a recent effort, Hall-Andersen, Simkin, and Wagner have introduced formal definitions and analyzed several constructions. While their work thoroughly lays the formal foundations for data availability sampling, the constructions are either prohibitively expensive, use a trusted setup, or have a download complexity for light clients scales with a square root of the data size.

In this work, we make a significant step forward by proposing an efficient data availability sampling scheme without a trusted setup and only polylogarithmic overhead. To this end, we find a novel connection with interactive oracle proofs of proximity (IOPPs). Specifically, we prove that any IOPP meeting an additional consistency criterion can be turned into an erasure code commitment, and then, leveraging a compiler due to Hall-Andersen, Simkin, and Wagner, into a data availability sampling scheme. This new connection enables data availability to benefit from future results on IOPPs. We then show that the widely used FRI IOPP satisfies our consistency criterion and demonstrate that the resulting data availability sampling scheme outperforms the state-of-the-art asymptotically and concretely in multiple parameters.

Keywords: Data Availability Sampling, Interactive Oracle Proofs, FRI, Commitments, Reed-Solomon Codes

*Funded by the Concordium Foundation.
†Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 507237585.

Contents

1 Introduction **3**
1.1 Our Contributions . . . 4
1.2 Related Work . . . 4
1.3 Technical Overview . . . 5

2 Preliminaries **6**

3 From IOPPs to Data Availability Sampling **9**

4 Instantiation from FRI **17**
4.1 Construction . . . 17
4.2 Analysis . . . 18
4.3 Extensions: Batched FRI . . . 25

5 Efficiency Evaluation **25**

A Background on Data Availability Sampling **32**

B Merkle Trees **33**

C Extension: Batched FRI **35**
C.1 Construction . . . 36
C.2 Analysis . . . 36

D Script for Parameter Computation **40**

1 Introduction

A blockchain is a distributed system, running among a set of parties known as full nodes. These parties keep track of a large and continuously growing dataset by storing it redundantly on each node. This provides high levels of fault tolerance, even if all but one nodes crash the data is preserved, but also induces a large storage overhead on the network. The data is structured as sequence of blocks, which themselves are comprised of a small block header and a larger block content. Full nodes store all blocks and always ensure that both the headers and the contents are well-formed. As blockchains continue to steadily gain more and more popularity, they are also required to store more and more data. For a blockchain like Ethereum, a full node currently needs to continuously store hundreds of gigabytes of data.

To make blockchains more accessible to a wider public, who may not be willing to store this much data, one can also participate in these distributed systems as a light node. Such nodes do not store the block contents, but only keep track of the headers. Having access to only the block headers does not allow them to verify the validity of the whole blockchain. Still, light nodes can performing basic operations, such as sending and receiving digital currency, assuming that blockchain is indeed in a valid state and assuming that the nodes can access whatever data they need. While light nodes may not be interested in storing all the data, they still would like to ensure the data is in principle available in the network, in case they do need to download parts of it. A naive way for the light nodes to check the availability of data is to just download it, but this would be at odds with the goal of being a light node. Preferably, they would like a means of checking the availability of all data that is more communication efficient than this naive approach.

Data Availability Sampling. The concept of data availability sampling (DAS), originally introduced by Bassam et al. [ASBK21], addresses the problem outlined above. In this setting we have a possibly malicious block proposer, who encodes a bit string data, representing the block content, into a short commitment com and a longer encoding string π. The commitment com can then be added to the block header and allows light nodes to ensure the full availability of data via a communication-efficient interactive protocol between them and the network that claims to store the data encoding π. We note that clearly no individual light nodes can make sure that all the data is fully available, unless they download it in full. However, multiple light nodes can separately query the network and if they individually all like the responses they see, then they know that all of the data is available, despite no single light node having downloaded all of it. Bassam et al. present a solution for DAS based on two-dimensional erasure codes and Merkle trees. Their solution has been the basis for a similar construction that is soon going to be deployed within the Ethereum blockchain[1]. For a cryptographic primitive that is being deployed in the real world, in a system with millions of financial transactions per day[2], it is undoubtedly of crucial importance to understand both the precise security guarantees and the best possible constructions.

Foundations of Data Availability Sampling. A recent work by Hall-Andersen, Simkin, and Wagner [HASW23] formally defines the concept of DAS schemes and provides several provably secure construction with different trade-offs. An important insight from their work is a reduction from the DAS problem to a conceptually simpler primitive they call erasure code commitments. Informally, an erasure code is a function that takes a message as input and produces a codeword, which is a vector of symbols. Even if some of the symbols are lost, one can still recover the original message from the remaining ones. An erasure code commitment allows for taking an encoded message and committing to it in way that ensures that any opened symbol for a given commitment is consistent with the *same* overall codeword.

To see why this is non-trivial, it is helpful to consider a simple, but flawed construction from vector commitments. Naively, one could imagine using a vector commitment to commit to all the symbols of a codeword, i.e. the i-th entry in the vector is the i-th symbol of the codeword. Opening individual positions allows for checking individual symbols, but a vector commitment provides no guarantees that all individual symbols will satisfy some overarching requirement like being consistent with a unique codeword.

Existing Constructions. The work of Hall-Andersen, Simkin, and Wagner presents four constructions, all of which have certain drawbacks. The first construction they consider combines an arbitrary vector commitment with a succinct non-interactive argument of knowledge (SNARKs). Unfortunately, SNARKs

[1] https://ethereum.org/developers/docs/data-availability
https://ethereum.org/roadmap/danksharding

[2] https://etherscan.io/txs

currently induce large computational overheads and require non-falsifiable assumptions [GW11]. The second construction they consider is the one that is envisioned by Ethereum, which relies on two-dimensional Reed-Solomon codes in combination with the polynomial commitment scheme of Kate, Zaverucha, and Goldberg [KZG10]. Those commitments require a trusted setup, which is difficult to realize in practice. Even for a blockchain as large as Ethereum, who recently ran such a modestly sized setup, this took significant engineering and community efforts[3]. Other distributed systems may not have the possibility of running such a large-scale distributed setup and ideally, we would like to just avoid such setups. The third and fourth construction they consider rely on (homomorphic) collision-resistant hash functions and the random oracle model. These constructions have asymptotically and concretely larger commitments and bandwidth overheads than the first two. In a DAS scheme constructed from erasure code commitments, the light nodes randomly probe symbols and verify them against the commitment, meaning that they need to download both the commitment and some of the symbols. For an input data of size D, the commitments in the last two constructions are of size $\Omega(\lambda\sqrt{D})$ and each symbol is of size $\Omega(\sqrt{D})$, where λ is the security parameter. Ideally, we would like to reduce the bandwidth overhead of light nodes by having only a *polylogarithmic* dependency on D.

1.1 Our Contributions

In this work we propose a new data availability sampling scheme by constructing a new erasure code commitment. Our construction has the following characteristics:

Model and Assumptions: Our construction does not require a trusted setup and is information-theoretically secure in the random oracle model.

Efficiency: For input data of size D, commitments of our scheme are of size $\mathcal{O}(\lambda^2 \log^2(D))$, the encoding is of size $\lambda D \log^2(D)$, and each symbol is of size $\lambda \log^2(D)$.

Compared to the previous state of the art for hashing-based constructions, this is a significant theoretical improvement as we go from $\Omega(\sqrt{D})$ to $\mathcal{O}(\log^2(D))$ overheads. In the context of blockchains this means that our construction is friendlier to light nodes, as these only retrieve the commitments and retrieve symbols during the availability sampling. On the flip side, we induce a larger storage overhead on the network that holds the encoded data. When supporting light nodes is the main goal, then our solution in this work is preferable over previous hashing-based ones. When supporting light nodes efficiently without a trusted setup is the main goal, our solution is preferable over all previous approaches.

DAS from Special IOPPs. On a technical level, we prove a formal connection between DAS and interactive oracle proofs of proximity (IOPPs) [BBHR18] with certain additional properties. IOPPs have recently received a large amount of interest, due to their applications in succinct proof systems [BCG+16, BBHR18, BBHR19, BCR+19, BCG+19, BGKS20, BCI+20a, COS20]. Using the connection between IOPPs and DAS established in this work, improvements on IOPPs can lead to better DAS schemes in the future.

Special IOPPs from FRI. We show that the IOPPs with additional properties that we define in this work can be instantiated. More concretely, we show that the fast Reed-Solomon (FRI) IOPP of Ben-Sasson et al. [BBHR18] satisfies our IOPP notion and can, therefore, be used to instantiate our framework. Proving that the construction of Ben-Sasson et al. satisfies our extended notion of an IOPP constitutes the main technical contribution of this work. We call our FRI-based construction FRIDA[4].

Benchmarks. Finally, we provide benchmarks in our work for various parameter ranges and compare the construction in this work with those of Hall-Andersen, Simkin, and Wagner [HASW23] .

1.2 Related Work

Several existing cryptographic primitives appear superficially quite similar to DAS, but are actually distinctly different. In the following we will briefly discuss the most important ones among them and we refer the reader to the more elaborate comparison provided in the work of Hall-Andersen, Simkin, and Wagner [HASW23].

[3] https://ceremony.ethereum.org

[4] Named in honor of Frida Lyngstad from the pop band ABBA, definitely nothing to do with the fact that it gives us a FRI-based data availability sampling scheme.

Proofs of Retrievability. Proofs of retrievability [JK07, ABC+07, SW08, DVW09, CKW13, SSP13] allow a client to encode some data and store it on a possibly untrusted server. In this setting the client is trusted, meaning that encodings are always generated honestly, and availability of data is defined via an extractor, which can recover the full encoded data from a polynomial number of interactions with the server storing the data. In the case of DAS, we do not assume that the encoding is generated honestly and the number of interactions with the encoding that are needed to recover the data is an efficiency metric and not a proof artefact.

Verifiable Information Dispersal. In verifiable information dispersal [Rab89, CT05, NNT21] a possibly malicious client encodes some data and then runs an interactive protocol with n server out of which at most t are malicious to store the encoded data on them. In DAS, there is no need to run an interactive protocol and one also does not need to know how or where exactly the encoded data is stored. This is indeed important in real-world applications of DAS, where the storage servers may not be known upfront and where they may change over time. As a cryptographic primitive DAS allows for more flexibility as it is fully agnostic to the underlying networking.

Functional Commitments. Erasure code commitments can be seen as a special case of a more general primitive known as functional commitments [LRY16]. It is natural to ask, whether one could simply use an existing functional commitment as an erasure code commitment. This approach does currently not seem to lead anywhere as existing functional commitment constructions [BNO21, CFT22, WW23, dCP23, BCFL23] appear to be less efficient than existing erasure code commitments.

1.3 Technical Overview

Before proceeding with the technical content of this work, let us provide some high-level intuition for what erasure code commitments are, what IOPPs are, and how we aim to construct one from the other.

Erasure Code Commitments. Let $\mathcal{C}\colon \Gamma^k \to \Lambda^n$ be an erasure code and let us denote the image of that code by $\mathcal{C}(\Gamma^k)$. An erasure code commitment for a code $\mathcal{C}$ takes a message $m \in \Gamma^k$ as input and produces a commitment com, which can be seen as a vector commitment to the symbols $\mathcal{C}(m) \in \Lambda^n$ with an additional security guarantee. Similar to a standard vector commitment, it should be position-binding in the sense that no index $i \in [n]$ can be opened to two different symbols. Additionally, the commitment should satisfy a notion called code-binding, which ensures that for any subset of positions $I \subseteq [n]$, the corresponding opened symbols at those positions are consistent with at least one codeword in the image of $\mathcal{C}$. For an erasure code, which ensures that any t symbols allow for reconstructing the original message, code-binding means that any for subset $I \subseteq [n]$ with $|I| \geq t$ the opened symbols must be consistent with exactly one codeword. Note that this rules out naive constructions in which one simply applies a standard vector commitment to the codeword.

IOPPs. An interactive oracle proof of proximity for some code $\mathcal{C}$ is a protocol between a prover and a public-coin verifier. In this context, oracles can be thought of as vectors of symbols in the sky, which can be accessed in arbitrary positions without needing to read the whole vector. Both parties start with oracle access to a vector of symbols and the prover claims that this vector is close to the code $\mathcal{C}$. By close we mean that there is at least one codeword that does not differ from the vector in the oracle in too many positions. We say that the vector in the oracle is within the the unique decoding radius, if there is exactly one such codeword, i.e., the closest codeword is unique. The verifier wants to be convinced that the prover's statement is true, but wants to read as few symbols as possible from the oracle. Towards this goal, the prover and the verifier engage in multiple rounds of interactions. In each round the prover will provide a new oracle to the verifier, who may read some positions in the given oracle and provide the prover with fresh random coins. At the end of the interaction, the verifier will output a bit, indicating whether they believe the prover's claim or not. IOPPs should satisfy a standard completeness notion as well as a soundness notion, which states that no malicious prover can falsely convince the verifier of a codeword being close to the code, despite it being far away. Since the verifier is a public-coin algorithm, one can use the Fiat-Shamir transformation and replace them by a random oracle, thereby making the proof non-interactive. To avoid relying on the idealized model with oracles in the sky, one can use Merkle trees and let the prover commit to the vectors it would have placed in the oracles.

From IOPP to Erasure Code Commitment. Conceptually, IOPPs have some semblance of erasure code commitments. The initial oracle can be seen as a commitment to some encoded message and the remaining IOPP transcript can be seen as a proof of "well-formedness" for that commitment. To open

a symbol in the committed vector of symbols, one can provide a Merkle path consistent with the root node value for the first oracle. If, during the IOPP, we require the prover to convince the verifier that the codeword in that oracle is close in the sense of being within the unique decoding radius, then intuitively it may appear to directly give us a valid erasure code commitment construction.

Unfortunately, this approach is flawed. Note that a codeword being close to a code does not mean being exactly a valid codeword. The vector in the initial oracle may agree with some unique valid codeword on most positions, but may still be erroneous in a few other positions. Let us again assume that the used code ensures that any t symbols allow for recovering the original message. The prover could open $t-1$ valid positions and 2 erroneous ones, which would violate code-binding as those $t+1$ points are not consistent a unique codeword.

To get around this issue, we define a new notion of *opening-consistency* for IOPPs, which basically prevents this bad event from happening. It ensures that, given a valid proof transcript, one cannot open any of the symbols in the initial oracle that are not consistent with the unique closest codeword.

We then formally prove that FRI, the IOPP construction of Ben-Sasson et al. [BBHR18], has this new property. On an intuitive level, FRI achieves opening-consistency by not only opening symbols in the initial oracle, but also opening correlated additional symbols in all subsequent oracles. These additionally revealed symbols from the other oracles ensure that the opened symbol in the initial oracle is well-formed.

Proving that FRI has opening-consistency is surprisingly difficult and requires a rather involved technical proof. It requires us to redo part of the original FRI analysis and then some more. We believe that the notion of opening-consistency as well as our proof that FRI satisfies this property may be of independent interest, beyond their applications in this work.

Why Not Use FRI as a Polynomial Commitment? The astute reader may notice that one can construct polynomial commitments from FRI [VP19], which can be seen as erasure code commitments for Reed-Solomon codes. This is indeed possible and would result in a valid construction, albeit at the cost of being both asymptotically and concretely less efficient. Concretely, the construction would have openings that are larger by a λ multiplicative factor. The difference between this approach and ours has to do with the openings that are supported: a polynomial commitment can be opened over the entire field, but erasure code commitments for the Reed-Solomon code only need to support openings over the evaluation domain. This difference allows us to obtain a more efficient that does not require the generic approach via generic polynomial commitments.

2 Preliminaries

In this section, we recall some necessary background and fix notation.

Notation. The set of the first L natural numbers is denoted as $[L] = \{1, \ldots, L\} \subseteq \mathbb{N}$. Let S be a finite set. Then we write $s \xleftarrow{\$} S$ to say that s is sampled uniformly at random from S. We always assume algorithms to be probabilistic unless stated otherwise. Let Alg be an algorithm. We denote the (upper bound on the) running time of Alg by $\mathbf{T}(\mathsf{Alg})$. If Alg is deterministic, then $y := \mathsf{Alg}(x)$ means that we run Alg on input x and y is assigned to the output. If Alg is probabilistic, then we instead write $y := \mathsf{Alg}(x; \rho)$ to make the random coins ρ of Alg explicit. When we write $y \leftarrow \mathsf{Alg}(x)$, then ρ should be understood as uniform. If we write $y \in \mathsf{Alg}(x)$, we mean that y is a possible output of Alg on input x. We use standard cryptographic notions such as a security parameter λ, PPT (probabilistic polynomial-time) algorithms, and negligible functions.

Codes and Distance. An erasure code is a mapping $\mathcal{C} \colon \Gamma^k \to \Lambda^n$. It maps a message of length k over alphabet Γ to a codeword of length n over alphabet Λ. The code has *reception efficiency* t if any t symbols of a codeword are sufficient to reconstruct the message. Throughout, we will not only identify the code with the mapping, but also with the image $\mathcal{C}(\Gamma^k)$ and simply write $c \in \mathcal{C}$ to state that c is in image of $\mathcal{C}$. For strings $x \in \Sigma^\ell$ and $y \in \Sigma^\ell$ with the same length ℓ over the same alphabet Σ, we write $\delta(x, y)$ to denote the *relative Hamming distance*, i.e., $\delta(x, y) := |\{i \in [\ell] \mid x_i \neq y_i\}|/\ell$. A code $\mathcal{C}$ as above has *minimum distance* δ if $\delta = \min\{\delta(x, y) \mid x, y \in \mathcal{C}, x \neq y\}$. The *unique decoding radius* of $\mathcal{C}$ is the maximum distance δ^* such that for every x with $\delta(x, \mathcal{C}) \leq \delta^*$, then there is a unique closest codeword for x. One can see that $\delta^* = \lfloor (\delta n - 1)/2 \rfloor / n$.

Erasure Code Commitments. We recall the definition of erasure code commitments from [HASW23]. In a nutshell, such a commitment scheme allows to commit to codewords $c \in \mathcal{C}$ in some code $\mathcal{C}$.

Definition 1 (Erasure Code Commitment Scheme)**.** Let $\mathcal{C}\colon \Gamma^k \to \Lambda^n$ be an erasure code. An erasure code commitment scheme for $\mathcal{C}$ is a tuple $\mathsf{CC} = (\mathsf{Setup}, \mathsf{Com}, \mathsf{Open}, \mathsf{Ver})$ of PPT algorithms, with the following syntax:

- $\mathsf{Setup}(1^\lambda) \to \mathsf{ck}$ takes as input the security parameter and outputs a commitment key ck.
- $\mathsf{Com}(\mathsf{ck}, m) \to (\mathsf{com}, St)$ takes as input a commitment key ck and a string $m \in \Gamma^k$, and outputs a commitment com and a state St.
- $\mathsf{Open}(\mathsf{ck}, St, i) \to \tau$ takes as input a commitment key ck, a state St, and an index $i \in [n]$, and outputs an opening τ.
- $\mathsf{Ver}(\mathsf{ck}, \mathsf{com}, i, \hat{m}_i, \tau) \to b$ is deterministic, takes as input a commitment key ck, a commitment com, and index $i \in [n]$, a symbol $\hat{m}_i \in \Lambda$, and an opening τ, and outputs a bit $b \in \{0,1\}$.

Further, we require that the following completeness property holds: For every $\mathsf{ck} \in \mathsf{Setup}(1^\lambda)$, every $m \in \Gamma^k$, and every $i \in [n]$, we have

$$\Pr\left[\mathsf{Ver}(\mathsf{ck}, \mathsf{com}, i, \hat{m}_i, \tau) = 1 \;\middle|\; \begin{array}{l} (\mathsf{com}, St) \leftarrow \mathsf{Com}(\mathsf{ck}, m), \\ \hat{m} := \mathcal{C}(m), \\ \tau \leftarrow \mathsf{Open}(\mathsf{ck}, St, i) \end{array}\right] = 1.$$

Most importantly, erasure code commitments should be position-binding and code-binding, as defined next. Informally, position-binding ensures that no adversary can open the codeword at one position to two different values. Code-binding states that whatever an adversary opens is consistent with the code, i.e., the objects to which one commits are really codewords.

Definition 2 (Position-Binding of CC)**.** Let $\mathsf{CC} = (\mathsf{Setup}, \mathsf{Com}, \mathsf{Open}, \mathsf{Ver})$ be an erasure code commitment scheme for an erasure code $\mathcal{C}$. We say that CC is position-binding, if for every PPT algorithm $\mathcal{A}$, the following advantage is negligible:

$$\mathsf{Adv}^{\mathsf{pos\text{-}bind}}_{\mathcal{A},\mathsf{CC}}(\lambda) := \Pr\left[\begin{array}{ll} & \hat{m} \neq \hat{m}' \\ \wedge & \mathsf{Ver}(\mathsf{ck}, \mathsf{com}, i, \hat{m}, \tau) = 1 \\ \wedge & \mathsf{Ver}(\mathsf{ck}, \mathsf{com}, i, \hat{m}', \tau') = 1 \end{array} \;\middle|\; \begin{array}{l} \mathsf{ck} \leftarrow \mathsf{Setup}(1^\lambda), \\ (\mathsf{com}, i, \hat{m}, \tau, \hat{m}', \tau') \leftarrow \mathcal{A}(\mathsf{ck}) \end{array}\right].$$

Definition 3 (Code-Binding of CC)**.** Let $\mathsf{CC} = (\mathsf{Setup}, \mathsf{Com}, \mathsf{Open}, \mathsf{Ver})$ be an erasure code commitment scheme for an erasure code $\mathcal{C}$. We say that CC is code-binding, if for every PPT algorithm $\mathcal{A}$, the following advantage is negligible:

$$\mathsf{Adv}^{\mathsf{code\text{-}bind}}_{\mathcal{A},\mathsf{CC}}(\lambda) := \Pr\left[\begin{array}{l} \neg\left(\exists c \in \mathcal{C}(\Gamma^k) : \forall i \in I : c_i = \hat{m}_i\right) \\ \wedge\ \forall i \in I :\ \mathsf{Ver}(\mathsf{ck}, \mathsf{com}, i, \hat{m}_i, \tau_i) = 1 \end{array} \;\middle|\; \begin{array}{l} \mathsf{ck} \leftarrow \mathsf{Setup}(1^\lambda), \\ (\mathsf{com}, I, (\hat{m}_i, \tau_i)_{i \in I}) \leftarrow \mathcal{A}(\mathsf{ck}) \end{array}\right].$$

Data Availability Sampling. One of the main observations of Hall-Andersen, Simkin, and Wagner [HASW23] is that erasure code commitments that are position-binding and code-binding can be generically and efficiently transformed into data availability sampling schemes. A data availability sampling scheme, as defined in [HASW23], allows constrained clients to verify that some data is available when having oracle access to an encoding provided by an untrusted party. The precise meaning of available is specified below. More concretely, in the honest setting, a party encodes some data data into an encoding π and a short commitment com. The clients download π and want to verify that some data is available. To do so, they can query π at arbitrary positions and then decide whether to accept or reject. We say that data is *available* if it can be extracted from the resulting transcripts. *Completeness* requires that, given all parties are honest and there are enough clients, the original data data is extracted. *Soundness* ensures that if enough clients accept, then some data is available, i.e., one can extract something from the transcripts. This should hold if the commitment and all responses provided to the clients are determined by the adversary, meaning we make no assumption about the network that stores the encoding. Furthermore, we require *consistency*: no two sets of transcripts for the same commitment result in different data.

In [HASW23], it has been shown that one can obtain a data availability sampling scheme as follows: the data data is encoded using an erasure code. The commitment com is an erasure code commitment. The encoding π contains the symbols of the codeword along with the corresponding openings for the

commitment. Clients sample random positions, query them, and check that the opening verifies. How to sample positions is studied extensively in [HASW23], but for simplicity the reader may think of sampling positions uniformly and independently at random. Reconstructing the data from enough transcripts is possible by the properties of an erasure code. That is, the parameters for completeness and soundness depend on the parameters of the code. Further, one can show that consistency follows from position-binding and code-binding. We give an informal summary in the following theorem. For completeness, we give more detailed background,including the formal definition of data availability sampling schemes in Appendix A.

Theorem 1 ([HASW23], Informal). *Any erasure code commitment scheme that is position-binding and code-binding can be transformed into secure data availability sampling scheme.*

Interactive Oracle Proofs of Proximity. In a (public-coin) interactive oracle proof (IOP) [BCS16] a prover and a verifier are interacting in a number of rounds. In each round i, the verifier first sends a random challenge ρ_i, and the prover responds with a proof string π_i, to which the verifier has random (also known as oracle) access. The verifier finally makes queries to the π_i and decides to accept (output 1) or reject (output 0). In this work, we consider IOPs of proximity (IOPPs), which are closely related to IOPs. Namely, in an IOPP for a code[5] $\mathcal{C}$, the first oracle given to the verifier is an alleged codeword $\mathbf{c}$ claimed to be in $\mathcal{C}$. The IOPP convinces the verifier that $\mathbf{c}$ is close to $\mathcal{C}$. In the following definition, we restrict ourselves to public coin IOPPs, where all challenges sent by the verifier are sampled uniformly at random. With this, we can model the verifier by a single deterministic algorithm V that gets as explicit input all challenges ρ_i sent to the prover and a final random tape ρ_{r+1}. It gets oracle access to all proof strings (aka oracles) and outputs 1 (for accept) or 0 (for reject).

Definition 4 (IOP of Proximity)**.** Let $\mathcal{C}$ be a code. An interactive oracle proof of proximity (IOPP) for $\mathcal{C}$ with r rounds and proximity error β is a pair (P, V) of algorithms such that the following hold:

- **Completeness.** For every $\mathbf{c} \in \mathcal{C}$, we have

$$\Pr_{\rho_1,\ldots,\rho_{r+1}}\left[\mathsf{V}^{\mathbf{c},\pi_1,\ldots,\pi_r}(\rho_1,\ldots,\rho_r,\rho_{r+1}) = 1 \;\middle|\; \begin{array}{r} \pi_1 := \mathsf{P}(\mathbf{c},\rho_1), \\ \vdots \\ \pi_r := \mathsf{P}(\mathbf{c},\rho_1,\ldots,\rho_r) \end{array}\right] = 1.$$

- **Proximity.** For every $0 \le \delta \le 1$ and every $\mathbf{c}$ for which $\delta(\mathcal{C},\mathbf{c}) > \delta$, and every (unbounded) $\tilde{\mathsf{P}}$, we have

$$\Pr_{\rho_1,\ldots,\rho_r,\rho_{r+1}}\left[\mathsf{V}^{\mathbf{c},\pi_1,\ldots,\pi_r}(\rho_1,\ldots,\rho_r,\rho_{r+1}) = 1 \;\middle|\; \begin{array}{r} \pi_1 := \tilde{\mathsf{P}}(\mathbf{c},\rho_1), \\ \vdots \\ \pi_r := \tilde{\mathsf{P}}(\mathbf{c},\rho_1,\ldots,\rho_r) \end{array}\right] \le \beta(\delta).$$

The definition of proximity is merely given for completeness. For our transformation, we will need a more fine-grained notion which is not satisfied (with meaningful parameters) by every IOPP. The following two definitions introduce notation and terminology for IOPPs. Namely, we define what constitutes transcripts, and we introduce a notation to denote the queries made by the verifier.

Definition 5 (Transcripts)**.** Let (P, V) be an IOPP with r rounds for a code $\mathcal{C}$. We define:

- **Complete Transcripts.** A sequence $T = (\mathbf{c}, \rho_1, \pi_1, \ldots, \rho_r, \pi_r, \rho_{r+1})$ containing an alleged codeword $\mathbf{c}$, verifier messages ρ_i, and proof strings π_i is called a complete transcript.
- **Partial Transcript.** Let T be a complete transcript. Any prefix T' of T is called a partial transcript, where T' is called a prover-turn (resp. verifier-turn) transcript if T' contains an even (resp. odd) number of elements.
- **Transcript Extension.** Let T be a partial transcript and m be any message. Then, $T \circ m$ denotes the unique partial transcript that ends with m, contains one element more than T, and has T as a prefix, i.e., the concatenation of T and m.

[5] Usually, IOPPs are defined for families of codes. We assume the code is fixed.

Definition 6 (Query Sets of IOPP)**.** Let $\mathcal{C}$ be a code, and let (P, V) be an IOPP for $\mathcal{C}$ with r rounds. Let $T = (\mathbf{c}, \rho_1, \pi_1, \ldots, \rho_r, \pi_r, \rho_{r+1})$ be any complete transcript. We define the following sets:

$$\mathcal{Q}_0(T) := \{j \in |\mathbf{c}| \mid \mathsf{V}^{\mathbf{c},\pi_1,\ldots,\pi_r}(\rho_1, \ldots, \rho_r, \rho_{r+1}) \text{ queries } \mathbf{c}_j\},$$
$$\mathcal{Q}_i(T) := \{j \in |\pi_i| \mid \mathsf{V}^{\mathbf{c},\pi_1,\ldots,\pi_r}(\rho_1, \ldots, \rho_r, \rho_{r+1}) \text{ queries } \pi_{i,j}\} \text{ for all } i \in [r].$$

An important class of IOPPs is the class of non-adaptive IOPPs. Informally, an IOPP is said to be non-adaptive, if the queries that the verifier issues do not depend on previous responses. A prominent example of such a non-adaptive IOPP is the FRI IOPP [BBHR18] which we will explore in Section 4.

Definition 7 (Non-Adaptive IOPP)**.** Let $\mathcal{C}$ be a code, and let (P, V) be an IOPP for $\mathcal{C}$ with r rounds. We say that (P, V) is non-adaptive, if for any two complete transcripts $T = (\mathbf{c}, \rho_1, \pi_1, \ldots, \rho_r, \pi_r, \rho_{r+1})$ and $T' = (\mathbf{c}', \rho_1, \pi'_1, \ldots, \rho_r, \pi'_r, \rho_{r+1})$ with the same verifier challenges, we have

$$\forall i \in \{0, \ldots, r\} : \ \mathcal{Q}_i(T) = \mathcal{Q}_i(T').$$

Merkle Trees. To transform IOPPs to erasure code commitments, we use (a variant of) the BCS transformation [BCS16]. For that, we use Merkle trees [Mer88] instantiated with a random oracle[6]. To this end, let $\mathsf{H}\colon \{0,1\}^* \to \{0,1\}^\lambda$ be a random oracle. We denote by $\mathsf{Root}^{\mathsf{H}}$ the algorithm that takes as input a sequence $x_1, \ldots, x_\ell \in \Sigma$ of ℓ symbols over some alphabet Σ, and outputs the Merkle root $\mathsf{root} \in \{0,1\}^\lambda$. We denote by $\mathsf{Path}^{\mathsf{H}}$ the algorithm that takes as input $x_1, \ldots, x_\ell \in \Sigma$ as above and an index $j \in [\ell]$, and outputs an authentication path path for the jth position, i.e., for x_j. We assume that path contains the length ℓ of the underlying sequence, the position j, and the value x_i, and denote these by $\mathsf{LengthOf}^{\mathsf{H}}$, $\mathsf{PositionOf}^{\mathsf{H}}$, and $\mathsf{ValueOf}^{\mathsf{H}}$, respectively. Finally, we denote by $\mathsf{RootFromPath}^{\mathsf{H}}$ the algorithm that takes as input a path path recomputes the Merkle root. We postpone the detailed definitions to Appendix B. There, we also show that Merkle trees (in the random oracle model) are extractable. That is, a reduction can extract parts of the underlying sequence of values from a given Merkle root, such that every opening is then consistent with these parts. This is folklore and has been used (implicitly) in several works, e.g., [Val08, BCS16, HKLN20]. We make it explicit, which could be useful in other contexts as well.

3 From IOPPs to Data Availability Sampling

In this section, we present our transformation turning an (non-adaptive) IOPP into an erasure code commitment. With the transformation in [HASW23], this yields a data availability sampling scheme. For our construction to work, we require the IOPP to satisfy a certain soundness property, which we call opening-consistency.

Required Properties of IOPPs. We start with the definition of query-selection. Informally, it states that one can efficiently force the verifier to query a specific position of the codeword by deriving a corresponding random tape. In other words, one can open a specific position of the codeword.

Definition 8 (Query-Selection of IOPP)**.** Let $\mathcal{C}$ be a code, and let (P, V) be an IOPP for $\mathcal{C}$ with r rounds. We say that (P, V) is query-selectable if there is a deterministic algorithm $\mathsf{QSelect}$, such that for any partial transcript of the form $T' = (\mathbf{c}, \rho_1, \pi_1, \ldots, \rho_r, \pi_r)$ and any $j \in |\mathbf{c}|$, we have

$$j \in \mathcal{Q}_0(T' \circ \rho_{r+1}) \text{ for } \rho_{r+1} := \mathsf{QSelect}(\rho_1, \ldots, \rho_r, j).$$

As our main new notion, we introduce opening-consistency. Intuitively, we want to ensure that (1) if the verifier accepts, then the alleged codeword is within the unique decoding radius, and (2) one can never open any position for which the alleged codeword disagrees with its unique closest codeword. Here,

[6] Actually, one could use any extractable vector commitment scheme instead of a Merkle tree. We stick to Merkle trees.

by opening a position we mean providing a final randomness ρ_{r+1} such that the verifier queries this position. We also want to (3) characterize our notion in a fine-grained way where we define certain events over the verifier's randomness in individual rounds. This is essential when we want to compile the interactive IOPP to a non-interactive scheme. With this motivation in mind, let us now explain our notion of opening-consistency. First, to tackle (3), we introduce the notion of *suitable transcripts*. This notion is defined with respect to two sets of partial transcripts, namely, a *bad set* Bad and a *lucky set* Lucky. In each round in which the verifier sends a challenge ρ_i, $i \leq r$, we assume that the prover can be lucky, i.e., the partial transcript ending with ρ_i is in Lucky. However, we require that the probability that this happens is small, where the probability is only taken over the random choice of ρ_i. Further, when the prover has sent all of its messages, i.e., the partial transcript ends with π_r, then the transcript can be in a bad state, i.e., it is in Bad. In this case, we want that the probability that the verifier accepts – taken over the random choice of its final randomness ρ_{r+1} – can be upper bounded. Now, we say that a transcript is suitable, if it is not in the bad set and in no round it was in the lucky set. With this, we know that if we see an accepting transcript that came from a malicious prover interacting with an honest verifier, then with high probability it is suitable. This definition addresses (3) but is not very meaningful on its own. Namely, setting $\mathsf{Bad} = \mathsf{Lucky} = \emptyset$ would be a valid choice, and all transcripts would be suitable. We want that suitable transcripts guarantee something meaningful which we can use for proving code-binding, namely, (1) and (2). For (1), we require that any suitable transcript satisfies that the alleged codeword $\mathbf{c}$ is within the unique decoding radius, i.e., a unique closest codeword $\mathbf{c}^* \in \mathcal{C}$ exists. We highlight that this is no probabilistic statement. For (2), we require that – for a suitable transcript – it is not possible to find a randomness ρ_{r+1} for which the verifier queries a position at which $\mathbf{c}$ and $\mathbf{c}^*$ disagree. Now, setting $\mathsf{Bad} = \mathsf{Lucky} = \emptyset$ is not a valid choice anymore. Indeed, defining the right sets Bad and Lucky is one of the main challenges when proving opening-consistency for a given IOPP.

Definition 9 (Suitable Transcripts)**.** Let $\mathcal{C}$ be a code, and let (P, V) be an IOPP for $\mathcal{C}$ with r rounds. Let Bad be a set containing partial transcripts, and let Lucky be a set containing non-empty partial prover-turn transcripts. Assume that $T = (\mathbf{c}, \rho_1, \pi_1, \ldots, \rho_r, \pi_r)$ is a partial verifier-turn transcript. We say that T is a suitable transcript with respect to Bad and Lucky, if $T \notin \mathsf{Bad}$ and any prover-turn prefix T' of T satisfies $T' \notin \mathsf{Lucky}$.

Definition 10 (Opening-Consistency of IOPP)**.** Let $\mathcal{C}$ be a code, and let (P, V) be an IOPP for $\mathcal{C}$ with r rounds. Then, (P, V) is said to be opening-consistent with errors $\varepsilon_1, \varepsilon_2$ if there is are sets Bad and Lucky, such that the following properties hold:

- **No Luck.** For every $i \in [r]$, and any partial verifier-turn transcript T that contains $2i-1$ elements, we have
$$\Pr_{\rho_i}[T \circ \rho_i \in \mathsf{Lucky}] \leq \varepsilon_1.$$
- **Bad is Rejected.** Let $T = (\mathbf{c}, \rho_1, \pi_1, \ldots, \rho_r, \pi_r)$ be a transcript with $T \in \mathsf{Bad}$. Then, we have
$$\Pr_{\rho_{r+1}}[\mathsf{V}^{\mathbf{c}, \pi_1, \ldots, \pi_r}(\rho_1, \ldots, \rho_r, \rho_{r+1}) = 1] \leq \varepsilon_2.$$
- **Suitable is Close.** Let $T = (\mathbf{c}, \rho_1, \pi_1, \ldots, \rho_r, \pi_r)$ be a suitable transcript with respect to Bad and Lucky (see Definition 9). Then, $\mathbf{c}$ is within the unique decoding radius δ^* of $\mathcal{C}$, i.e., $\delta(\mathcal{C}, \mathbf{c}) \leq \delta^*$.
- **Inconsistent is Rejected.** Let $T = (\mathbf{c}, \rho_1, \pi_1, \ldots, \rho_r, \pi_r)$ be a suitable transcript with respect to Bad and Lucky (see Definition 9). Let $\mathbf{c}^* \in \mathcal{C}$ denote the unique closest codeword[7]. Let ρ_{r+1} such that there is a position $j \in \mathcal{Q}_0(T \circ \rho_{r+1})$ with $\mathbf{c}_j \neq \mathbf{c}^*_j$. For any such transcript, it holds that
$$\mathsf{V}^{\mathbf{c}, \pi_1, \ldots, \pi_r}(\rho_1, \ldots, \rho_r, \rho_{r+1}) = 0.$$

Remark 1 (Round-By-Round Soundness)*.* A notion that is related to opening-consistency is round-by-round soundness [CCH+19]. It states that there is a set of partial transcript called the *doomed set*. If the alleged codeword $\mathbf{c}$ is far from the code, then the transcript $(\mathbf{c})$ is doomed. Further, in every round the probability to leave the doomed set is small, where probability is taken over the verifier's challenge in that

[7]The unique closest codeword exists because of the *suitable is close* property.

round. Finally, a complete transcript in the doomed set is always rejected. While being useful for many applications, we have found that this notion is not sufficient in our context. Especially, round-by-round soundness gives no guarantee similar to the *inconsistent is rejected* property we demand.

Remark 2 (List Decoding). By our definition of opening-consistency, we are interested primarily in soundness when the proximity parameter is δ^*, i.e., we want to be convinced that the alleged codeword is within the unique decoding radius. On the other hand, when using IOPPs to construct arguments of knowledge, the proximity parameter can be chosen larger then δ^*, potentially leading to more efficient instantiations. This is because a knowledge extractor can rely on decoding and identify a witness for the given statement in the list. In the context of code-binding, it is essential that the adversary can not arbitrarily pick openings consistent with one of many distinct close codewords. Hence, we do not see a way to leverage a proximity parameter larger than δ^*.

Commitment Construction. Using a variant of the transformation in [BCS16], we construct an erasure-code commitment scheme $\mathsf{CC} = (\mathsf{Setup}, \mathsf{Com}, \mathsf{Open}, \mathsf{Ver})$ for a code $\mathcal{C}\colon \Gamma^k \to \Lambda^n$ from a given IOPP (P, V) for $\mathcal{C}$. We assume that the IOPP has r rounds, the length of codewords is n, and each proof string π_i of the IOPP has length ℓ_i. For convenience, we set $\ell_0 := n$. We further assume that IOPP is non-adaptive and query-selectable with algorithm $\mathsf{QSelect}$. For the construction, we use Merkle trees over a random oracle $\mathsf{H}\colon \{0,1\}^* \to \{0,1\}^\lambda$ (see Appendix B). We further use a random oracle $\mathsf{H}'\colon \{0,1\}^* \to \{0,1\}^\lambda$ and a random oracle $\hat{\mathsf{H}}\colon \{0,1\}^* \to \{0,1\}^z$, where we assume that all challenges $\rho_1, \ldots, \rho_r, \rho_{r+1}$ of the IOPP have length z. Before we describe the actual scheme CC, we define two subroutines that we will be using: the algorithm $\mathsf{OpenAuth}$ computes Merkle path for each query that the IOPP verifier issues for a given transcript; algorithm $\mathsf{CheckAuth}$ verifies that a transcript is accepting, where the proof strings in the transcript are given as Merkle roots and queries are answered using Merkle paths.

- Subroutine $\mathsf{OpenAuth}(\mathbf{c}, (\pi_i)_{i=1}^r, (\rho_i)_{i=1}^r, \rho_{r+1}) \to \mathsf{auth}$:
  1. Set $T := (\mathbf{c}, \rho_1, \pi_1, \ldots, \rho_r, \pi_r, \rho_{r+1})$.
  2. For each query $j \in \mathcal{Q}_0(T)$, set $\mathsf{path}_{0,j} := \mathsf{Path}^{\mathsf{H}}(\mathbf{c}, j)$.
  3. For each $i \in [r]$ and each query $j \in \mathcal{Q}_i(T)$, set $\mathsf{path}_{i,j} := \mathsf{Path}^{\mathsf{H}}(\pi_i, j)$.
  4. Set and return $\mathsf{auth} := ((\mathsf{path}_{i,j})_{j \in \mathcal{Q}_i})_{i=0}^r$.
- Subroutine $\mathsf{CheckAuth}((\mathsf{root}_i)_{i=0}^r, (\rho_i)_{i=1}^r, \rho_{r+1}, \mathsf{auth}) \to b$:
  1. Parse $\mathsf{auth} = ((\mathsf{path}_{i,j})_{j \in \mathcal{Q}_i})_{i=0}^r$.
  2. If there is an $i \in \{0, \ldots, r\}$ and a $j \in \mathcal{Q}_i$ such that $\mathsf{PositionOf}^{\mathsf{H}}(\mathsf{path}_{i,j}) \neq j$, return $b := 0$.
  3. If there is an $i \in \{0, \ldots, r\}$ and a $j \in \mathcal{Q}_i$ such that $\mathsf{LengthOf}^{\mathsf{H}}(\mathsf{path}_{i,j}) \neq \ell_i$, return $b := 0$.
  4. If there is an $i \in \{0, \ldots, r\}$ and a $j \in \mathcal{Q}_i$ such that $\mathsf{RootFromPath}^{\mathsf{H}}(\mathsf{path}_{i,j}) \neq \mathsf{root}_i$, return $b := 0$.
  5. Run $b_{\mathsf{V}} := \mathsf{V}^{\mathbf{c}, \pi_1, \ldots, \pi_r}(\rho_1, \ldots, \rho_r, \rho_{r+1})$ not knowing $\mathbf{c}$ and π_i. Answer queries as follows:
     - If V queries a position of $\mathbf{c}$ which is not in $\mathcal{Q}_0$, return 0.
     - Answer any query of position j in $\mathbf{c}$ with $\mathsf{ValueOf}^{\mathsf{H}}(\mathsf{path}_{0,j})$.
     - If V queries a position of π_i for some i which is not in $\mathcal{Q}_i$, return 0.
     - Answer any query of position j in π_i with $\mathsf{ValueOf}^{\mathsf{H}}(\mathsf{path}_{i,j})$.
  6. If there is a $j \in \mathcal{Q}_0$ such that V never queried position j of $\mathbf{c}$, return 0.
  7. If there is a $j \in \mathcal{Q}_i$ for some i such that V never queries position j of π_i, return 0.
  8. Return $b := b_{\mathsf{V}}$.

Let $L \in \mathbb{N}$ be a repetition parameter. Intuitively, we can think of L as being (proportial) to the number of queries that the verifier of the IOPP makes in total. With that, the erasure code commitment $\mathsf{CC} = (\mathsf{Setup}, \mathsf{Com}, \mathsf{Open}, \mathsf{Ver})$ is given as follows:

- $\mathsf{Setup}(1^\lambda) \to \mathsf{ck}$: Return $\mathsf{ck} := \bot$.

- $\mathsf{Com}(\mathsf{ck}, m) \to (\mathsf{com}, St)$:
  1. Compute $\mathbf{c} := \mathcal{C}(m)$ and $\mathsf{root}_0 := \mathsf{Root}^{\mathsf{H}}(\mathbf{c})$.
  2. Set $\mathsf{hst}_{-1} := 1^\lambda$, $\mathsf{hst}_0 := \mathsf{H}'(\mathsf{root}_0, \mathsf{hst}_{-1}, 0)$ and for each $i \in [r]$, do the following:
     (a) Derive $\rho_i := \hat{\mathsf{H}}(\mathsf{hst}_{i-1})$.
     (b) Compute $\pi_i := \mathsf{P}(\mathbf{c}, \rho_1, \ldots, \rho_i)$.
     (c) Commit to π_i using a Merkle tree, i.e., $\mathsf{root}_i := \mathsf{Root}^{\mathsf{H}}(\pi_i)$.
     (d) Update the state $\mathsf{hst}_i := \mathsf{H}'(\mathsf{root}_i, \mathsf{hst}_{i-1}, i)$.
  3. For $l \in [L]$ do the following:
     (a) Derive $\rho_{r+1}^{(l)} := \hat{\mathsf{H}}(\mathsf{hst}_r, l)$.
     (b) Run $\mathsf{auth}^{(l)} := \mathsf{OpenAuth}(\mathbf{c}, (\pi_i)_{i=1}^r, (\rho_i)_{i=1}^r, \rho_{r+1}^{(l)})$.
  4. Set $\mathsf{com} := \left((\mathsf{root}_i)_{i=0}^r, (\mathsf{auth}^{(l)})_{l=1}^L\right)$ and $St := (\mathbf{c}, (\pi_i)_{i=1}^r, (\rho_i)_{i=1}^r)$.
- $\mathsf{Open}(\mathsf{ck}, St, j) \to \tau$:
  1. Parse $St = (\mathbf{c}, (\pi_i)_{i=1}^r, (\rho_i)_{i=1}^r)$.
  2. Derive $\rho_{r+1} := \mathsf{QSelect}(\rho_1, \ldots, \rho_r, j)$.
  3. Run $\tau := \mathsf{auth} := \mathsf{OpenAuth}(\mathbf{c}, (\pi_i)_{i=1}^r, (\rho_i)_{i=1}^r, \rho_{r+1})$.
- $\mathsf{Ver}(\mathsf{ck}, \mathsf{com}, j, c, \tau) \to b$:
  1. Parse $\mathsf{com} = \left((\mathsf{root}_i)_{i=0}^r, (\mathsf{auth}^{(l)})_{l=1}^L\right)$ and $\tau = \mathsf{auth}$.
  2. Set $\mathsf{hst}_{-1} := 1^\lambda$, $\mathsf{hst}_0 := \mathsf{H}'(\mathsf{root}_0, \mathsf{hst}_{-1}, 0)$ and for each $i \in [r]$, do the following:
     (a) Derive $\rho_i := \hat{\mathsf{H}}(\mathsf{hst}_{i-1})$.
     (b) Update the state $\mathsf{hst}_i := \mathsf{H}'(\mathsf{root}_i, \mathsf{hst}_{i-1}, i)$.
  3. For $l \in [L]$ do the following:
     (a) Set $\rho_{r+1}^{(l)} := \hat{\mathsf{H}}(\mathsf{hst}_r, l)$.
     (b) If $\mathsf{CheckAuth}((\mathsf{root}_i)_{i=0}^r, (\rho_i)_{i=1}^r, \rho_{r+1}^{(l)}, \mathsf{auth}^{(l)}) = 0$, return $b := 0$.
  4. Set $\rho_{r+1} := \mathsf{QSelect}(\rho_1, \ldots, \rho_r, j)$.
  5. If $\mathsf{CheckAuth}((\mathsf{root}_i)_{i=0}^r, (\rho_i)_{i=1}^r, \rho_{r+1}, \mathsf{auth}) = 0$, return $b := 0$.
  6. Parse $\mathsf{auth} = ((\mathsf{path}_{i,j'})_{j' \in \mathcal{Q}_i})_{i=0}^r$.
  7. If $j \notin \mathcal{Q}_0$ or $c \neq \mathsf{ValueOf}^{\mathsf{H}}(\mathsf{path}_{0,j})$, return $b := 0$. Otherwise, return $b := 1$.

Completeness of the scheme can easily be verified.

Security. We show position-binding and code-binding. Intuitively, position-binding follows directly from position-binding of Merkle trees and the query-selection property of the IOPP.

Lemma 1. *Let* $\mathsf{H}\colon \{0,1\}^* \to \{0,1\}^\lambda$ *be a random oracle. Then, for every algorithm* $\mathcal{A}$ *that makes at most* Q_{H} *queries to random oracle* H*, we have*

$$\mathsf{Adv}^{\mathsf{pos\text{-}bind}}_{\mathcal{A},\mathsf{CC}}(\lambda) \leq \frac{Q_{\mathsf{H}}^2}{2^\lambda}.$$

Proof. Let $\mathcal{A}$ be an adversary against position-binding of CC. By definition of position-binding, $\mathcal{A}$ gets as input a commitment key $\mathsf{ck} = \bot$ of CC and ouptuts a commitment $\mathsf{com} = \left((\mathsf{root}_i)_{i=0}^r, (\mathsf{auth}^{(l)})_{l=1}^L\right)$, a position $j \in [n]$, and two alleged openings $c, \tau = \mathsf{auth}$ and $c', \tau' = \mathsf{auth}'$ for position j. If $\mathcal{A}$ breaks position-binding, then $c \neq c'$ and $\mathsf{Ver}(\mathsf{ck}, \mathsf{com}, j, c, \tau) = 1$ and $\mathsf{Ver}(\mathsf{ck}, \mathsf{com}, j, c', \tau') = 1$. We will argue that this implies a collision in the random oracle H used to define Merkle trees, which occurs with probability at most $Q_{\mathsf{H}}^2/2^\lambda$. To this end, recall that both invocations of Ver first compute the same $\rho_1, \ldots, \rho_r$

and $\rho_{r+1}^{(l)}$ (for $l \in [L]$) from com by invoking $\hat{\mathsf{H}}$ properly. Then, both invocations compute the same $\rho_{r+1} = \mathsf{QSelect}(\rho_1, \ldots, \rho_r, j)$. Write $\mathsf{auth} = ((\mathsf{path}_{i,j'})_{j' \in \mathcal{Q}_i})_{i=0}^r$ and $\mathsf{auth}' = ((\mathsf{path}'_{i,j'})_{j' \in \mathcal{Q}'_i})_{i=0}^r$. Then, we know that $\mathsf{CheckAuth}((\mathsf{root}_i)_{i=0}^r, (\rho_i)_{i=1}^r, \rho_{r+1}, \mathsf{auth}) = 1$ and $\mathsf{CheckAuth}((\mathsf{root}_i)_{i=0}^r, (\rho_i)_{i=1}^r, \rho_{r+1}, \mathsf{auth}') = 1$. In particular, in both of these invocations of $\mathsf{CheckAuth}$, V is run on input $\rho_1, \ldots, \rho_{r+1}$, and by the query-selection property of the IOPP, we know that it must query the jth position of its first oracle, which is answered by $\mathsf{ValueOf}^{\mathsf{H}}(\mathsf{path}_{0,j})$ and $\mathsf{ValueOf}^{\mathsf{H}}(\mathsf{path}'_{0,j})$, respectively. As Ver checks that $c = \mathsf{ValueOf}^{\mathsf{H}}(\mathsf{path}_{0,j})$ (resp. $c' = \mathsf{ValueOf}^{\mathsf{H}}(\mathsf{path}'_{0,j})$), we know that

$$\mathsf{ValueOf}^{\mathsf{H}}(\mathsf{path}_{0,j}) = c \neq c' = \mathsf{ValueOf}^{\mathsf{H}}(\mathsf{path}'_{0,j}).$$

By definition of $\mathsf{CheckAuth}$, we also know that

$$\mathsf{RootFromPath}^{\mathsf{H}}(\mathsf{path}_{0,j}) = \mathsf{root}_0 = \mathsf{RootFromPath}^{\mathsf{H}}(\mathsf{path}'_{0,j}).$$

Thus, we have Merkle paths for different openings but the same root, which implies a collision for H. □

Theorem 2. *Let $\mathcal{C}$ be a code with unique decoding radius $\delta^* \in [0,1]$. Assume that (P, V) is a non-adaptive query-selectable IOPP for $\mathcal{C}$, and assume that it is opening-consistent with errors $\varepsilon_1, \varepsilon_2$. Let $\mathsf{H}\colon \{0,1\}^* \to \{0,1\}^\lambda, \mathsf{H}'\colon \{0,1\}^* \to \{0,1\}^\lambda$, and $\hat{\mathsf{H}}\colon \{0,1\}^* \to \{0,1\}^z$ be random oracles.*

Then, for every algorithm $\mathcal{A}$ that makes at most $Q_{\mathsf{H}}, Q_{\mathsf{H}'}, Q_{\hat{\mathsf{H}}}$ queries to random oracles $\mathsf{H}, \mathsf{H}', \hat{\mathsf{H}}$, respectively, we have

$$\mathsf{Adv}^{\mathsf{code\text{-}bind}}_{\mathcal{A},\mathsf{CC}}(\lambda) \leq \frac{2Q_{\mathsf{H}'}^2 + Q_{\mathsf{H}'}Q_{\hat{\mathsf{H}}} + 2Q_{\mathsf{H}}^2 + Q_{\mathsf{H}}Q_{\mathsf{H}'}}{2^\lambda} + Q_{\mathsf{H}'} \max\{\varepsilon_1, \varepsilon_2^L\}.$$

Proof Strategy. In the code-binding game, adversary $\mathcal{A}$ has access to random oracles $\mathsf{H}, \mathsf{H}', \hat{\mathsf{H}}$ which are simulated by the game lazily via maps $h[\cdot], h'[\cdot], \hat{h}[\cdot]$, respectively. The adversary outputs a commitment $\mathsf{com} = \left((\mathsf{root}_i)_{i=0}^r, (\mathsf{auth}^{(l)})_{l=1}^L\right)$, a set $J \subseteq [n]$ of indices, and a list $(c_j, \tau_j)_{j \in J}$ of alleged openings at positions $j \in J$. It breaks code-binding if all openings verify, i.e., $\mathsf{Ver}(\mathsf{ck}, \mathsf{com}, j, c_j, \tau_j) = 1$ for all $j \in J$, and there is no codeword $\mathbf{c} \in \mathcal{C}$ such that $\mathbf{c}$ is consistent with all c, i.e., for any $\mathbf{c} \in \mathcal{C}$ there is a $j \in J$ with $\mathbf{c}_j \neq c_j$.

To argue that this can not happen, we want to use opening-consistency of the underlying IOPP. However, opening-consistency defines properties of transcripts of the IOPP, whereas the commitment construction CC only implicitly deals with transcripts. Abstractly, a Merkle root root and a state hst that are submitted to random oracle H' are meant to represent a transcript. Thus, our first step (games $\mathbf{G}_0$ to $\mathbf{G}_3$) will be to extract such transcripts from triples $(\mathsf{root}, \mathsf{hst}, i)$ submitted to H'. To be a bit more precise, we use Merkle tree extraction (Appendix B, Lemma 13) to map such a triple to a partial verifier-turn transcript of the form $(\mathbf{c}, \rho_1, \pi_1, \ldots, \rho_i, \pi_i)$. Then, we argue that all Merkle paths contained in $\mathcal{A}$'s output are consistent with the extracted transcripts.

In the second step of the proof, we want to apply opening-consistency. Namely, consider the case where a transcript represented by $(\mathsf{root}, \mathsf{hst}, i)$ is extended with a verifier challenge: this happens when $\hat{\mathsf{H}}(\mathsf{hst}')$ is queried for $\mathsf{hst}' = \mathsf{H}'(\mathsf{root}, \mathsf{hst}, i)$. Having ensured that these two queries occur in order, opening-consistency (*no luck* property) tells us that, except with probability ε_1, the extended transcript is not in the lucky set. So, if ε_1 is sufficiently small, we can assume that no partial transcript induced by the adversary's commitment is in the lucky set. Additionally, by opening-consistency (*bad is rejected* property), we can assume, except with probability ε_2^L, that the almost complete transcript induced by the adversary's commitment is not in the bad set. These two steps are done in $\mathbf{G}_4$.

In our final step, we apply the *inconsistent is rejected* property of opening-consistency. For that, note that every opening c_j, τ_j submitted by $\mathcal{A}$ defines a transcript, precisely, a completion of T. Further, at least one of the openings j^* has to be inconsistent with the unique closest codeword of $\mathbf{c}$. We then use the *inconsistent is rejected* property of opening-consistency to argue that this opening must be rejected. We continue with the formal proof.

Proof. Let $\mathcal{A}$ be an adversary against code-binding of CC. We now formally prove the statement by presenting a sequence of games.

Game $\mathbf{G}_0$: Game $\mathbf{G}_0$ is the code-binding game of CC for adversary $\mathcal{A}$, as recalled in the proof strategy above. By definition, we have

$$\mathsf{Adv}^{\mathsf{code\text{-}bind}}_{\mathcal{A},\mathsf{CC}}(\lambda) = \Pr\left[\mathbf{G}_0 \Rightarrow 1\right].$$

Game $\mathbf{G}_1$: In this game, we introduce some bad events related to random oracle queries and let the game abort if any of these events occurs. The events are defined as follows:

- Event Coll: This event occurs, if $\mathsf{H}'(\mathsf{root},\mathsf{hst},i) = \mathsf{H}'(\mathsf{root}',\mathsf{hst}',i')$ for triples $(\mathsf{root},\mathsf{hst},i) \neq (\mathsf{root}',\mathsf{hst}',i')$ with $\mathsf{root},\mathsf{root}' \in \{0,1\}^\lambda, \mathsf{hst},\mathsf{hst}' \in \{0,1\}^\lambda, i,i' \in \mathbb{N}_0$.
- Event Chain: This event occurs, if $\mathsf{H}'(\mathsf{root},\mathsf{hst},i)$ is queried for some $\mathsf{root} \in \{0,1\}^\lambda, \mathsf{hst} \in \{0,1\}^\lambda$, $i \in \mathbb{N}_0$ for the first time, i.e., the hash value $\mathsf{hst}' = h'[\mathsf{root},\mathsf{hst},i]$ is sampled uniformly at random from $\{0,1\}^\lambda$, and at that time, we have $h'[\mathsf{root}',\mathsf{hst}',i'] \neq \bot$ for some $\mathsf{root}' \in \{0,1\}^\lambda, i' \in \mathbb{N}_0$ or we have $\mathsf{hst}' = \mathsf{hst}$.
- Event Cross: This event occurs, if $\mathsf{H}'(\mathsf{root},\mathsf{hst},i)$ is queried for some $\mathsf{root} \in \{0,1\}^\lambda, \mathsf{hst} \in \{0,1\}^\lambda$, $i \in \mathbb{N}_0$ for the first time, i.e., the hash value $\mathsf{hst}' = h'[\mathsf{root},\mathsf{hst},i]$ is sampled uniformly at random from $\{0,1\}^\lambda$, and at that time, we have $\hat{h}[\mathsf{hst}'] \neq \bot$ or $\hat{h}[\mathsf{hst}',l] \neq \bot$ for some $l \in [L]$.

We can easily bound the probability of each Chain and Coll using a union bound of all pairs of queries to H'. Namely, the probability of Chain (resp. Coll) is at most $Q^2_{\mathsf{H}'}/2^\lambda$. Similarly, the probability of Cross is at most $Q_{\mathsf{H}'}Q_{\hat{\mathsf{H}}}/2^\lambda$. Also, games $\mathbf{G}_0$ and $\mathbf{G}_1$ are identical unless one of the events occurs. In combination, we get

$$\left|\Pr\left[\mathbf{G}_0 \Rightarrow 1\right] - \Pr\left[\mathbf{G}_1 \Rightarrow 1\right]\right| \leq \frac{2Q^2_{\mathsf{H}'} + Q_{\mathsf{H}'}Q_{\hat{\mathsf{H}}}}{2^\lambda}.$$

Game $\mathbf{G}_2$: In this game, we introduce a map $\mathsf{Tra}[\cdot]$ that maps triples of the form $(\mathsf{root},\mathsf{hst},i) \in \{0,1\}^\lambda \times \{0,1\}^\lambda \times \mathbb{N}_0$ to partial verifier-turn transcripts containing an alleged codeword, i proof strings, and i challenges. We will then make some observations about this map.

Before reading how the map is defined, the reader may recall algorithm $\mathsf{ExtLeafs}$ for extracting leafs from a Merkle root (see Appendix B, Figure 2). Informally, this algorithm takes as input $\mathsf{root} \in \{0,1\}^\lambda$ and an integer $\ell \in \mathbb{N}$ and extracts ℓ leafs $\mathsf{leaf}_1,\ldots,\mathsf{leaf}_\ell$ represented by root. Some leafs may not be extractable, which is indicated by $\mathsf{leaf}_j = \bot$.

Now, we define how map Tra is populated throughout the game. Initially, the map $\mathsf{Tra}[\cdot]$ is empty, i.e., $\mathsf{Tra}[\mathsf{root},\mathsf{hst},i]$ for all triples $(\mathsf{root},\mathsf{hst},i) \in \{0,1\}^\lambda \times \{0,1\}^\lambda \times \mathbb{N}_0$. On a random oracle query $\mathsf{H}'(\mathsf{root},\mathsf{hst},i)$, we distinguish three cases:

- If the hash value is already defined, i.e., $h'[\mathsf{root},\mathsf{hst},i] \neq \bot$, then Tra is not changed. Otherwise, we distinguish the cases $i = 0$ and $i > 0$.
- In the case $i = 0$, the game runs $(\mathsf{leaf}_1,\ldots,\mathsf{leaf}_{\ell_0}) := \mathsf{ExtLeafs}(\mathsf{root},\ell_0)$. It then defines the alleged codeword $\mathbf{c}$ by

  $$\forall j \in [\ell_0] = [n] : \ \mathbf{c}_j := \begin{cases} \mathsf{leaf}_j & \text{if } \mathsf{leaf}_j \neq \bot \\ \gamma_\bot & \text{if } \mathsf{leaf}_j = \bot \end{cases},$$

  where we assume that all leaf_j are interpreted as symbols in Λ and $\gamma_\bot \in \Lambda$ is an arbitrary but fixed symbol in Λ. The game sets $\mathsf{Tra}[\mathsf{root},\mathsf{hst},i] := (\mathbf{c})$.
- In the case $i > 0$, the game runs $(\mathsf{leaf}_1,\ldots,\mathsf{leaf}_{\ell_i}) := \mathsf{ExtLeafs}(\mathsf{root},\ell_i)$. It then defines the proof string π_i by

  $$\forall j \in [\ell_i] : \ \pi_{i,j} := \begin{cases} \mathsf{leaf}_j & \text{if } \mathsf{leaf}_j \neq \bot \\ \tau_\bot & \text{if } \mathsf{leaf}_j = \bot \end{cases},$$

  where we assume that all leaf_j are interpreted as symbols of the alphabet of the ith proof string, and $\tau_\bot$ is an arbitrary but fixed symbol in that alphabet. Next, if there is no pair $(\mathsf{root}',\mathsf{hst}') \in \{0,1\}^\lambda \times \{0,1\}^\lambda$ such that $h[\mathsf{root}',\mathsf{hst}',i-1] = \mathsf{hst}$ and $\mathsf{Tra}[\mathsf{root}',\mathsf{hst}',i-1] \neq \bot$, the game does not change Tra. Otherwise, there can only be one such pair (see event Coll). Then, the game sets

  $$\mathsf{Tra}[\mathsf{root},\mathsf{hst},i] := \mathsf{Tra}[\mathsf{root}',\mathsf{hst}',i-1] \circ \rho_i \circ \pi_i \text{ for } \rho_i := \hat{\mathsf{H}}(\mathsf{hst}').$$

With this definition of the map $\mathsf{Tra}[\cdot]$, we make the following observations:

- When $\mathcal{A}$ terminates and outputs $\mathsf{com} = \left((\mathsf{root}_i)_{i=0}^r, (\mathsf{auth}^{(l)})_{l=1}^L\right)$, this commitment specifies partial verifier-turn transcripts as follows: recall that algorithm Ver sets $\mathsf{hst}_{-1} := 1^\lambda$, $\mathsf{hst}_0 := \mathsf{H}'(\mathsf{root}_0, \mathsf{hst}_{-1}, 0)$ and for each $i \in [r]$, it sets $\mathsf{hst}_i := \mathsf{H}'(\mathsf{root}_i, \mathsf{hst}_{i-1}, i)$. Then, for each $i \in \{0\} \cup [r]$, we can define the partial verifier-turn transcript $T_i^* := \mathsf{Tra}[\mathsf{root}_i, \mathsf{hst}_{i-1}, i]$.
- All T_i^* are defined, i.e., $T_i^* \neq \perp$. This can be seen inductively using that Chain does not occur.
- The transcripts are consistent, namely, T_i^* is a prefix of $T_{i'}^*$ for any $i < i'$. This is by definition and using that Coll does not occur.

It is clear that our changes do not change the view of $\mathcal{A}$ and do not affect the output of the game. Thus, we have

$$\Pr[\mathbf{G}_1 \Rightarrow 1] = \Pr[\mathbf{G}_2 \Rightarrow 1].$$

Game $\mathbf{G}_3$: In this game, we ensure that whatever the adversary opens with a Merkle path is consistent with the leafs extracted in $\mathbf{G}_2$. More precisely, we define the following bad event and let the game abort if it occurs:

- Event $\mathsf{BreakMerkle}$: This event occurs, if any of the Merkle paths that $\mathcal{A}$ submits is not consistent with the transcript T_r^*. More precisely, write $T_r^* = (\pi_0 := \mathbf{c}, \rho_1, \pi_1, \ldots, \rho_r, \pi_r)$ and recall that when $\mathcal{A}$ terminates, it outputs a commitment $\mathsf{com} = \left((\mathsf{root}_i)_{i=0}^r, (\mathsf{auth}^{(l)})_{l=1}^L\right)$ and a list $(c_j, \tau_j)_{j \in J}$. We let P be the set of pairs (i, path) such that path is contained in an $\mathsf{auth}^{(l)}$ for some $l \in [L]$ or in an τ_j for some $j \in J$ for the ith proof string (or for the alleged codeword meaning $i = 0$). Then, the event occurs if there is a pair $(i, \mathsf{path}) \in P$ such that $\mathsf{ValueOf}^{\mathsf{H}}(\mathsf{path}) \neq \pi_{i,k}$, where $k := \mathsf{PositionOf}^{\mathsf{H}}(\mathsf{path})$.

We can bound the probability of event $\mathsf{BreakMerkle}$ by giving a reduction that runs in the game in Lemma 13. We sketch the reduction. It is run with oracle access to H, an oracle $\mathsf{SubRoot}$, and an oracle $\mathsf{SubPath}$. It simulates $\mathbf{G}_2$ for $\mathcal{A}$ by forwarding the access to H and providing oracles H' and $\hat{\mathsf{H}}'$ as in $\mathbf{G}_2$. Note that the reduction can internally keep a copy of h by proper bookkeeping, and therefore it can run $\mathsf{ExtLeafs}$ as in $\mathbf{G}_2$ (alternatively, the reduction could simulate $\mathbf{G}_1$ instead, which would also be fine). Further, on every random oracle query $\mathsf{H}'(\mathsf{root}, \mathsf{hst}, i)$, the reduction calls $\mathsf{SubRoot}(\mathsf{root}, \ell_i)$. Then, once $\mathcal{A}$ terminates and outputs a commitment $\mathsf{com} = \left((\mathsf{root}_i)_{i=0}^r, (\mathsf{auth}^{(l)})_{l=1}^L\right)$ and a list $(c_j, \tau_j)_{j \in J}$, the reduction constructs the set P as in the definition of $\mathsf{BreakMerkle}$ and calls $\mathsf{SubPath}(\mathsf{root}_i, \mathsf{path})$ for each pair $(i, \mathsf{path}) \in P$. We see that the reduction makes at most Q_{H} many queries to H and at most $Q_{\mathsf{H}'}$ many queries to $\mathsf{SubRoot}$. It simulates game $\mathbf{G}_2$ perfectly for $\mathcal{A}$. Further, if $\mathsf{BreakMerkle}$ occurs and the game outputs 1, then $\mathsf{bad} = 1$ in the game in Lemma 13, where we use the definition of $\mathsf{CheckAuth}$. With that, Lemma 13 yields

$$|\Pr[\mathbf{G}_2 \Rightarrow 1] - \Pr[\mathbf{G}_3 \Rightarrow 1]| \leq \frac{2Q_{\mathsf{H}}^2 + Q_{\mathsf{H}} Q_{\mathsf{H}'}}{2^\lambda}.$$

Game $\mathbf{G}_4$: In this game, we introduce two more bad events $\mathsf{EvLucky}$ and $\mathsf{EvBadAcc}$, which may occur on queries to random oracle H', and let the game abort whenever they occur. For that, let Bad and Lucky be the sets from the definition of opening-consistency (Definition 10). The bad event $\mathsf{EvLucky}$ occurs if a partial verifier-turn transcript (represented by a triple $(\mathsf{root}, \mathsf{hst}, i)$) is queried to H' and its extension is in the lucky set Lucky. The bad event $\mathsf{EvBadAcc}$ occurs if all L extensions of an almost complete transcript in Bad are accepting. More precisely:

- Event $\mathsf{EvLucky}$: This event occurs, if there is an entry $T = \mathsf{Tra}[\mathsf{root}, \mathsf{hst}, i]$ in the map Tra such that $0 \leq i < r$ and $T \circ \rho_{i+1} \in \mathsf{Lucky}$, where $\mathsf{hst}' = \mathsf{H}'(\mathsf{root}, \mathsf{hst}, i)$ and $\rho_{i+1} = \hat{\mathsf{H}}(\mathsf{hst}')$.
- Event $\mathsf{EvBadAcc}$: This event occurs, if there is an entry $T = \mathsf{Tra}[\mathsf{root}, \mathsf{hst}, r]$ in the map Tra such that $T \in \mathsf{Bad}$ and for all $l \in [L]$, we have

$$\mathsf{V}^{\mathbf{c}, \pi_1, \ldots, \pi_r}(\rho_1, \ldots, \rho_r, \rho_{r+1}^{(l)}) = 1,$$

  where $T = (\mathbf{c}, \rho_1, \pi_1, \ldots, \rho_r, \pi_r)$, $\mathsf{hst}' = \mathsf{H}'(\mathsf{root}, \mathsf{hst}, r)$ and $\rho_{r+1}^{(l)} = \hat{\mathsf{H}}(\mathsf{hst}', l)$.

First, fix i with $0 \leq i < r$ and consider a fixed query $(\mathsf{root}, \mathsf{hst}, i)$ to random oracle H', adding an entry to the map Tra. As we assume that Cross does not occur, the transcript $T = \mathsf{Tra}[\mathsf{root}, \mathsf{hst}, i]$ is fixed before ρ_{i+1} is sampled at random. So, it follows from the *no luck* property of opening-consistency that the probability of $\mathsf{EvLucky}$ for this fixed query is at most ε_1. Similarly, for a fixed query to H', we can use the *bad is rejected* property of opening-consistency to bound the probability for event $\mathsf{EvBadAcc}$ in a fixed random oracle query by ε_2. The only difference is that the bad event has to occur for *all* $l \in [L]$. As the $\rho_{r+1}^{(l)}$ are all sampled independently, the probability of $\mathsf{EvBadAcc}$ in that fixed query is at most ε_2^L. With a union bound over all queries, we get

$$|\Pr[\mathbf{G}_3 \Rightarrow 1] - \Pr[\mathbf{G}_4 \Rightarrow 1]| \leq \Pr[\mathsf{Leave}] \leq Q_{\mathsf{H}'} \max\{\varepsilon_1, \varepsilon_2^L\}.$$

In the final part of the proof, we bound the probability that $\mathbf{G}_4$ outputs 1. For that, first recall that δ^* denotes the unique decoding radius of $\mathcal{C}$. Further, recall the notation introduced in $\mathbf{G}_2$. Namely, $T_i^* = \mathsf{Tra}[\mathsf{root}_i, \mathsf{hst}_{i-1}, i]$ for all $i \in \{0\} \cup [r]$, where the commitment output by $\mathcal{A}$ is $\mathsf{com} = \left((\mathsf{root}_i)_{i=0}^r, (\mathsf{auth}^{(l)})_{l=1}^L\right)$. We have also established that all T_i^* are prefixes of each other and so we can write them as

$$T_i^* = (\pi_0 = \mathbf{c}, \rho_1, \pi_1, \ldots, \rho_i, \pi_i) \text{ for } i \in \{0\} \cup [r].$$

Also, recall that $\mathcal{A}$ not only submits the commitment com, but also a list $(c_j, \tau_j)_{j \in J}$ of alleged openings at positions $j \in J$, where each $j \in J$ defines a final verifier randomness $\rho_{j,r+1} = \mathsf{QSelect}(\rho_1, \ldots, \rho_r, j)$ as in algorithm Ver.

First Claim. If game $\mathbf{G}_4$ outputs 1, then T_r^* is suitable with respect to Bad and Lucky (see Definition 9).

Proof of First Claim. To prove the claim, we need to argue that $T_r^* \notin \mathsf{Bad}$ and that no prover-turn prefix of T_r^* is in Lucky. First, because $\mathsf{EvLucky}$ does not occur (see $\mathbf{G}_4$), the latter claim holds. It remains to argue that $T_r^* \notin \mathsf{Bad}$. For that, assume towards contradiction that $T_r^* \in \mathsf{Bad}$ and $\mathbf{G}_4$ outputs 1. Recall that algorithm Ver defines L final verifier challenges $\rho_{r+1}^{(l)}$ by $\rho_{r+1}^{(l)} = \hat{\mathsf{H}}(\mathsf{hst}_r, l)$ for $l \in [L]$. As $\mathsf{EvBadAcc}$ does not occur, we know that there is an $l^* \in [L]$ such that the complete transcript $T_r^* \circ \rho_{r+1}^{(l^*)}$ is rejected, i.e.,

$$\mathsf{V}^{\mathbf{c}, \pi_1, \ldots, \pi_r}(\rho_1, \ldots, \rho_r, \rho_{r+1}^{(l^*)}) = 0.$$

We claim that algorithm $\mathsf{CheckAuth}$ as invoked in Ver with input $((\mathsf{root}_i)_{i=0}^r, (\rho_i)_{i=1}^r, \rho_{r+1}^{(l^*)}, \mathsf{auth}^{(l^*)})$ outputs 0. For that, note that $\mathsf{CheckAuth}$ internally runs V with explicit input $(\rho_1, \ldots, \rho_r, \rho_{r+1}^{(l^*)})$ and with access to oracles simulated on the fly using the Merkle openings in $\mathsf{auth}^{(l^*)}$. As we assume IOPP to be non-adaptive, the queries that V issues are fixed given $(\rho_1, \ldots, \rho_r, \rho_{r+1}^{(l^*)})$, and because event $\mathsf{BreakMerkle}$ (see $\mathbf{G}_3$) does not occur, we know that the queries are answered exactly as if we run the verifier with oracles $\mathbf{c}, \pi_1, \ldots, \pi_r$. Therefore, $\mathsf{CheckAuth}$ returns 0, meaning that Ver returns 0, meaning that $\mathbf{G}_4$ does not output 1, a contradiction.

Second Claim. Assume that game $\mathbf{G}_4$ outputs 1 and let $\mathbf{c}^* \in \mathcal{C}$ denote the unique closest codeword to $\mathbf{c}$, which exists due to the previous claim and the *suitable is close* property of opening-consistency. Then, there is a $j^* \in J$ such that $\mathbf{c}_{j^*} \neq \mathbf{c}_{j^*}^*$ and $j^* \in \mathcal{Q}_0(T_r^* \circ \rho_{j^*,r+1})$.

Proof of Second Claim. Assume that game $\mathbf{G}_4$ outputs 1. Recall that $\mathcal{A}$ outputs a list $(c_j, \tau_j)_{j \in J}$ of alleged openings at positions $j \in J$. There is at least one $j^* \in J$ such that $c_{j^*} \neq \mathbf{c}_{j^*}^*$, as otherwise, $\mathbf{c}^*$ would be consistent with the c_j and $\mathcal{A}$ would not break code-binding, i.e., $\mathbf{G}_4$ would not output 1. Further, recall that Ver internally parses $\tau_{j^*} = \mathsf{auth} = ((\mathsf{path}_{i,j'})_{j' \in \mathcal{Q}_i})_{i=0}^r$ and asserts that $j^* \in \mathcal{Q}_0$. Also, observe that algorithm $\mathsf{CheckAuth}$ called in Ver ensures that $\mathcal{Q}_0 = \mathcal{Q}_0(T_r^* \circ \rho_{j^*,r+1})$, and thus $j^* \in \mathcal{Q}_0(T_r^* \circ \rho_{j^*,r+1})$, which is what we wanted to show.

Finishing the Proof. We conclude by applying the *inconsistent is rejected* property of opening-consistency: we first assume towards contradiction that $\mathbf{G}_4$ outputs 1. Then, our claims show that the transcript T_r^* is a suitable transcript and that there is $j^* \in J$ such that $\mathbf{c}_{j^*} \neq \mathbf{c}_{j^*}^*$ and $j^* \in \mathcal{Q}_0(T_r^* \circ \rho_{j^*,r+1})$. The *inconsistent is rejected* property of opening-consistency now states that

$$\mathsf{V}^{\mathbf{c}, \pi_1, \ldots, \pi_r}(\rho_1, \ldots, \rho_r, \rho_{j^*,r+1}) = 0.$$

Using that $\mathsf{BreakMerkle}$ does not occur, we see that $\mathsf{Ver}(\mathsf{ck}, \mathsf{com}, j^*, c_{j^*}, \tau_{j^*}) = 0$, similar to how we argued in the proof of our first claim. This means that $\mathbf{G}_4$ does not output 1, a contradiction. We get

$$\Pr[\mathbf{G}_4 \Rightarrow 1] = 0.$$

□

4 Instantiation from FRI

In this section, we instantiate our transformation from Section 3 with the FRI IOPP [BBHR18]. For that, we first recall FRI, using notation inspired by [BBHR18] and [BGK+23]. It will be clear that it is non-adaptive (see Definition 7) and query-selectable (see Definition 8). To apply the transformation from Section 3, we need to show opening-consistency (see Definition 10).

4.1 Construction

Before we formally specify our construction, we recall background on Reed-Solomon codes and give an informal overview.

Reed-Solomon Codes. Before recalling FRI, we define the codes that we consider, namely, Reed-Solomon codes over evaluation domains of certain structure. To this end, let $\mathbb{F}$ be a finite field and $d \in \mathbb{N}$. Further, let $\mathcal{L} \subseteq \mathbb{F}$ be a set. The Reed-Solomon code over $\mathbb{F}$ with evaluation domain $\mathcal{L}$ and degree bound d, denoted by $\mathcal{RS}[d, \mathcal{L}, \mathbb{F}]$, is defined to be the set of all codewords of the form $(f(x))_{x\in\mathcal{L}}$, where $f \in \mathbb{F}[X]$ is a polynomial of degree less than d. Note that we can naturally interpret codewords $f \in \mathcal{RS}[d, \mathcal{L}, \mathbb{F}]$ as vectors in $\mathbb{F}^{|\mathcal{L}|}$ or as maps $f\colon \mathcal{L} \to \mathbb{F}$. One can easily see that this code has minimum distance $(|\mathcal{L}| - d + 1)/|\mathcal{L}|$, rate $\rho = d/|\mathcal{L}|$, and unique decoding radius $(1 - \rho)/2$.

For FRI, we make the following additional assumptions: we assume that $\mathcal{L} \subseteq \mathbb{F}^*$ is a multiplicative subgroup, we assume that $|\mathcal{L}| = 2^n$ and $d = 2^k$ for integers k, n with $k \leq n/2$, meaning that the rate of the code is $\rho = 2^{k-n}$. Let ω_k be a primitive kth root of unity in $\mathbb{F}$. Then, we assume that for any $x \in \mathbb{F}^*$, we have $x \in \mathcal{L}$ if and only if $\omega_k^i x \in \mathcal{L}$ for all $i \in [k]$. Following earlier works [BBHR18, BGK+23], we call $\mathcal{L}$ a smooth multiplicative subgroup.

Overview. Before formally defining the FRI IOPP, we summarize its main idea, thereby introducing more notation. Assume that the prover wants to convince the verifier that the alleged codeword $\mathbf{c}$ is close to the code $\mathcal{C}_0 = \mathcal{RS}[d_0, \mathcal{L}_0, \mathbb{F}]$ for a finite field $\mathbb{F}$, a degree bound $d_0 = 2^k \in \mathbb{N}$, and an evaluation domain $\mathcal{L}_0$ with $|\mathcal{L}_0| = 2^n$ as above. We interpret $\mathbf{c} = G_0$ as a function $G_0\colon \mathcal{L}_0 \to \mathbb{F}$ to which the verifier has oracle access. In the first round, the verifier sends a random challenge $\rho_1 \xleftarrow{\$} \mathbb{F}$ to the prover. The prover uses ρ_1 to derive a new function $G_1\colon \mathcal{L}_1 \to \mathbb{F}$ using a folding procedure which will be explained below. We let $F = 2^\eta$ for an integer $\eta > 0$ be a parameter describing the fan-in of the folding. The prover now claims that G_1 is in a new Reed-Solomon code $\mathcal{C}_1 = \mathcal{RS}[d_1, \mathcal{L}_1, \mathbb{F}]$, where $d_1 = d_0/F = 2^{k-\eta}$. Here, $\mathcal{L}_1 = \{q(x) \mid x \in \mathcal{L}_0\}$ is a new evaluation domain that relates to $\mathcal{L}_0$ via a fixed map $q\colon \mathbb{F} \to \mathbb{F}$ which we assume to be F-to-one on each $\mathcal{L}_i$, i.e., each element in $\mathcal{L}_{i+1}$ has exactly F preimages in $\mathcal{L}_i$ under q. We denote the set of preimages of $s \in \mathcal{L}_{i+1}$ in $\mathcal{L}_i$ by $q^{-1}(s)$. For concreteness, let $q(x) := x^F$. The function $\pi_1 = G_1$ is sent as a new oracle to the verifier. This process is repeated, namely, in each round i, the verifier sends $\rho_i \xleftarrow{\$} \mathbb{F}$, the prover derives $G_i\colon \mathcal{L}_i \to \mathbb{F}$ from G_{i-1} and claims that it is in $\mathcal{C}_i = \mathcal{RS}[d_i, \mathcal{L}_i, \mathbb{F}]$ for $\mathcal{L}_i = \{q(x) \mid x \in \mathcal{L}_{i-1}\}$ and $d_i = d_{i-1}/F$. Note that $d_i = 2^{k-i\cdot\eta}$ for each i. The process is repeated until the final round r, in which the verifier checks that $G_r \in \mathcal{C}_r$ by querying G_r entirely. For example, we can set $r = k/\eta$ (ignoring divisibility issues) such that $d_r = 2^0 = 1$, i.e., the verifier would expect $\pi_r = G_r$ to be a constant. The phase until now is sometimes called the commit phase or the folding phase. Finally, the verifier runs a so-called query phase[8]: it samples $\rho_{r+1} = s_0 \xleftarrow{\$} \mathcal{L}_0$ and uses it to check consistency between G_0 and G_1. Then, it uses $s_1 = q(s_0)$ to check consistency between G_1 and G_2, and so on.

Algebraic Hash Function. We still need to explain how the prover derives G_i from G_{i-1} and ρ_i, and how the verifier checks consistency of this derivation in the query phase. This is done using a so-called algebraic hash function H_{ρ_i} following the terminology in [BGK+23]. This function is indexed by a verifier challenge $\rho_i \in \mathbb{F}$ and maps a function $G_{i-1}\colon \mathcal{L}_{i-1} \to \mathbb{F}$ to a function $H_{\rho_i}[G_{i-1}]\colon \mathcal{L}_i \to \mathbb{F}$. The honest prover defines $G_i := H_{\rho_i}[G_{i-1}]$. The algebraic hash function has the following useful properties: first, if $G_{i-1} \in \mathcal{C}_{i-1}$, then $H_{\rho_i}[G_{i-1}] \in \mathcal{C}_i$ for all $\rho_i \in \mathbb{F}$. Second, for every $s \in \mathcal{L}_i$, one can efficiently compute $H_{\rho_i}[G_{i-1}](s)$ from $G_{i-1}(q^{-1}(s))$, i.e., from the images under G_{i-1} of the preimages of s under q in $\mathcal{L}_{i-1}$. The verifier uses this to check consistency of G_{i-1} and G_i.

[8] The query phase is usually repeated in parallel, but we include this repetition as part of the transformation in Section 3 instead of including it here.

To formally define the function H, we denote by $\mathsf{Interpolate}(z, S)$ the algorithm that takes as input an element $z \in \mathbb{F}$ and a set $S \subseteq \mathbb{F}^2$, defines the unique polyomial P of degree less than $|S|$ such that $P(x) = y$ for all $(x, y) \in S$, and outputs $P(z)$. With this algorithm at hand, we define

$$H_{\rho_i}[G_{i-1}](s_i) := \mathsf{Interpolate}\left(\rho_i, \{(s_{i-1}, G_{i-1}(s_{i-1})) \mid s_{i-1} \in q^{-1}(s_i)\}\right)$$

for all $i \in r$, all $\rho_i \in \mathbb{F}$, all $G_{i-1} \colon \mathcal{L}_{i-1} \to \mathbb{F}$, and all $s_i \in \mathcal{L}_i$. We still need to argue that if $G_{i-1} \in \mathcal{C}_{i-1}$, then $H_{\rho_i}[G_{i-1}] \in \mathcal{C}_i$ for all $\rho_i \in \mathbb{F}$, which is important for completeness of the protocol. The reader can also consult [BBHR18] for a proof. So, assume that $G_{i-1} \in \mathcal{C}_{i-1}$. Then, it is easy to see that there are polynomials $G_{i-1,j}$ of degree less than $d_{i-1}/F = d_i$ such that for the bivariate polynomial $\hat{G}(X, Y) := \sum_{j=0}^{F-1} X^j G_{i-1,j}(Y)$, we have $G_{i-1}(x) = \hat{G}(x, x^F)$ for all $x \in \mathcal{L}_{i-1}$. Now fix $s_i \in \mathcal{L}_i$. We get

$$\begin{aligned}
H_{\rho_i}[G_{i-1}](s_i) &= \mathsf{Interpolate}\left(\rho_i, \{(s_{i-1}, G_{i-1}(s_{i-1})) \mid s_{i-1} \in q^{-1}(s_i)\}\right) \\
&= \mathsf{Interpolate}\left(\rho_i, \{(s_{i-1}, \hat{G}(s_{i-1}, s_{i-1}^F)) \mid s_{i-1} \in q^{-1}(s_i)\}\right) \\
&= \mathsf{Interpolate}\left(\rho_i, \{(s_{i-1}, \hat{G}(s_{i-1}, s_i)) \mid s_{i-1} \in q^{-1}(s_i)\}\right) \\
&= \hat{G}(\rho_i, s_i),
\end{aligned}$$

where we have used that the degree of $\hat{G}(X, s_i)$ in X is less than F. This means that $H_{\rho_i}[G_{i-1}](s_i) = \hat{G}(\rho_i, s_i)$ is of degree less than d_i in s_i.

Construction. With the notation introduced above at hand, we can now formally describe the FRI prover P and the verifier V, where $\rho_1, \ldots, \rho_r \in \mathbb{F}$ and $\rho_{r+1} \in \mathcal{L}_0$.

- $\mathsf{P}(\mathbf{c} = G_0, \rho_1, \ldots, \rho_i) \to \pi_i$ for $i \in \{1, \ldots, r\}$:
  1. Assume $G_0 \colon \mathcal{L}_0 \to \mathbb{F}$ and for each $1 \leq i' < i$ let $G_{i'} \colon \mathcal{L}_{i'} \to \mathbb{F}$ be the output of $\mathsf{P}(\mathbf{c}, \rho_1, \ldots, \rho_{i'})$.
  2. Let $G_i = H_{\rho_i}[G_{i-1}]$. Note that $G_i \colon \mathcal{L}_i \to \mathbb{F}$. Set $\pi_i := G_i$.
- $\mathsf{V}^{\mathbf{c}=G_0, \pi_1=G_1, \ldots, \pi_r=G_r}(\rho_1, \ldots, \rho_r, \rho_{r+1} = s_0) \to b$:
  1. If $G_r \in \mathcal{C}_r$, set $b_0 := 1$. Otherwise, set $b_0 := 0$.
  2. For each $i \in [r]$, do the following:
     (a) Set $s_i := q(s_{i-1})$.
     (b) If $G_i(s_i) = \mathsf{Interpolate}\left(\rho_i, \{(s'_{i-1}, G_{i-1}(s'_{i-1})) \mid s'_{i-1} \in q^{-1}(s_i)\}\right)$, set $b_i := 1$.
     (c) Else, set $b_i := 0$.
  3. Set $b := b_0 \wedge \cdots \wedge b_r$.

4.2 Analysis

Completeness follows directly from our discussion above. It is also clear that the IOPP is non-adaptive (see Definition 7) and query-selectable (see Definition 8).

Blockwise Distance. For some parts of our analysis, we use the notion of blockwise distance, as introduced in [BBHR18]. However, we emphasize that when we refer to the unique closest codeword of a function, we still refer to Hamming distance. Roughly speaking, blockwise distance of two functions measures (as a relative portion) how many groups of preimages under the map q are different.

Definition 11 (Blockwise Distance)**.** Let $i \in [r]$ and let $G, G' \colon \mathcal{L}_{i-1} \to \mathbb{F}$. The blockwise distance $\delta^{\mathsf{B}}(G, G')$ is defined as

$$\delta^{\mathsf{B}}(G, G') := \frac{1}{|\mathcal{L}_i|} \left|\{s_i \in \mathcal{L}_i \mid \exists s_{i-1} \in \mathcal{L}_{i-1} : q(s_{i-1}) = s_i \wedge G(s_{i-1}) \neq G'(s_{i-1})\}\right|.$$

The definition naturally extends to the distance of a function $G \colon \mathcal{L}_i \to \mathbb{F}$ to the code $\mathcal{C}_i$. Throughout our analysis, we use the following lemma that relates the blockwise distance to Hamming distance and the algebraic hash function H.

Lemma 2 (Properties of Blockwise Distance)**.** *Let $i \in [r]$ and consider functions $G, G' \colon \mathcal{L}_{i-1} \to \mathbb{F}$. Then, the following properties hold:*

1. *We have $\delta(G, G') \leq \delta^{\mathsf{B}}(G, G')$.*
2. *We have $\delta^{\mathsf{B}}(G, G') \leq F \cdot \delta(G, G')$.*
3. *For any $\rho \in \mathbb{F}$, we have $\delta(H_\rho[G], H_\rho[G']) \leq \delta^{\mathsf{B}}(G, G')$.*

Proof. To prove the first statement, define the following sets:

$$\begin{aligned} D &:= \{s_{i-1} \in \mathcal{L}_{i-1} \mid G(s_{i-1}) \neq G'(s_{i-1})\}, \\ D_{s_i} &:= \{s_{i-1} \in \mathcal{L}_{i-1} \mid q(s_{i-1}) = s_i \wedge G(s_{i-1}) \neq G'(s_{i-1})\} \text{ for all } s_i \in \mathcal{L}_i, \\ D^{\mathsf{B}} &:= \{s_i \in \mathcal{L}_i \mid D_{s_i} \neq \emptyset\}. \end{aligned}$$

Note that the sets D_{s_i} partition the set D. Therefore, we have

$$\delta(G, G') = \frac{|D|}{|\mathcal{L}_{i-1}|} = \frac{|D|}{F|\mathcal{L}_i|} = \frac{\sum_{s_i \in \mathcal{L}_i} |D_{s_i}|}{F|\mathcal{L}_i|}.$$

Further, note that $\sum_{s_i \in \mathcal{L}_i} |D_{s_i}| \leq F|D^{\mathsf{B}}|$ as each non-empty set in the summation has at most F elements and there are exactly $|D^{\mathsf{B}}|$ non-empty sets. In combination, we get $\delta(G, G') \leq |D^{\mathsf{B}}|/|\mathcal{L}_i|$, which is exactly the definition of $\delta^{\mathsf{B}}(G, G')$, which finishes the proof of the first statement. The second claim follows directly by noting that $|D^{\mathsf{B}}| \leq |D|$. The third claim follows by noting that $H_\rho[G](s_i)$ and $H_\rho[G'](s_i)$ can only differ if there is a preimage s_{i-1} of s_i such that $G(s_{i-1})$ and $G'(s_{i-1})$ differ. □

Correlated Agreement. In addition, we recall a correlated agreement lemma from [BCI+20a], Theorem 6.1 in the eprint version [BCI+20b], restated in our notation.

Lemma 3 (Correlated Agreement, Theorem 6.1 in [BCI+20b])**.** *Let $K \in \mathbb{N}$ be an integer. Consider the code $\mathcal{V} = \mathcal{RS}[d, \mathcal{L}, \mathbb{F}]$ for some $d, \mathcal{L}, \mathbb{F}$, and let $\rho = d/|\mathcal{L}|$ denote its rate and let $\delta \leq (1-\rho)/2$. Let $u_0, \ldots, u_{K-1} \colon \mathcal{L} \to \mathbb{F}$ be functions. Then, one of the following two holds:*

1. *we have $\Pr_\rho\left[\delta\left(\sum_{j=0}^{K-1} \rho^j u_j, \mathcal{V}\right) \leq \delta\right] \leq (K-1) \cdot |\mathcal{L}|/|\mathbb{F}|$, with probability taken over $\rho \xleftarrow{\boxdot} \mathbb{F}$, or*
2. *there exist $v_0, \ldots, v_{K-1} \in \mathcal{V}$ such that for*

$$\mathsf{Disagree} = \{x \in \mathcal{L} \mid (u_0(x), \ldots, u_{K-1}(x)) \neq (v_0(x), \ldots, v_{K-1}(x))\},$$

we have $|\mathsf{Disagree}| \leq \delta|\mathcal{L}|$.

Using correlated agreement, we show the following lemma. We note that in [BGK+23] a similar lemma is stated and it is claimed that it follows from correlated agreement, without an explicit proof for that. We, however, give a detailed proof for completeness.

Lemma 4. *Let $\delta \leq (1-\rho)/2$ be within the unique decoding radius. Let $i \in [r]$ and let $G_{i-1} \colon \mathcal{L}_{i-1} \to \mathbb{F}$ be such that $\delta^{\mathsf{B}}(G_{i-1}, \mathcal{C}_{i-1}) > \delta$. Then, we have*

$$\Pr_{\rho_i \xleftarrow{\boxdot} \mathbb{F}}[\delta(H_{\rho_i}[G_{i-1}], \mathcal{C}_i) \leq \delta] \leq \frac{(F-1)|\mathcal{L}_i|}{|\mathbb{F}|}.$$

Proof. We consider a function $G_{i-1} \colon \mathcal{L}_{i-1} \to \mathbb{F}$ and the experiment of sampling $\rho_i \xleftarrow{\boxdot} \mathbb{F}$. The proof consists of two steps: (1) we find functions $u_0, \ldots, u_{F-1}$ to apply correlated agreement, and (2) we apply correlated agreement to conclude that we are in one of two cases. For one case, we get the desired bound we need to show. For the other case, we argue that it contradicts our assumption that $\delta^{\mathsf{B}}(G_{i-1}, \mathcal{C}_{i-1}) > \delta$.

Let us start with our first step, namely, defining functions $u_0, \ldots, u_{F-1}$. For each $s_i \in \mathcal{L}_i$, fix an arbitrary ordering $s_{i-1,1}, \ldots, s_{i-1,F} \in \mathcal{L}_{i-1}$ of its F preimages under q. Further, consider the Vandermonde

matrix $\mathbf{V}_{s_i} \in \mathbb{F}^{F \times F}$ where the jth row is $(1, s_{i-1,j}, \ldots, s_{i-1,j}^{F-1})$. Note that this matrix is invertible. Define the vector $\mathbf{g}_{s_i} \in \mathbb{F}^F$ where the jth coordinate of $\mathbf{g}_{s_i}$ is $G_{i-1}(s_{i-1,j})$. With this notation, we have

$$H_{\rho_i}[G_{i-1}](s_i) = (1, \rho_i, \ldots, \rho_i^{F-1}) \cdot \mathbf{V}_{s_i}^{-1} \cdot \mathbf{g}_{s_i}.$$

by definition of H. Now, for each $j \in \{0, \ldots, F-1\}$, let $u_j(s_i)$ denote the $(j+1)$st entry of $\mathbf{V}_{s_i}^{-1} \cdot \mathbf{g}_{s_i}$. With this, we have functions $u_0, \ldots, u_{F-1} \colon \mathcal{L}_i \to \mathbb{F}$ such that

$$H_{\rho_i}[G_{i-1}] = \sum_{j=0}^{F-1} \rho_i^j u_j.$$

Our second step is to use the correlated agreement (Lemma 3) with $K := F$, code $\mathcal{V} := \mathcal{C}_i$ and functions $u_0, \ldots, u_{F-1}$ and show how to finish the proof. The correlated agreement states that one of two cases holds. In the simple case, we have

$$\Pr_{\rho_i}\left[\delta\left(H_{\rho_i}[G_{i-1}], \mathcal{C}_i\right) \leq \delta\right] \leq \frac{(F-1)|\mathcal{L}_i|}{|\mathbb{F}|}.$$

Then, we are done, as this is exactly what we have to show. In the other case, we want to derive a contradiction to the assumption that $\delta^{\mathsf{B}}\left(G_{i-1}, \mathcal{C}_{i-1}\right) > \delta$. Hence, we could have never been in this case. So, to derive the contradiction, consider this other case. The correlated agreement states that there are polynomials $v_0, \ldots, v_{F-1} \in \mathbb{F}[X]$ of degree less than d_i such that $|\mathsf{Disagree}| \leq \delta|\mathcal{L}_i|$, where

$$\mathsf{Disagree} = \left\{s_i \in \mathcal{L}_i \mid (u_0(s_i), \ldots, u_{F-1}(s_i)) \neq (v_0(s_i), \ldots, v_{F-1}(s_i))\right\}.$$

Define the vector $\mathbf{v}_{s_i} \in \mathbb{F}^F$ where the jth entry of $\mathbf{v}_{s_i}$ is $v_{j-1}(s_i)$. Using this and the definition of the u_j, we get

$$\mathsf{Disagree} = \left\{s_i \in \mathcal{L}_i \mid \mathbf{V}_{s_i}^{-1} \cdot \mathbf{g}_{s_i} \neq \mathbf{v}_{s_i}\right\} = \left\{s_i \in \mathcal{L}_i \mid \mathbf{g}_{s_i} \neq \mathbf{V}_{s_i} \cdot \mathbf{v}_{s_i}\right\}.$$

Note that the jth entry of $\mathbf{V}_{s_i} \cdot \mathbf{v}_{s_i}$ is $\sum_{j=0}^{F-1} s_{i-1,j}^j v_j(s_i)$. Hence, there are at most $|\mathsf{Disagree}|$ F-tuples for which G_{i-1} disagrees with the polynomial $v := \sum_{j=0}^{F-1} X^j v_j(X^F) \in \mathbb{F}[X]$. The degree of v is less than $F \cdot d_i = d_{i-1}$. This is a contradiction to the assumption that $\delta^{\mathsf{B}}\left(G_{i-1}, \mathcal{C}_{i-1}\right) > \delta$, finishing the proof. □

Proof of Opening-Consistency. We are now ready to prove the opening-consistency (see Definition 10) of the FRI IOPP. For that, our first step is to define suitable sets Lucky (Definition 12) and Bad (Definition 13). We start with the definition of the lucky set Lucky. Intuitively, Lucky contains transcripts for which the oracle G_{i-1} sent by the prover and its closest codeword G_{i-1}^* collide under H_{ρ_i}. Additionally, it contains transcripts for which the distance to the code when honestly folding decreases.

Definition 12 (Lucky Set for FRI)**.** Consider the FRI IOPP and let $\delta^* = (1-\rho)/2$ be the unique decoding radius. We define the set $\mathsf{Lucky} = \mathsf{LuckyColl} \cup \mathsf{LuckyDist}$ of partial prover-turn transcripts as follows:

- **Lucky Collision.** A partial prover-turn transcript $(G_0, \rho_1, \ldots, G_{i-1}, \rho_i)$ is in $\mathsf{LuckyColl}$, if and only if the following two properties hold:
  1. G_{i-1} is within the unique decoding radius, i.e., $\delta\left(G_{i-1}, \mathcal{C}_{i-1}\right) \leq \delta^*$. Let $G_{i-1}^* \in \mathcal{C}_{i-1}$ be the unique closest codeword.
  2. There exists an $s_i \in \mathcal{L}_i$ such that $H_{\rho_i}[G_{i-1}](s_i) = H_{\rho_i}[G_{i-1}^*](s_i)$, but there is an $s_{i-1} \in \mathcal{L}_{i-1}$ with $q(s_{i-1}) = s_i$ and $G_{i-1}(s_{i-1}) \neq G_{i-1}^*(s_{i-1})$.
- **Lucky Distortion.** A partial prover-turn transcript $(G_0, \rho_1, \ldots, G_{i-1}, \rho_i)$ is in $\mathsf{LuckyDist}$ if and only if (a) $\delta^{\mathsf{B}}\left(G_{i-1}, \mathcal{C}_{i-1}\right) > \delta^*$ and $\delta\left(H_{\rho_i}[G_{i-1}], \mathcal{C}_i\right) \leq \delta^*$ or (b) $\delta\left(H_{\rho_i}[G_{i-1}], \mathcal{C}_i\right) < \delta^{\mathsf{B}}\left(G_{i-1}, \mathcal{C}_{i-1}\right) \leq \delta^*$.

Next, we define the set Bad for FRI. The definition of this set is not obvious and is partially inspired by the original analysis of FRI [BBHR18]. Concretely, a transcript can only be in the bad set if no prefix was lucky, and if one of three bad properties hold. Either, the final oracle that the prover sends is not in the code. In this case, it is clear that the verifier rejects so it is reasonable to mark such a transcript as bad. Or, one of the oracles is too far away from its respective code. Intuitively, the analysis of FRI should guarantee that such transcripts are rejected as well. The final case is where for one of the rounds, taking the closest codeword and folding do not commute. Marking such transcripts as bad will turn out to be crucial for proving the *inconsistent is rejected* property.

Definition 13 (Bad Set for FRI)**.** Let $\delta^* = (1-\rho)/2$ be the unique decoding radius. We define the set Bad of partial transcripts as follows. A partial transcript $(G_0, \rho_1, G_1, \ldots, \rho_r, G_r)$ is in Bad, if and only if no prover-turn prefix of it is in Lucky, and at least one of the following properties holds:

1. We have $G_r \notin \mathcal{C}_r$, or
2. There is an $i \in \{0, \ldots, r-1\}$ such that $\delta^{\mathsf{B}}(G_i, \mathcal{C}_i) > \delta^*$, or
3. For all $i \in \{0, \ldots, r-1\}$ we have $\delta^{\mathsf{B}}(G_i, \mathcal{C}_i) \leq \delta^*$, but there is an $i \in [r]$ such that $G_i^* \neq H_{\rho_i}[G_{i-1}^*]$, where the G_i^*'s denote the unique closest codewords of the G_i's respectively.

Now that these sets are defined, we can prove the four properties of opening-consistency, namely the *no luck* property (Lemma 5), the *bad is rejected* property (Lemma 8), the *suitable is close* property (Lemma 9), and the *inconsistent is rejected* property (Lemma 10). To make our analysis self-contained, we give full proofs, even if variants of some statements needed on the way have been proven in the original analysis of FRI [BBHR18] using a different notation.

Lemma 5 (No Luck)**.** *The FRI IOPP satisfies the* no luck *property of opening-consistency (Definition 10) with* Lucky *as in Definition 12 and* $\epsilon_1 \leq 2(F-1)|\mathcal{L}_0|/|\mathbb{F}|$.

Proof. Let $i \in [r]$, let $T = (G_0, \rho_1, \ldots, G_{i-1})$ be a partial verifier-turn transcript, and consider the experiment of sampling $\rho_i \xleftarrow{\$} \mathbb{F}$. We need to bound the probability of the event $T \circ \rho_i \in \mathsf{Lucky}$. By a union bound and the definition of $\mathsf{Lucky} = \mathsf{LuckyColl} \cup \mathsf{LuckyDist}$, we have

$$\Pr_{\rho_i}[T \circ \rho_i \in \mathsf{Lucky}] \leq \Pr_{\rho_i}[T \circ \rho_i \in \mathsf{LuckyColl}] + \Pr_{\rho_i}[T \circ \rho_i \in \mathsf{LuckyDist}].$$

We bound these events separately.

Claim. We have $\Pr_{\rho_i}[T \circ \rho_i \in \mathsf{LuckyColl}] \leq (F-1)|\mathcal{L}_0|/|\mathbb{F}|$.

Proof of Claim. If $\delta(\mathcal{C}_{i-1}, G_{i-1}) > \delta^*$, then we are done by definition of $\mathsf{LuckyColl}$. Otherwise, let $G_{i-1}^* \in \mathcal{C}_{i-1}$ be the unique closest codeword. For each $s_i \in \mathcal{L}_i$, denote by $\mathsf{LuckyColl}_{s_i}$ the event that $T \circ \rho_i \in \mathsf{LuckyColl}$ because of s_i. With a union bound, we get

$$\Pr_{\rho_i}[T \circ \rho_i \in \mathsf{LuckyColl}] \leq \sum_{s_i \in \mathcal{L}_i} \Pr_{\rho_i}\left[\mathsf{LuckyColl}_{s_i}\right].$$

In the following, we fix an arbitrary $s_i \in \mathcal{L}_i$ and bound the probability of $\mathsf{LuckyColl}_{s_i}$ by $(F-1)/|\mathbb{F}|$. Then, as $|\mathcal{L}_i| \leq |\mathcal{L}_0|$, we will get the desired bound. We now turn to bounding the probability of $\mathsf{LuckyColl}_{s_i}$. If $\mathsf{LuckyColl}_{s_i}$ occurs, we must have $H_{\rho_i}[G_{i-1}](s_i) = H_{\rho_i}[G_{i-1}^*](s_i)$ and

$$\{(s_{i-1}, G_{i-1}(s_{i-1})) \mid s_{i-1} \in q^{-1}(s)\} \neq \{(s_{i-1}, G_{i-1}^*(s_{i-1})) \mid s_{i-1} \in q^{-1}(s)\}.$$

The expressions $H_{\rho_i}[G_{i-1}](s_i)$ and $H_{\rho_i}[G_{i-1}^*](s_i)$ that must be equal if $\mathsf{LuckyColl}_{s_i}$ occurs are both polynomials of degree (at most) $F-1$ in the variable ρ_i. Denote the vectors of their coefficients by $\mathbf{p} \in \mathbb{F}^F$ and $\mathbf{p}^* \in \mathbb{F}^F$, respectively. If we can argue that they are different, then we are done. To argue that they are different, first fix some notation. Namely, fix an arbitrary ordering $s_{i-1,1}, \ldots, s_{i-1,F} \in \mathcal{L}_{i-1}$ of the F preimages of s_i under q. Consider the Vandermonde matrix $\mathbf{V}_{s_i} \in \mathbb{F}^{F \times F}$ where the jth row is $(1, s_{i-1,j}, \ldots, s_{i-1,j}^{F-1})$. This matrix is invertible. Define the vector $\mathbf{g}_{s_i} \in \mathbb{F}^F$ where the jth coordinate is $G_{i-1}(s_{i-1,j})$, and define the vector $\mathbf{g}_{s_i}^* \in \mathbb{F}^F$ where the jth coordinate is $G_{i-1}^*(s_{i-1,j})$. If $\mathsf{LuckyColl}_{s_i}$, we know $\mathbf{g}_{s_i} \neq \mathbf{g}_{s_i}^*$. But then, using invertibility of $\mathbf{V}_{s_i}$, we also know that $\mathbf{p} = \mathbf{V}_{s_i}^{-1}\mathbf{g}_{s_i} \neq \mathbf{V}_{s_i}^{-1}\mathbf{g}_{s_i}^* = \mathbf{p}^*$.

Claim. We have $\Pr_{\rho_i}[T \circ \rho_i \in \mathsf{LuckyDist}] \leq (F-1)|\mathcal{L}_0|/|\mathbb{F}|$.

Proof of Claim. The claim follows directly from Lemma 4 and from $|\mathcal{L}_i| \leq |\mathcal{L}_0|$. □

To prove the *bad is rejected* property, we first prove two statements that appear (with variations) in the original FRI paper [BBHR18].

Lemma 6 (Variant of Lemma 4.4 in [BBHR18])**.** *Let $\delta^* = (1-\rho)/2$ be the unique decoding radius. Let $T = (G_0, \rho_1, G_1, \ldots, \rho_r, G_r)$ be a partial transcript. Let $i^* \in \{0, \ldots, r-1\}$ be fixed and assume that the following properties hold:*

1. *No prover-turn prefix of T is in* Lucky*.*
2. *We have $G_r \in \mathcal{C}_r$.*
3. *For every $i \in \{i^*, \ldots, r\}$, we have $\delta(G_i, \mathcal{C}_i) \leq \delta^*$. Let the G_i^*'s denote the unique closest codewords of the G_i's respectively.*
4. *For every $i \in \{i^*+1, \ldots, r\}$, we have $G_i^* = H_{\rho_i}[G_{i-1}^*]$.*

Then, we have

$$\Pr_{s_0 \xleftarrow{\boxtimes} \mathcal{L}_0}\left[\mathsf{V}^{G_0, G_1, \ldots, G_r}(\rho_1, \ldots, \rho_r, s_0) = 0 \mid G_{i^*}(s_{i^*}) \neq G_{i^*}^*(s_{i^*})\right] = 1,$$

where $s_{i^*} \in \mathcal{L}_{i^*}$ is defined as in algorithm V*.*

Proof. Consider the transcript $T = (G_0, \rho_1, G_1, \ldots, \rho_r, G_r)$. Further, consider the experiment of sampling $s_0 \xleftarrow{\boxtimes} \mathcal{L}_0$ and running the verifier. To recall, the verifier defines a sequence $s_0, \ldots, s_r$, where each s_i individually follows a uniform distribution over $\mathcal{L}_i$. We condition on $G_{i^*}(s_{i^*}) \neq G_{i^*}^*(s_{i^*})$ as in the statement and want to show that the verifier rejects. To this end, fix the index i to be the maximum i such that $G_i(s_i) \neq G_i^*(s_i)$. Note that this maximum exists because we condition on $G_{i^*}(s_{i^*}) \neq G_{i^*}^*(s_{i^*})$, and note that it satisfies $i < r$ because $G_r^* = G_r$. To finish the proof, we claim that $G_{i+1}(s_{i+1}) \neq H_{\rho_{i+1}}[G_i](s_{i+1})$, making the verifier reject. We now prove the claim by contradiction. Assume towards contradiction that $G_{i+1}(s_{i+1}) = H_{\rho_{i+1}}[G_i](s_{i+1})$. By the choice of i, we have $G_{i+1}^*(s_{i+1}) = G_{i+1}(s_{i+1})$. By our assumption in the lemma, we have $H_{\rho_{i+1}}[G_i^*] = G_{i+1}^*$. In combination, this yields $H_{\rho_{i+1}}[G_i^*](s_{i+1}) = H_{\rho_{i+1}}[G_i](s_{i+1})$. Therefore, the prover-turn prefix $(G_0, \rho_1, G_1, \ldots, \rho_{i+1})$ is in Lucky (concretely, in LuckyColl), a contradiction. □

Lemma 7 (Variant of Soundness in [BBHR18])**.** *Let $\delta^* = (1-\rho)/2$ be the unique decoding radius. Let $T = (G_0, \rho_1, G_1, \ldots, \rho_r, G_r)$ be a partial transcript and assume that the following properties hold:*

1. *No prover-turn prefix of T is in* Lucky*.*
2. *We have $G_r \in \mathcal{C}_r$.*

Further, assume that at least one of the following holds:

1. *There is an $i \in \{0, \ldots, r-1\}$ such that $\delta^{\mathsf{B}}(G_i, \mathcal{C}_i) > \delta^*$, or*
2. *For all $i \in \{0, \ldots, r-1\}$ we have $\delta^{\mathsf{B}}(G_i, \mathcal{C}_i) \leq \delta^*$, but there is an $i \in [r]$ such that $G_i^* \neq H_{\rho_i}[G_{i-1}^*]$, where the G_i^*'s denote the unique closest codewords of the G_i's respectively.*

Then, we have

$$\Pr_{s_0 \xleftarrow{\boxtimes} \mathcal{L}_0}\left[\mathsf{V}^{G_0, G_1, \ldots, G_r}(\rho_1, \ldots, \rho_r, s_0) = 1\right] \leq 1 - \delta^*.$$

Proof. To prove the lemma, we fix a partial transcript $T = (G_0, \rho_1, G_1, \ldots, \rho_r, G_r)$ as above, and define the following sets Far and Incons:

$$\begin{aligned}\mathsf{Far} &:= \{i \in \{0, \ldots, r-1\} \mid \delta^{\mathsf{B}}(G_i, \mathcal{C}_i) > \delta^*\},\\ \mathsf{Incons} &:= \{i \in \{0, \ldots, r-1\} \mid i \notin \mathsf{Far} \wedge i+1 \notin \mathsf{Far} \wedge G_{i+1}^* \neq H_{\rho_{i+1}}[G_i^*]\}.\end{aligned}$$

Note that G_i^* and G_{i+1}^* in the definition of Incons are uniquely defined, due to the definition of Far. By our assumptions, we know $\mathsf{Far} \cup \mathsf{Incons} \neq \emptyset$. Thus, the maximum $i := \max \mathsf{Far} \cup \mathsf{Incons}$ exists. We also

know that $i < r$. Further, by definition of i, we have $i+1 \notin \mathsf{Far}$. Therefore, the unique closest codeword $G^*_{i+1} \in \mathcal{C}_{i+1}$ for G_{i+1} exists. Define the sets Dis and Find as follows:

$$\mathsf{Dis} := \{x \in \mathcal{L}_{i+1} \mid G^*_{i+1}(x) \neq G_{i+1}(x)\},$$
$$\mathsf{Find} := \{x \in \mathcal{L}_{i+1} \mid H_{\rho_{i+1}}[G_i](x) \neq G_{i+1}(x)\}.$$

Intuitively, Dis is the set of positions for which G_i and its closest codeword disagree, and Find is the set of positions for which the consistency check in the verifier would fail. Now that this notation is fixed, the lemma by combining the following three claims.

Claim. We have $\delta\left(G^*_{i+1}, H_{\rho_{i+1}}[G_i]\right) \geq \delta^*$.

Proof of Claim. To prove the claim, we consider two cases. In the first case, we have $i \in \mathsf{Far}$. Then, $\delta\left(G_i, \mathcal{C}_i\right) > \delta^*$. By our assumption that no prover-turn prefix of T is in Lucky, we get that $\delta\left(G^*_{i+1}, H_{\rho_{i+1}}[G_i]\right) \geq \delta\left(\mathcal{C}_{i+1}, H_{\rho_{i+1}}[G_i]\right) \geq \delta^*$, finishing the proof of the claim for this case. In the second case, we have $i \notin \mathsf{Far}$. Therefore, $i+1 \notin \mathsf{Far}$ and $G^*_{i+1} \neq H_{\rho_{i+1}}[G^*_i]$ by definition of $\mathsf{Far} \cup \mathsf{Incons}$. Recall that both G^*_{i+1} and $H_{\rho_{i+1}}[G^*_i]$ are in the code $\mathcal{C}_{i+1}$. This yields

$$\begin{aligned}
1 - \rho \leq \delta\left(G^*_{i+1}, H_{\rho_{i+1}}[G^*_i]\right) &\leq \delta\left(G^*_{i+1}, H_{\rho_{i+1}}[G_i]\right) + \delta\left(H_{\rho_{i+1}}[G_i], H_{\rho_{i+1}}[G^*_i]\right) \\
&\leq \delta\left(G^*_{i+1}, H_{\rho_{i+1}}[G_i]\right) + \delta^{\mathsf{B}}\left(G_i, G^*_i\right) \\
&\leq \delta\left(G^*_{i+1}, H_{\rho_{i+1}}[G_i]\right) + \delta^*,
\end{aligned}$$

where we used Lemma 2 and $i \notin \mathsf{Far}$. By rearranging, we get $\delta\left(G^*_{i+1}, H_{\rho_{i+1}}[G_i]\right) \geq 1 - \rho - \delta^* = \delta^* \geq \delta^*$, finishing the proof of the claim.

Claim. We have $|\mathsf{Dis} \cup \mathsf{Find}|/|\mathcal{L}_{i+1}| \geq \delta\left(G^*_{i+1}, H_{\rho_{i+1}}[G_i]\right)$.

Proof of Claim. Note that for every $x \in \mathcal{L}_{i+1} \setminus (\mathsf{Dis} \cup \mathsf{Find})$, we have $G^*_{i+1}(x) = G_{i+1}(x) = H_{\rho_{i+1}}[G_i](x)$. Hence, the set of positions for which G^*_{i+1} and $H_{\rho_{i+1}}[G_i]$ disagree can have size at most $|\mathsf{Dis} \cup \mathsf{Find}|$. The claim follows.

Claim. We have $\Pr_{s_0 \xleftarrow{\boxtimes} \mathcal{L}_0}\left[\mathsf{V}^{G_0, G_1, \ldots, G_r}(\rho_1, \ldots, \rho_r, s_0) = 1\right] \leq 1 - (|\mathsf{Dis} \cup \mathsf{Find}|/|\mathcal{L}_{i+1}|)$.

Proof of Claim. To prove the claim, we consider two cases. In the first case, we have $i+1 = r$. Then, by assumption we have $\mathsf{Dis} = \emptyset$, and by definition of the verifier, if it accepts, we know $s_{i+1} \notin \mathsf{Find}$, i.e.,

$$\begin{aligned}
\Pr_{s_0 \xleftarrow{\boxtimes} \mathcal{L}_0}\left[\mathsf{V}^{G_0, G_1, \ldots, G_r}(\rho_1, \ldots, \rho_r, s_0) = 1\right] &\leq \Pr_{s_0 \xleftarrow{\boxtimes} \mathcal{L}_0}\left[s_{i+1} \notin \mathsf{Find}\right] \\
&= 1 - \frac{|\mathsf{Find}|}{|\mathcal{L}_{i+1}|} = 1 - \frac{|\mathsf{Dis} \cup \mathsf{Find}|}{|\mathcal{L}_{i+1}|}.
\end{aligned}$$

In the second case, we have $i+1 < r$. Again, we know that if the verifier accepts, then $s_{i+1} \notin \mathsf{Find}$. We now invoke Lemma 6 with $i^* := i+1$. Note that the conditions for Lemma 6 are satisfied due to the definition of i and the assumptions in the lemma we are about to prove. By Lemma 6 with $i^* := i+1$, we know that if the verifier accepts, then $s_{i+1} \notin \mathsf{Dis}$. Therefore, we get

$$\begin{aligned}
\Pr_{s_0 \xleftarrow{\boxtimes} \mathcal{L}_0}\left[\mathsf{V}^{G_0, G_1, \ldots, G_r}(\rho_1, \ldots, \rho_r, s_0) = 1\right] &\leq \Pr_{s_0 \xleftarrow{\boxtimes} \mathcal{L}_0}\left[s_{i+1} \notin \mathsf{Find} \wedge s_{i+1} \notin \mathsf{Dis}\right] \\
&= 1 - \frac{|\mathsf{Dis} \cup \mathsf{Find}|}{|\mathcal{L}_{i+1}|},
\end{aligned}$$

finishing the proof of the claim. □

Lemma 8 (Bad is Rejected)**.** *The FRI IOPP satisfies the* bad is rejected *property of opening-consistency (Definition 10) with* Bad *as in Definition 13 and* $\epsilon_2 \leq 1 - \delta^*$.

Proof. Let $T = (G_0, \rho_1, G_1, \ldots, \rho_r, G_r)$ be a partial transcript such that $T \in \mathsf{Bad}$, where Bad is as defined in Definition 13. We consider the experiment of sampling $\rho_{r+1} = s_0 \xleftarrow{\boxtimes} \mathcal{L}_0$ and running the verifier on the complete transcript $T \circ s_0$. We have to upper bound the probability of the event that the verifier accepts. Consider two cases. In the first case, $G_r \notin \mathcal{C}_r$. In this case, it is clear that the verifier rejects.

So, assume $G_r \in \mathcal{C}_r$. Then, by definition of Bad, there is an $i \in \{0, \ldots, r-1\}$ such that $\delta^{\mathsf{B}}(G_i, \mathcal{C}_i) > \delta^*$, or for all $i \in \{0, \ldots, r-1\}$ we have $\delta^{\mathsf{B}}(G_i, \mathcal{C}_i) \leq \delta^*$, but there is an $i \in [r]$ such that $G_i^* \neq H_{\rho_i}[G_{i-1}^*]$, where the G_i^*'s denote the unique closest codewords of the G_i's respectively. This means that we can apply Lemma 7 and the lemma follows. □

Lemma 9 (Suitable is Close)**.** *The FRI IOPP satisfies the* suitable is close *property of opening-consistency (Definition 10) with* Bad *as in Definition 13 and* Lucky *as in Definition 12.*

Proof. Let $T = (G_0, \rho_1, \ldots, G_r)$ be a suitable transcript (see Definition 9) with respect to the sets Bad as in Definition 13 and Lucky as in Definition 12. To recall, suitable means that $T \notin \mathsf{Bad}$ and no prefix of T is in Lucky. By the specific definition of Bad and Lucky, we especially have that for all $i \in \{0, \ldots, r-1\}$, it holds that $\delta^{\mathsf{B}}(G_i, \mathcal{C}_i) \leq \delta^*$, which implies $\delta(G_i, \mathcal{C}_i) \leq \delta^*$ by Lemma 2. Especially, this holds for G_0, which is what we had to show. □

Lemma 10 (Inconsistent is Rejected)**.** *The FRI IOPP satisfies the* inconsistent is rejected *property of opening-consistency (Definition 10) with* Bad *as in Definition 13 and* Lucky *as in Definition 12.*

Proof. Let $T = (G_0, \rho_1, \ldots, G_r)$ be a suitable transcript (see Definition 9) with respect to the sets Bad as in Definition 13 and Lucky as in Definition 12. To recall, suitable means that $T \notin \mathsf{Bad}$ and no prefix of T is in Lucky. We have seen in the proof of Lemma 9 that – by definition of Bad and Lucky – for each $i \in \{0, \ldots, r\}$, the distance of G_i to its respective code $\mathcal{C}_i$ is at most δ^*. For each $i \in \{0, \ldots, r\}$, let $G_i^* \in \mathcal{C}_i$ denote the unique closest codeword for G_i. By definition of Bad, we know that for every $i \in [r]$, it holds that $G_i^* = H_{\rho_i}[G_{i-1}^*]$. We now consider completing the transcript T with $\rho_{r+1} = s_0 \in \mathcal{L}_0$. For each $i \in [r]$, let s_i be as in the FRI verifier, i.e., $s_i = q(s_{i-1})$. Now, recalling the definition of the *inconsistent is rejected* property (Definition 10), we need to assume that there is a query $x \in \mathcal{Q}_0(T \circ s_0) \subseteq \mathcal{L}_0$ such that $G_0^*(x) \neq G_0(x)$, and we need to show that the complete transcript $T \circ s_0$ is rejected. We define the index i_0 to be the minimum $i \in \{0, \ldots, r\}$ such that for every query $s_i' \in \mathcal{L}_i$ made by the verifier it holds that $G_i(s_i') = G_i^*(s_i')$. Note that this index exists because $G_r = G_r^*$. Also, because there is a query $x \in \mathcal{Q}_0(T \circ s_0) \subseteq \mathcal{L}_0$ such that $G_0^*(x) \neq G_0(x)$, we know that $i_0 > 0$. Now, assume towards contradiction that $T \circ s_0$ is accepted. Then, we know that

$$\forall i \in [r]: \quad G_i(s_i) = \mathsf{Interpolate}\left(\rho_i, \{(s_{i-1}, G_{i-1}(s_{i-1})) \mid s_{i-1} \in q^{-1}(s)\}\right).$$

This yields the following equation:

$$H_{\rho_{i_0}}[G_{i_0-1}](s_{i_0}) = G_{i_0}(s_{i_0}) = G_{i_0}^*(s_{i_0}).$$

Here, the first equality is because we assume the verifier accepts and the second equality follows from the definition of i_0. Further, we have

$$G_{i_0}^*(s_{i_0}) = H_{\rho_{i_0}}[G_{i_0-1}^*](s_{i_0}),$$

due to $G_i^* = H_{\rho_i}[G_{i-1}^*]$ for all i. So, we get $H_{\rho_{i_0}}[G_{i_0-1}](s_{i_0}) = H_{\rho_{i_0}}[G_{i_0-1}^*](s_{i_0})$. By definition of i_0, we also know that there is an $s_{i_0-1}' \in \mathcal{L}_{i_0-1}$ with $q(s_{i_0-1}') = s_i$ and $G_{i_0-1}(s_{i_0-1}') \neq G_{i_0-1}^*(s_{i_0-1}')$. Therefore, we get

$$(G_0, \rho_1, \ldots, G_{i_0-1}, \rho_{i_0}) \in \mathsf{LuckyColl} \subseteq \mathsf{Lucky},$$

contradicting the assumption that the transcript T is suitable. □

The following theorem summarizes what we have shown.

Theorem 3 (Opening-Consistency of FRI). *The FRI IOPP is opening-consistent with errors $\varepsilon_1, \varepsilon_2$ with respect to the sets* Bad *and* Lucky *as defined in Definitions 12 and 13, where*

$$\varepsilon_1 \leq \frac{2(F-1)|\mathcal{L}_0|}{|\mathbb{F}|}, \quad \varepsilon_2 \leq 1 - \delta^*.$$

4.3 Extensions: Batched FRI

Using FRI as analyzed above would result in a small commitment but in a large encoding. To reduce this size, we split the data into several polynomials and then do the encoding using the batched variant of FRI [BCI+20a]. In our terminology, batched FRI with batch size $B \in \mathbb{N}$ is an IOPP for the interleaved Reed-Solomon code. This code contains codewords of the form $\mathbf{G} = (\mathbf{G}_j)_{j=1}^{B}$ where each $\mathbf{G}_j$ is in the Reed-Solomon code. The main idea is to run FRI on a random linear combination of the $\mathbf{G}_j$. Namely, in an initial folding round, the verifier sends $\xi \xleftarrow{\$} \mathbb{F}$ to the prover. The prover responds with $G_0 = \sum_{j=1}^{B} \xi^{j-1}\mathbf{G}_j$. Then, prover and verifier engage in the FRI protocol for codeword G_0. In the query phase, the verifier samples $s_0 \xleftarrow{\$} \mathcal{L}_0$ performs the consistency checks as in FRI. In addition, the verifier checks consistency of $\mathbf{G}$ and G_0 by checking $G_0(s_0) = \sum_{j=1}^{B} \xi^{j-1}\mathbf{G}_j(s_0)$. We refer to Appendix C for the detailed description of the construction and its full analysis. Especially, we prove the following theorem, which summarizes the opening-consistency of batched FRI.

Theorem 4 (Opening-Consistency of Batched FRI). *The batched FRI IOPP is opening-consistent with errors $\varepsilon_1, \varepsilon_2$ with respect to the sets* BadBatch *and* LuckyBatch *as defined in Definitions 18 and 19, where*

$$\varepsilon_1 \leq \frac{2(\max\{F, B\} - 1)|\mathcal{L}_0|}{|\mathbb{F}|}, \quad \varepsilon_2 \leq 1 - \delta^*.$$

To prove opening-consistency, we need to define a suitable lucky set LuckyBatch and a bad set BadBatch. Roughly, we take the respective sets Lucky and Bad from our analysis of the non-batched version (Section 4) and add suitable conditions to account for the initial batching round. To understand what we need to change, it is instructive to view the mapping from $\mathbf{G}$ to G_0 (parameterized by ξ) as a algebraic hash function. That is, we can think of the protocol as essentially being FRI but in the initial round a different algebraic hash function is used. Our analysis follows this intuition and naturally applies the strategy from our proof for the non-batched version. For details, we refer to Appendix C.

5 Efficiency Evaluation

We now discuss the efficiency of our FRI-based data availability sampling scheme and compare it to existing schemes[9]. To recall, we first obtain an erasure code commitment scheme for the (interleaved) Reed-Solomon code by applying the compiler in Section 3 to the (batched) FRI IOPP that we have analyzed. Applying the transformation from [HASW23], we then obtain a data availability sampling scheme, which we call FRIDA.

Efficiency Metrics. We follow the evaluation framework in [HASW23]. That is, we compare the *size of commitments* as clients always need to download the commitment. We also evaluate the *size of an encoding symbol*. To recall (see also Appendix A), a symbol of the encoding consists of one symbol of the codeword of the erasure code and the respective commitment opening. We may also consider the *communication per query*, which is the size of an index in the encoding (client upload) plus the size of an encoding symbol (client download). In natural cases, however, communication per query and size of an encoding symbol is almost the same. We also evaluate the total *size of the encoding*, which is what has to be stored in the network. The encoding contains one symbol per symbol of the codeword. As the final metric, we consider the *total communication cost*. For that, we first evaluate how many random queries clients need to issue before we can be sure that the probability of reconstructing the data is asymptotically overwhelming, or concretely, at least $1 - 2^{-40}$. This is done following the methodology in [HASW23] using a generalized coupon collector bound, where we assume clients sample their queries uniformly with replacement. We then multiply this number of samples with the communication per query.

Existing Constructions. We compare data availability schemes which are based purely on hash functions, where we assume SHA-256 for our concrete efficiency evaluation. As a baseline, we consider the trivial schemes Naive, Merkle as done in [HASW23]. To recall, in Naive the entire data is put into a single symbol of the encoding, and the commitment is simply a hash of the data. This scheme is ideal in terms of total communication but disqualifies because of its huge communication complexity per query, namely, clients need to download the entire data. Scheme Merkle is uses the identity code (i.e., the codeword is

[9] All numbers, tables, and graphs are computed using the Python scripts given in Appendix D.

Scheme	Commitment	Encoding	Symbol	Samples
Naive	λ	D	D	1
Merkle	λ	$D\lambda \log D$	$\lambda \log(D)$	$D\lambda + D \log D$
Hash	$\lambda\sqrt{D}$	D	$\sqrt{D}$	$\lambda + \sqrt{D}$
FRIDA	$\lambda^2 \log^2(D)$	$D\lambda \log^2(D)$	$\lambda \log^2(D)$	$\lambda + D$

Table 1: Asymptotic efficiency comparison of different data availability sampling schemes based on hash functions. Here, $D = |\mathsf{data}|$ denotes the size of the encoded data, λ denotes the security parameter. We compare the size of commitments, encodings, the size of a symbol in the encoding determining the communication complexity per query, and the number of samples such that data can be reconstructed with overwhelming probability in λ. For all codes that are used, we treat the rate and field as a constant to avoid clutter.

the data) and Merkle trees [Mer88] as a vector commitment. This scheme disqualifies in terms of total communication, because we would need to collect all symbols of the encoding to reconstruct. The main competitor of our new scheme FRIDA is the hash-based construction Hash introduced in [HASW23]. We instantiate Hash exactly as in [HASW23]. For completeness, we also consider the concrete efficiency of the scheme Tensor from [HASW23], which relies on polynomial commitments and the tensor code of two Reed-Solomon codes with rate 1/2. While this scheme relies on pairings, the algebraic group model [FKL18], and stronger assumptions, it is the most relevant scheme in practice, due to its envisioned use in Ethereum. Note that we can not implement this scheme using hash-based polynomial commitments, as it requires that the commitments are homomorphic.

Setting FRIDA's Parameters. To evaluate the concrete efficiency for our scheme FRIDA, we need to set parameters, e.g., the fan-in F or the batch size B. We also need to decide at which level we stop folding, i.e., how to set the number of rounds r or equivalently, the dimension of $\mathcal{C}_r$. To find good parameters, we first fix the same rate as for Hash, namely, $\rho = 1/4$ and a field size of 2^{128}. Then, we implement the following strategy: we iterate through reasonable choices for F, e.g., $F \in \{4, 8, 16\}$ and the dimension of $\mathcal{C}_r$. For each such choice, we first determine a good batch size B by minimizing the size of an opening while iterating over different choices of B. Then, we compute the gap between the number of field elements we could maximally represent with this fan-in, dimension, and batch size and the number of field elements needed to represent the data. We pick the choice that minimizes this gap. To determine the number of the repetitions L of the query phase (see Theorem 2), we aim for 40 bits statistical security while accounting for 2^{60} random oracle queries. We additionally increase the security level by 20 bits using grinding techniques [Sta21, Hab22].

Asymptotic Efficiency. In Table 1, we compare the asymptotic efficiency of data availability sampling schemes based only on hash functions. This includes our scheme FRIDA, the scheme Hash from [HASW23] and the two baseline schemes Naive and Merkle. To avoid clutter, we assume Reed-Solomon codes with constant rate and field sizes for both Hash and FRIDA. We set the repetition parameters for Hash accordingly. Moreover, for FRIDA construction, we assume constant fan-in F and batch size B. Assuming constant rate, setting the number of repetitions of the query phase to $L = \Theta(\lambda)$ is sufficient. With this, we get bounds that only depend on the data size $D = |\mathsf{data}|$ and the security parameters λ. Comparing the results, we see that while Hash is optimal in terms of encoding size, in terms of commitment size and the size of a single encoding symbol, FRIDA is an exponential improvement. In terms of the number of samples needed to reconstruct, FRIDA performs slightly worse than Hash. Note that we could also set the batch size to $B = \Theta(\sqrt{D})$, which would result in an encoding length of $\Theta(\sqrt{D})$ symbols and to the same number of samples as in Hash. However, this would also result in encoding symbols of size $\Omega(\sqrt{D})$. Computing the encoding takes $O(\lambda D)$ time for Naive (the cost of hashing the data), it takes $O(\lambda D \log(D))$ time for Merkle (dominated by writing down authentication paths), $O(D \log(D) + \lambda D)$ for Hash (dominated by computing and hashing the Reed-Solomon codewords), and $O(\lambda D \log^2(D))$ for FRIDA (dominated by writing down the opening proofs).

Concrete Efficiency. In Table 2, we compare the concrete efficiency of data availability sampling schemes, including FRIDA. We also show how efficiency metrics develop when increasing the data size in Figure 1. While FRIDA's main drawback is the total size of the encoding, it outperforms Hash in terms of commitment size and communication per query. From the perspective of a single client, these are the

	Scheme	Commitment [KB]	Encoding [MB]	Communication Costs Per Query [KB]	Total [MB]
$D = 1$ MB	Naive	0.03	1.00	1000.00	1.00
	Merkle	0.03	4.25	0.55	156.40
	Tensor	6.96	8.07	0.10	15.70
	Hash	256.00	4.00	2.00	1.76
	FRIDA	255.10	17.56	2.15	7.60
$D = 32$ MB	Naive	0.03	32.00	32000.00	32.00
	Merkle	0.03	176.00	0.71	7089.80
	Tensor	39.22	256.32	0.10	456.52
	Hash	1448.45	128.05	11.32	55.32
	FRIDA	464.83	1031.80	3.94	444.34
$D = 128$ MB	Naive	0.03	128.00	128000.00	128.00
	Merkle	0.03	768.00	0.77	32007.29
	Tensor	78.38	1024.01	0.10	1807.95
	Hash	2896.38	512.03	22.63	220.78
	FRIDA	495.81	4395.63	4.19	1892.81

Table 2: Concrete efficiency comparison of data availability sampling schemes. For given sizes $D = |\mathsf{data}|$ of the encoded data, we compare the commitment size, the encoding size, the communication complexity per query, and the total communication complexity such that data can be reconstructed with probability at least $1 - 2^{-40}$.

metrics of interest. We see that this improvement over Hash becomes significant especially for large data sizes, matching our asymptotic comparison. On the other hand, the main advantage of Hash is the total communication complexity, which is also in line with our asymptotic bounds. For all metrics, we see that FRIDA gets us closer to the efficiency of Tensor.

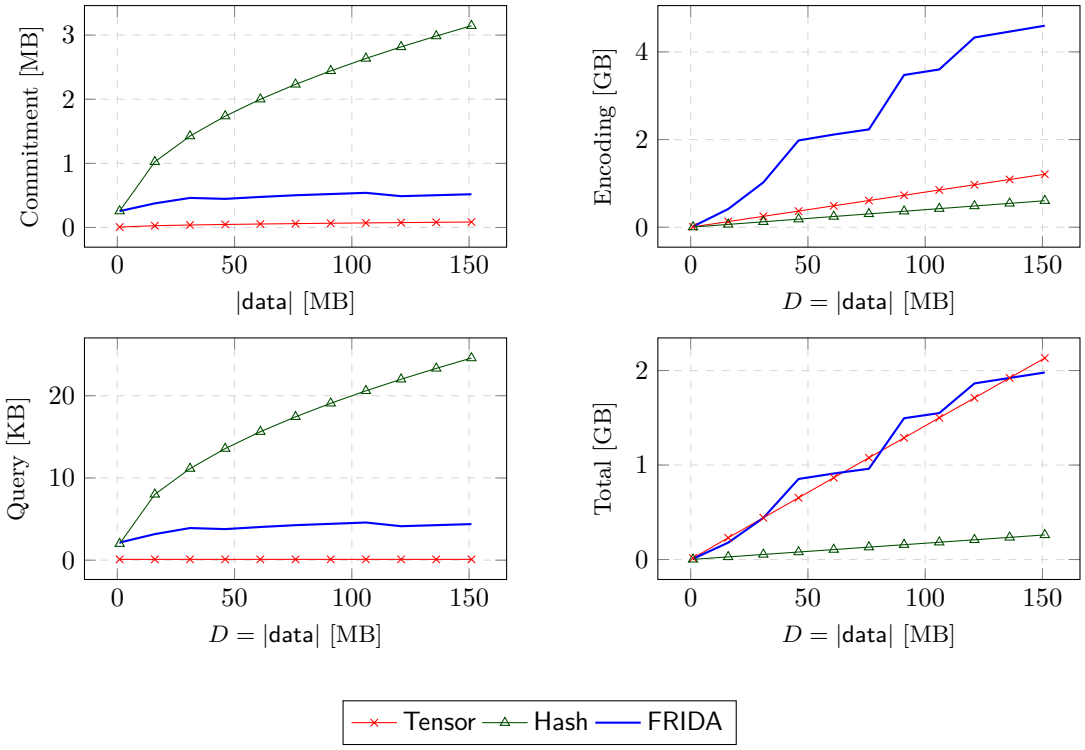

Figure 1: Plot of the efficiency of data availability sampling schemes. We plot the commitment size, encoding size, communication per query, and total communication needed to reconstruct data with probability at least $1 - 2^{-40}$ depending on the size of the encoded data $D = |\mathsf{data}|$. We omit the trivial baseline schemes Naive and Merkle.

References

[ABC+07] Giuseppe Ateniese, Randal C. Burns, Reza Curtmola, Joseph Herring, Lea Kissner, Zachary N. J. Peterson, and Dawn Song. Provable data possession at untrusted stores. In Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson, editors, *ACM CCS 2007*, pages 598–609. ACM Press, October 2007. (Cited on page 5.)

[ASBK21] Mustafa Al-Bassam, Alberto Sonnino, Vitalik Buterin, and Ismail Khoffi. Fraud and data availability proofs: Detecting invalid blocks in light clients. In Nikita Borisov and Claudia Díaz, editors, *Financial Cryptography and Data Security - 25th International Conference, FC 2021, Virtual Event, March 1-5, 2021, Revised Selected Papers, Part II*, volume 12675 of *Lecture Notes in Computer Science*, pages 279–298. Springer, 2021. (Cited on page 3.)

[BBHR18] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Fast reed-solomon interactive oracle proofs of proximity. In Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella, editors, *ICALP 2018*, volume 107 of *LIPIcs*, pages 14:1–14:17. Schloss Dagstuhl, July 2018. (Cited on page 4, 6, 9, 17, 18, 21, 22.)

[BBHR19] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable zero knowledge with no trusted setup. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 701–732. Springer, Heidelberg, August 2019. (Cited on page 4.)

[BCFL23] David Balbás, Dario Catalano, Dario Fiore, and Russell W. F. Lai. Chainable functional commitments for unbounded-depth circuits. In Guy N. Rothblum and Hoeteck Wee, editors, *Theory of Cryptography - 21st International Conference, TCC 2023, Taipei, Taiwan, November 29 - December 2, 2023, Proceedings, Part III*, volume 14371 of *Lecture Notes in Computer Science*, pages 363–393. Springer, 2023. (Cited on page 5.)

[BCG+16] Eli Ben-Sasson, Alessandro Chiesa, Ariel Gabizon, Michael Riabzev, and Nicholas Spooner. Short interactive oracle proofs with constant query complexity, via composition and sumcheck. Cryptology ePrint Archive, Report 2016/324, 2016. https://eprint.iacr.org/2016/324. (Cited on page 4.)

[BCG+19] Eli Ben-Sasson, Alessandro Chiesa, Lior Goldberg, Tom Gur, Michael Riabzev, and Nicholas Spooner. Linear-size constant-query IOPs for delegating computation. In Dennis Hofheinz and Alon Rosen, editors, *TCC 2019, Part II*, volume 11892 of *LNCS*, pages 494–521. Springer, Heidelberg, December 2019. (Cited on page 4.)

[BCI+20a] Eli Ben-Sasson, Dan Carmon, Yuval Ishai, Swastik Kopparty, and Shubhangi Saraf. Proximity gaps for reed-solomon codes. In *61st FOCS*, pages 900–909. IEEE Computer Society Press, November 2020. (Cited on page 4, 19, 25, 35.)

[BCI+20b] Eli Ben-Sasson, Dan Carmon, Yuval Ishai, Swastik Kopparty, and Shubhangi Saraf. Proximity gaps for reed-solomon codes. Cryptology ePrint Archive, Report 2020/654, 2020. https://eprint.iacr.org/2020/654. (Cited on page 19.)

[BCR+19] Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P. Ward. Aurora: Transparent succinct arguments for R1CS. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part I*, volume 11476 of *LNCS*, pages 103–128. Springer, Heidelberg, May 2019. (Cited on page 4.)

[BCS16] Eli Ben-Sasson, Alessandro Chiesa, and Nicholas Spooner. Interactive oracle proofs. In Martin Hirt and Adam D. Smith, editors, *TCC 2016-B, Part II*, volume 9986 of *LNCS*, pages 31–60. Springer, Heidelberg, October / November 2016. (Cited on page 8, 9, 11, 34.)

[BGK+23] Alexander R. Block, Albert Garreta, Jonathan Katz, Justin Thaler, Pratyush Ranjan Tiwari, and Michał Zając. Fiat-shamir security of fri and related snarks. Cryptology ePrint Archive, Paper 2023/1071, 2023. https://eprint.iacr.org/2023/1071. (Cited on page 17, 19.)

[BGKS20] Eli Ben-Sasson, Lior Goldberg, Swastik Kopparty, and Shubhangi Saraf. DEEP-FRI: Sampling outside the box improves soundness. In Thomas Vidick, editor, *ITCS 2020*, volume 151, pages 5:1–5:32. LIPIcs, January 2020. (Cited on page 4.)

[BNO21] Dan Boneh, Wilson Nguyen, and Alex Ozdemir. Efficient functional commitments: How to commit to private functions. Cryptology ePrint Archive, Report 2021/1342, 2021. https://eprint.iacr.org/2021/1342. (Cited on page 5.)

[CCH+19] Ran Canetti, Yilei Chen, Justin Holmgren, Alex Lombardi, Guy N. Rothblum, Ron D. Rothblum, and Daniel Wichs. Fiat-Shamir: from practice to theory. In Moses Charikar and Edith Cohen, editors, *51st ACM STOC*, pages 1082–1090. ACM Press, June 2019. (Cited on page 10.)

[CFT22] Dario Catalano, Dario Fiore, and Ida Tucker. Additive-homomorphic functional commitments and applications to homomorphic signatures. In Shweta Agrawal and Dongdai Lin, editors, *ASIACRYPT 2022, Part IV*, volume 13794 of *LNCS*, pages 159–188. Springer, Heidelberg, December 2022. (Cited on page 5.)

[CKW13] David Cash, Alptekin Küpçü, and Daniel Wichs. Dynamic proofs of retrievability via oblivious RAM. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 279–295. Springer, Heidelberg, May 2013. (Cited on page 5.)

[COS20] Alessandro Chiesa, Dev Ojha, and Nicholas Spooner. Fractal: Post-quantum and transparent recursive proofs from holography. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part I*, volume 12105 of *LNCS*, pages 769–793. Springer, Heidelberg, May 2020. (Cited on page 4.)

[CT05] Christian Cachin and Stefano Tessaro. Asynchronous verifiable information dispersal. In Pierre 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. (Cited on page 5.)

[dCP23] Leo de Castro and Chris Peikert. Functional commitments for all functions, with transparent setup and from SIS. In Carmit Hazay and Martijn Stam, editors, *EUROCRYPT 2023, Part III*, volume 14006 of *LNCS*, pages 287–320. Springer, Heidelberg, April 2023. (Cited on page 5.)

[DVW09] Yevgeniy Dodis, Salil P. Vadhan, and Daniel Wichs. Proofs of retrievability via hardness amplification. In Omer Reingold, editor, *TCC 2009*, volume 5444 of *LNCS*, pages 109–127. Springer, Heidelberg, March 2009. (Cited on page 5.)

[FKL18] Georg Fuchsbauer, Eike Kiltz, and Julian Loss. The algebraic group model and its applications. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part II*, volume 10992 of *LNCS*, pages 33–62. Springer, Heidelberg, August 2018. (Cited on page 26.)

[GW11] Craig Gentry and Daniel Wichs. Separating succinct non-interactive arguments from all falsifiable assumptions. In Lance Fortnow and Salil P. Vadhan, editors, *43rd ACM STOC*, pages 99–108. ACM Press, June 2011. (Cited on page 4.)

[Hab22] Ulrich Haböck. A summary on the FRI low degree test. Cryptology ePrint Archive, Report 2022/1216, 2022. https://eprint.iacr.org/2022/1216. (Cited on page 26.)

[HASW23] Mathias Hall-Andersen, Mark Simkin, and Benedikt Wagner. Foundations of data availability sampling. Cryptology ePrint Archive, Paper 2023/1079, 2023. https://eprint.iacr.org/2023/1079. (Cited on page 3, 4, 6, 7, 8, 9, 25, 26, 32, 33, 40.)

[HKLN20] Eduard Hauck, Eike Kiltz, Julian Loss, and Ngoc Khanh Nguyen. Lattice-based blind signatures, revisited. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part II*, volume 12171 of *LNCS*, pages 500–529. Springer, Heidelberg, August 2020. (Cited on page 9, 34.)

[JK07] Ari Juels and Burton S. Kaliski Jr. Pors: proofs of retrievability for large files. In Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson, editors, *ACM CCS 2007*, pages 584–597. ACM Press, October 2007. (Cited on page 5.)

[KZG10] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In Masayuki Abe, editor, *ASIACRYPT 2010*, volume 6477 of *LNCS*, pages 177–194. Springer, Heidelberg, December 2010. (Cited on page 4.)

[LRY16] Benoît Libert, Somindu C. Ramanna, and Moti Yung. Functional commitment schemes: From polynomial commitments to pairing-based accumulators from simple assumptions. In Ioannis Chatzigiannakis, Michael Mitzenmacher, Yuval Rabani, and Davide Sangiorgi, editors, *ICALP 2016*, volume 55 of *LIPIcs*, pages 30:1–30:14. Schloss Dagstuhl, July 2016. (Cited on page 5.)

[Mer88] Ralph C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *CRYPTO'87*, volume 293 of *LNCS*, pages 369–378. Springer, Heidelberg, August 1988. (Cited on page 9, 26.)

[NNT21] Kamilla Nazirkhanova, Joachim Neu, and David Tse. Information dispersal with provable retrievability for rollups. Cryptology ePrint Archive, Report 2021/1544, 2021. https://eprint.iacr.org/2021/1544. (Cited on page 5.)

[Rab89] Michael O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *J. ACM*, 36(2):335–348, 1989. (Cited on page 5.)

[SSP13] Elaine Shi, Emil Stefanov, and Charalampos Papamanthou. Practical dynamic proofs of retrievability. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 325–336. ACM Press, November 2013. (Cited on page 5.)

[Sta21] StarkWare. ethSTARK documentation. Cryptology ePrint Archive, Report 2021/582, 2021. https://eprint.iacr.org/2021/582. (Cited on page 26.)

[SW08] Hovav Shacham and Brent Waters. Compact proofs of retrievability. In Josef Pieprzyk, editor, *ASIACRYPT 2008*, volume 5350 of *LNCS*, pages 90–107. Springer, Heidelberg, December 2008. (Cited on page 5.)

[Val08] Paul Valiant. Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In Ran Canetti, editor, *TCC 2008*, volume 4948 of *LNCS*, pages 1–18. Springer, Heidelberg, March 2008. (Cited on page 9, 34.)

[VP19] Alexander Vlasov and Konstantin Panarin. Transparent polynomial commitment scheme with polylogarithmic communication complexity. Cryptology ePrint Archive, Report 2019/1020, 2019. https://eprint.iacr.org/2019/1020. (Cited on page 6.)

[WW23] Hoeteck Wee and David J. Wu. Succinct vector, polynomial, and functional commitments from lattices. In Carmit Hazay and Martijn Stam, editors, *EUROCRYPT 2023, Part III*, volume 14006 of *LNCS*, pages 385–416. Springer, Heidelberg, April 2023. (Cited on page 5.)

A Background on Data Availability Sampling

Here, we recall the formal definition of data availability sampling and the transformation from erasure code commitments to data availability sampling. The reader may consult [HASW23] for more background. We first present the definition of data availability sampling, taken verbally from [HASW23].

Definition 14 (Data Availability Sampling Scheme)**.** A data availability sampling scheme (DAS) with data alphabet Γ, encoding alphabet Σ, data length $K \in \mathbb{N}$, encoding length $N \in \mathbb{N}$, query complexity $Q \in \mathbb{N}$, and threshold $T \in \mathbb{N}$ is a tuple $\mathsf{DAS} = (\mathsf{Setup}, \mathsf{Encode}, \mathsf{V}, \mathsf{Ext})$ of algorithms with the following syntax:

- $\mathsf{Setup}(1^\lambda) \to \mathsf{par}$ is a PPT algorithm that takes as input the security parameter, and outputs system parameters par. All algorithms get par implicitly as input.
- $\mathsf{Encode}(\mathsf{data}) \to (\pi, \mathsf{com})$ is a deterministic polynomial time algorithm that takes as input data $\mathsf{data} \in \Gamma^K$ and outputs an encoding $\pi \in \Sigma^N$ and a commitment com.
- $\mathsf{V} = (\mathsf{V}_1, \mathsf{V}_2)$ is a pair of algorithms, where
  - $\mathsf{V}_1^{\pi,Q}(\mathsf{com}) \to \mathsf{tran}$ is a PPT algorithm that has Q-time oracle access to an encoding $\pi \in \Sigma^N$, gets as input a commitment com, and outputs a transcript tran, containing the Q queries to π and the respective responses.
  - $\mathsf{V}_2(\mathsf{com}, \mathsf{tran}) \to b$ is a deterministic polynomial time algorithm that takes as input a transcript tran, and outputs a bit $b \in \{0, 1\}$.
- $\mathsf{Ext}(\mathsf{com}, \mathsf{tran}_1, \ldots, \mathsf{tran}_\ell) \to \mathsf{data}/\bot$ is a deterministic polynomial time algorithm that takes as input a commitment com, a list of transcripts tran_i, and outputs data $\mathsf{data} \in \Gamma^K$ or an abort symbol $\bot$.

We require that the following properties are satisfied:

- **Completeness.** For any $\mathsf{par} \in \mathsf{Setup}(1^\lambda)$ and any integer $\ell = \mathsf{poly}(\lambda)$ with $\ell \geq T$, and all $\mathsf{data} \in \Gamma^K$, we have

$$\Pr\left[\forall i \in [\ell] : b_i = 1 \wedge \mathsf{data}' = \mathsf{data} \;\middle|\; \begin{array}{l} (\pi, \mathsf{com}) := \mathsf{Encode}(\mathsf{data}), \\ \forall i \in [\ell] : \mathsf{tran}_i \leftarrow \mathsf{V}_1^{\pi,Q}(\mathsf{com}), \\ \quad b_i := \mathsf{V}_2(\mathsf{com}, \mathsf{tran}_i), \\ \mathsf{data}' := \mathsf{Ext}(\mathsf{com}, \mathsf{tran}_1, \ldots, \mathsf{tran}_\ell) \end{array}\right] \geq 1 - \mathsf{negl}(\lambda).$$

- **Soundness.** For any stateful PPT algorithm $\mathcal{A}$ and any integer $\ell = \mathsf{poly}(\lambda)$ with $\ell \geq T$, the following advantage is negligible:

$$\mathsf{Adv}^{\mathsf{sound}}_{\mathcal{A},\ell,\mathsf{DAS}}(\lambda) := \Pr\left[\forall i \in [\ell] : b_i = 1 \wedge \mathsf{data}' = \bot \;\middle|\; \begin{array}{l} \mathsf{par} \leftarrow \mathsf{Setup}(1^\lambda),\ \mathsf{com} \leftarrow \mathcal{A}(\mathsf{par}), \\ (\mathsf{tran}_i)_{i=1}^{\ell} \leftarrow \mathsf{Interact}\left[\mathsf{V}_1, \mathcal{A}\right]_{Q,\ell}(\mathsf{com}), \\ \forall i \in [\ell] : b_i := \mathsf{V}_2(\mathsf{com}, \mathsf{tran}_i), \\ \mathsf{data}' := \mathsf{Ext}(\mathsf{com}, \mathsf{tran}_1, \ldots, \mathsf{tran}_\ell) \end{array}\right].$$

- **Consistency.** For any PPT algorithm $\mathcal{A}$ and any $\ell_1, \ell_2 = \mathsf{poly}(\lambda)$, the following advantage is negligible:

$$\mathsf{Adv}^{\mathsf{cons}}_{\mathcal{A},\ell_1,\ell_2,\mathsf{DAS}}(\lambda) := \Pr\left[\begin{array}{ll} & \mathsf{data}_1 \neq \bot \\ \wedge & \mathsf{data}_2 \neq \bot \\ \wedge & \mathsf{data}_1 \neq \mathsf{data}_2 \end{array} \;\middle|\; \begin{array}{l} \mathsf{par} \leftarrow \mathsf{Setup}(1^\lambda), \\ (\mathsf{com}, (\mathsf{tran}_{1,i})_{i=1}^{\ell_1}, (\mathsf{tran}_{2,i})_{i=1}^{\ell_2}) \leftarrow \mathcal{A}(\mathsf{par}), \\ \mathsf{data}_1 := \mathsf{Ext}(\mathsf{com}, \mathsf{tran}_{1,1}, \ldots, \mathsf{tran}_{1,\ell_1}), \\ \mathsf{data}_2 := \mathsf{Ext}(\mathsf{com}, \mathsf{tran}_{2,1}, \ldots, \mathsf{tran}_{2,\ell_2}) \end{array}\right].$$

Hall-Andersen, Simkin, and Wagner [HASW23] show how to generically turn an erasure code commitment for a code $\mathcal{C}$ into a data availability sampling scheme. For the formal transformation and its analysis, we refer to [HASW23]. Here, we only informally discuss this transformation how its resulting efficiency relates to the efficiency of the underlying erasure code commitment scheme. Intuitively, $\mathsf{Encode}(\mathsf{data})$

encodes data using the code $\mathcal{C}$ and commits to it using the erasure code commitment scheme. That is, the algorithm outputs an erasure code commitment com for data and an encoding π, where each symbol of π consists of a symbol of the codeword and the respective opening for the erasure code commitment. Clients determine a set of indices to query and accept if all openings are valid. For this work, we assume that indices are sampled uniformly and independently without replacement, while other variants of index sampling are also analyzed in [HASW23]. Given a set of transcripts, the extractor Ext first checks if all transcripts are accepting and enough indices are covered. More precisely, if the code has reception efficiency t and at least t symbols of the codeword have to be contained in the transcripts. If this is the case, the data can be reconstructed by the properties of the code. We informally summarize the efficiency of the sketched transformation in the following lemma.

Lemma 11 (Informal)**.** *Let $\mathcal{C}\colon \Gamma^k \to \Lambda^n$ be an erasure code and let CC be an erasure code commitment for $\mathcal{C}$ with reception efficiency $t \leq n$. Then, one can construct a data availability sampling scheme with the following characteristics:*

- ***Commitment.*** *The commitment is as output by CC.*
- ***Encoding.*** *The encoding contains n symbols and each symbol contains an element in Λ and an opening output by CC.*
- ***Threshold.*** *To reconstruct the data with overwhelming probability, one needs at least $\Omega(t + \lambda)$ samples of clients in total if t/n is constant. If $\mathcal{C}$ is the identity, one needs at least $\Omega(k\lambda + k \log k)$ samples.*

B Merkle Trees

Here, we make our definition of Merkle trees explicit. To recall, let $\mathsf{H}\colon \{0,1\}^* \to \{0,1\}^\lambda$ be a random oracle. We denote by $\mathsf{Root}^{\mathsf{H}}$ the algorithm that takes as input a sequence $x_1, \ldots, x_\ell \in \Sigma$ of ℓ symbols over some alphabet Σ, and outputs the Merkle root $\mathsf{root} \in \{0,1\}^\lambda$.

Definition 15 (Merkle Root)**.** Let Σ be a finite set (possibly depending on λ) and $\mathsf{H}\colon \{0,1\}^* \to \{0,1\}^\lambda$ be a random oracle. We define

$$\mathsf{Root}^{\mathsf{H}}(x_1) := \mathsf{H}(x_1),$$
$$\mathsf{Root}^{\mathsf{H}}(x_1, \ldots, x_\ell) := \mathsf{H}(\mathsf{Root}^{\mathsf{H}}(x_1, \ldots, x_{\lceil \ell/2 \rceil}), \mathsf{Root}^{\mathsf{H}}(x_{\lceil \ell/2 \rceil + 1}, \ldots, x_\ell)),$$

for any $\ell \in \mathbb{N}$ and any $x_1, \ldots, x_\ell \in \Sigma$.

Further, we denote by $\mathsf{Path}^{\mathsf{H}}$ the algorithm that takes as input $x_1, \ldots, x_\ell \in \Sigma$ as above and an index $j \in [\ell]$, and outputs an authentication path path for the jth position, i.e., for x_j. We assume that path contains the length ℓ of the underlying sequence, the position j, and the value x_i, and denote these by $\mathsf{LengthOf}^{\mathsf{H}}$, $\mathsf{PositionOf}^{\mathsf{H}}$, and $\mathsf{ValueOf}^{\mathsf{H}}$, respectively.

Definition 16 (Merkle Paths)**.** Let Σ be a finite set (possibly depending on λ) and $\mathsf{H}\colon \{0,1\}^* \to \{0,1\}^\lambda$ be a random oracle. For any $x_1, x_2 \in \Sigma$, we define

$$cop(x_1, 1) := \bot, \quad cop((x_1, x_2), 1) := (\mathsf{H}(x_2)), \quad cop((x_1, x_2), 2) := (\mathsf{H}(x_1)).$$

For any $\ell \in \mathbb{N}$ and any $x_1, \ldots, x_\ell \in \Sigma$, we define

$$cop((x_1, \ldots, x_\ell), j) := (\mathsf{Root}^{\mathsf{H}}(x_{\lceil \ell/2 \rceil + 1}, \ldots, x_\ell), cop(x_1, \ldots, x_{\lceil \ell/2 \rceil}, j))$$

if $1 \leq j \leq \lceil \ell/2 \rceil$. Otherwise, if $\lceil \ell/2 \rceil + 1 \leq j \leq \ell$, we define

$$cop((x_1, \ldots, x_\ell), j) := (\mathsf{Root}^{\mathsf{H}}(x_1, \ldots, x_{\lceil \ell/2 \rceil}, cop(x_{\lceil \ell/2 \rceil + 1}, \ldots, x_\ell, j - \lceil \ell/2 \rceil)).$$

Further, for any $\ell \in \mathbb{N}$ and any $x_1, \ldots, x_\ell \in \Sigma$, we define

$$\mathsf{Path}^{\mathsf{H}}((x_1, \ldots, x_\ell), j) := (\ell, j, x_j, cop((x_1, \ldots, x_\ell), j)).$$

For any such tuple $\mathsf{path} = (\ell, j, x_j, cop)$ we define $\mathsf{PositionOf}^{\mathsf{H}}(\mathsf{path}) := j$, $\mathsf{ValueOf}^{\mathsf{H}}(\mathsf{path}) := x_j$, and $\mathsf{LengthOf}^{\mathsf{H}}(\mathsf{path}) := \ell$.

We denote by $\mathsf{RootFromPath}^{\mathsf{H}}$ the algorithm that takes as input a path path recomputes the Merkle root as defined next.

Definition 17 (Recomputing Merkle Roots)**.** Let Σ be a finite set (possibly depending on λ) and $\mathsf{H}\colon \{0,1\}^* \to \{0,1\}^\lambda$ be a random oracle. For any $x \in \Sigma$ and $h \in \{0,1\}^\lambda$, we define

$$\begin{aligned}
\mathsf{RootFromPath}^{\mathsf{H}}(\mathsf{path}) &:= \mathsf{H}(x) && \text{for } \mathsf{path} = (1,1,x,\bot),\\
\mathsf{RootFromPath}^{\mathsf{H}}(\mathsf{path}) &:= \mathsf{H}(\mathsf{H}(x),h) && \text{for } \mathsf{path} = (2,1,x,(h)),\\
\mathsf{RootFromPath}^{\mathsf{H}}(\mathsf{path}) &:= \mathsf{H}(h,\mathsf{H}(x)) && \text{for } \mathsf{path} = (2,2,x,(h)).
\end{aligned}$$

Let $\ell \in \mathbb{N}$, $j \in [\ell]$, $x \in \Sigma$, $h_1,\dots,h_r \in \{0,1\}^\lambda$, and $\mathsf{path} = (\ell, j, x, (h_1,\dots,h_r))$. For $1 \le j \le \lceil \ell/2 \rceil$, we define

$$\mathsf{RootFromPath}^{\mathsf{H}}(\mathsf{path}) := \mathsf{H}(\mathsf{RootFromPath}^{\mathsf{H}}(\mathsf{path}'), h_1) \text{ for } \mathsf{path}' := (\lceil \ell/2 \rceil, j, x, (h_2,\dots,h_r)).$$

For $\lceil \ell/2 \rceil + 1 \le j \le \ell$, we define

$$\mathsf{RootFromPath}^{\mathsf{H}}(\mathsf{path}) := \mathsf{H}(h_1, \mathsf{RootFromPath}^{\mathsf{H}}(\mathsf{path}')) \text{ for } \mathsf{path}' := (\ell - \lceil \ell/2 \rceil, j - \lceil \ell/2 \rceil, x, (h_2,\dots,h_r)).$$

The next lemma states the correctness of these algorithms. We omit a proof as it follows easily by inspection.

Lemma 12. *Let Σ be a finite set (possibly depending on λ) and $\mathsf{H}\colon \{0,1\}^* \to \{0,1\}^\lambda$ be a random oracle. For any $\ell \in \mathbb{N}$, any $j \in [\ell]$, and any $x_1,\dots,x_\ell \in \Sigma$, we have*

$$\mathsf{Root}^{\mathsf{H}}(x_1,\dots,x_\ell) = \mathsf{RootFromPath}^{\mathsf{H}}(\mathsf{Path}^{\mathsf{H}}((x_1,\dots,x_\ell), j)).$$

In the following, we show that Merkle trees (in the random oracle model) are extractable, i.e., one can extract the underlying sequence of values from a given Merkle root, under certain conditions. This technique is (implicitly) used in several works, e.g., [Val08, BCS16, HKLN20].

Alg $\mathsf{ExtLeafs}(\mathsf{root}, \ell)$
01 **if** $h^{-1}[\mathsf{root}] = \bot$:
02 $\quad(\mathsf{leaf}_1,\dots,\mathsf{leaf}_\ell) := (\bot,\dots,\bot)$
03 $\quad$**return** $(\mathsf{leaf}_1,\dots,\mathsf{leaf}_\ell)$
04 **if** $\ell = 1$: **return** $h^{-1}[\mathsf{root}]$
05 **parse** $(\mathsf{root}_0, \mathsf{root}_1) := h^{-1}[\mathsf{root}]$
06 $\mathsf{leaf}_0 := \mathsf{ExtLeafs}(\mathsf{root}_0, \lceil \ell/2 \rceil)$
07 $\mathsf{leaf}_1 := \mathsf{ExtLeafs}(\mathsf{root}_1, \ell - \lceil \ell/2 \rceil)$
08 **return** $(\mathsf{leaf}_0, \mathsf{leaf}_1)$

Oracle $\mathsf{H}(x)$
09 **if** $h[x] = \bot$: $h[x] \xleftarrow{\$} \{0,1\}^\lambda$
10 $h^{-1}[h[x]] := x$
11 **return** $h[x]$

Oracle $\mathsf{SubRoot}(\mathsf{root}, \ell)$
12 **if** $(\mathsf{root}, \ell) \in \mathsf{Sub}$: **return**
13 $(\mathsf{leaf}_1,\dots,\mathsf{leaf}_\ell) := \mathsf{ExtLeafs}(\mathsf{root}, \ell)$
14 $\mathsf{Leafs}[\mathsf{root}, \ell] := (\mathsf{leaf}_1,\dots,\mathsf{leaf}_\ell)$

Oracle $\mathsf{SubPath}(\mathsf{root}, \mathsf{path})$
15 **parse** $\mathsf{path} = (\ell, j, x, \mathit{cop})$
16 **if** $(\mathsf{root}, \ell) \notin \mathsf{Sub}$: **return**
17 **if** $\mathsf{root} \neq \mathsf{RootFromPath}^{\mathsf{H}}(\mathsf{path})$:
18 $\quad$**return**
19 $(\mathsf{leaf}_1,\dots,\mathsf{leaf}_\ell) := \mathsf{Leafs}[\mathsf{root}]$
20 **if** $\mathsf{leaf}_j = \bot \vee \mathsf{leaf}_j \neq x$: $\mathsf{Bad} := 1$

Figure 2: Components used in Lemma 13: algorithm $\mathsf{ExtLeafs}$ for extraction of leafs from a given Merkle root, random oracle H, and the oracles $\mathsf{SubRoot}$ and $\mathsf{SubPath}$.

Lemma 13. *Let Σ be a finite set (possibly depending on λ) and $\mathsf{H}\colon \{0,1\}^* \to \{0,1\}^\lambda$ be a random oracle. Consider algorithm $\mathsf{ExtLeafs}$ and oracles $\mathsf{SubRoot}$ and $\mathsf{SubPath}$ in Figure 2. For any algorithm $\mathcal{A}$, consider the experiment of running $\mathcal{A}$ with oracle access to H, $\mathsf{SubRoot}$, and $\mathsf{SubPath}$, assume that oracle H is queried at most Q times in total, and assume that $\mathsf{SubRoot}$ is queried at most R times in total. Then, we have*

$$\Pr[\mathsf{Bad} = 1] \le \frac{2Q^2 + QR}{2^\lambda}.$$

Proof. For simplicity, we denote by Bad the event that $\mathsf{Bad} = 1$. Now, we define three events:

- Event Coll: This event occurs, if on a query $\mathsf{H}(x)$ for which $h[x] = \bot$, $h[x]$ is sampled and we have $h^{-1}[h[x]] \neq \bot$.
- Event Chain: This event occurs, if on a query $\mathsf{H}(x)$ for which $h[x] = \bot$, $h[x]$ is sampled and there exists a $v \in \{0,1\}^\lambda$ such that $h[h[x], v] \neq \bot$ or $h[v, h[x]] \neq \bot$.
- Event $\mathsf{RootGuess}$: This event occurs, if $\mathcal{A}$ makes a query $\mathsf{SubRoot}(\mathsf{root}, \ell)$ such that at that time $h^{-1}[\mathsf{root}] = \bot$, but later, a query $\mathsf{H}(x)$ evaluates to root.

We bound the probability of these events: First, notice that Coll occurs if two queries to H collide. This happens with probability at most $1/2^\lambda$ for each fixed pair of queries to H, and so the probability of Coll is at most $Q_\mathsf{H}^2/2^\lambda$ by a union bound. Second, Chain occurs if there is a query $\mathsf{H}(x)$ such that a later query $\mathsf{H}(x')$ evaluates to a length λ prefix or suffix of x. Again, for each fixed pair of queries, this happens with probability at most $1/2^\lambda$. Third, for each fixed query to $\mathsf{SubRoot}$ and each fixed query to $\mathsf{H}(x)$, the probability that $\mathsf{RootGuess}$ occurs for this pair is $1/2^\lambda$. A union bound yields

$$\Pr\left[\mathsf{Coll} \vee \mathsf{Chain} \vee \mathsf{RootGuess}\right] \leq \frac{2Q^2 + QR}{2^\lambda}.$$

In the following let root^* and $\mathsf{path}^* = (\ell^*, j^*, x^*, cop^*)$ be the (first) Merkle root and Merkle path that made the game set $\mathsf{Bad} = 1$, i.e., $\mathcal{A}$ first made a query $\mathsf{SubRoot}(\mathsf{root}^*, \ell^*)$ and at that point the game executed $(\mathsf{leaf}_1^*, \ldots, \mathsf{leaf}_{\ell^*}^*) := \mathsf{ExtLeafs}(\mathsf{root}^*, \ell^*)$, and later $\mathcal{A}$ made a query $\mathsf{SubPath}(\mathsf{root}^*, \mathsf{path}^*)$ with $\mathsf{leaf}_{j^*}^* = \bot$ or $\mathsf{leaf}_{j^*}^* \neq x^*$. We define three more events:

- Event $\mathsf{RootBot}$: This event occurs, if during the query $\mathsf{SubRoot}(\mathsf{root}^*, \ell^*)$, we have $h^{-1}[\mathsf{root}^*] = \bot$.
- Event $\mathsf{LeafBot}$: This event occurs, if we have $\mathsf{leaf}_{j^*}^* = \bot$.
- Event Incons: This event occurs, if we have $\mathsf{leaf}_{j^*}^* \neq \bot$ and $\mathsf{leaf}_{j^*}^* \neq x^*$.

Note that

$$\Pr\left[\mathsf{Bad}\right] \leq \Pr\left[\mathsf{Bad} \wedge \mathsf{RootBot}\right] + \Pr\left[\mathsf{Bad} \wedge \mathsf{LeafBot}\right] + \Pr\left[\mathsf{Bad} \wedge \mathsf{Incons}\right],$$

and the same holds if we condition on $\neg(\mathsf{RootGuess} \vee \mathsf{Coll} \vee \mathsf{Chain})$. We bound the probability of these terms individually. First, if $\mathsf{Bad} \wedge \mathsf{RootBot}$ occurs, it is easy to see that $\mathsf{RootGuess}$ has to occur, and thus

$$\Pr\left[\mathsf{Bad} \wedge \mathsf{RootBot} \mid \neg(\mathsf{RootGuess} \vee \mathsf{Coll} \vee \mathsf{Chain})\right] = 0.$$

Second, if $\mathsf{Bad} \wedge \mathsf{LeafBot}$ occurs, we can observe that Chain has to occur, and thus

$$\Pr\left[\mathsf{Bad} \wedge \mathsf{LeafBot} \mid \neg(\mathsf{RootGuess} \vee \mathsf{Coll} \vee \mathsf{Chain})\right] = 0.$$

Third, if $\mathsf{Bad} \wedge \mathsf{Incons}$ occurs, we know that Coll has to occur, and thus

$$\Pr\left[\mathsf{Bad} \wedge \mathsf{Incons} \mid \neg(\mathsf{RootGuess} \vee \mathsf{Coll} \vee \mathsf{Chain})\right] = 0.$$

In combination, we get what we wanted to show. □

C Extension: Batched FRI

To reduce the encoding size of our data availability sampling scheme from FRI, we extend our analysis to batched FRI [BCI+20a]. In this setting, a batch of functions is first folded into a single function G_0, to which regular FRI is then applied. We encourage the reader to first study Section 4 before reading this section.

C.1 Construction

We describe batched FRI using interleaved Reed-Solomon codes, which we first recall. Then, we give an informal overview and present the formal construction.

Interleaved Reed-Solomon Codes. Batched FRI is an IOPP for an interleaved Reed-Solomon code, which we define next. First, consider the Reed-Solomon code $\mathcal{RS}[d, \mathcal{L}, \mathbb{F}]$ as in Section 4, where $\mathbb{F}$ is a finite field, $d \in \mathbb{N}$ is a (strict) degree upper bound, and $\mathcal{L} \subseteq \mathbb{F}$ is an evaluation domain. Then, a tuple $\mathbf{G} = (\mathbf{G}_j)_{j=1}^B$ of $B \in \mathbb{N}$ functions $\mathbf{G}_1, \ldots, \mathbf{G}_B \colon \mathcal{L} \to \mathbb{F}$ is in the interleaved Reed-Solomon code $\mathcal{RS}[d, \mathcal{L}, \mathbb{F}]^{\equiv B}$ if and only if $\mathbf{G}_j \in \mathcal{RS}[d, \mathcal{L}, \mathbb{F}]$ for all $j \in [B]$. We emphasize that in the context of the interleaved code, a symbol of such a word $\mathbf{G} = (\mathbf{G}_j)_{j=1}^B$ is given as $(\mathbf{G}_j(s))_{j=1}^B \in \mathbb{F}^B$ where $s \in \mathcal{L}$, i.e., the code $\mathcal{RS}[d, \mathcal{L}, \mathbb{F}]^{\equiv B}$ has length $|\mathcal{L}|$ and each symbol consists of B field elements. Observe that with this, the Hamming distance of two such words $\mathbf{G} = (\mathbf{G}_j)_{j=1}^B$ and $\mathbf{G}' = (\mathbf{G}'_j)_{j=1}^B$ is given as

$$\delta(\mathbf{G}, \mathbf{G}') = \frac{1}{|\mathcal{L}|}\{s \in \mathcal{L} \mid \exists j \in [B] : \mathbf{G}_j(s) \neq \mathbf{G}'_j(s)\}.$$

Especially, the distance of $\mathbf{G}$ to $\mathcal{RS}[d, \mathcal{L}, \mathbb{F}]^{\equiv B}$ can in general be larger than the maximum over all individual distances $\delta(\mathbf{G}_j, \mathcal{RS}[d, \mathcal{L}, \mathbb{F}])$. Finally, one can see that if $\mathcal{RS}[d, \mathcal{L}, \mathbb{F}]$ has rate ρ and unique decoding radius $\delta^* = (1-\rho)/2$, then the code $\mathcal{RS}[d, \mathcal{L}, \mathbb{F}]^{\equiv B}$ also has rate ρ and unique decoding radius δ^*.

Overview. Let $B \in \mathbb{N}$ be a batching parameter. Batched FRI is an IOPP for the interleaved code $\mathcal{RS}[d_0, \mathcal{L}_0, \mathbb{F}]^{\equiv B} = \mathcal{C}_0{}^{\equiv B}$, where we rely on the same notation and conditions as in Section 4. That is, the prover claims that $\mathbf{G} = (\mathbf{G}_j)_{j=1}^B$ is in the interleaved code $\mathcal{RS}[d_0, \mathcal{L}_0, \mathbb{F}]^{\equiv B}$, where $\mathbf{G}_j \colon \mathcal{L}_0 \to \mathbb{F}$ for all $j \in [B]$. In the first step, the verifier sends a random $\xi \xleftarrow{\$} \mathbb{F}$ as an initial challenge. The prover responds with $G_0 \colon \mathcal{L}_0 \to \mathbb{F}$ where $G_0 = \sum_{j=1}^B \xi^{j-1}\mathbf{G}_j$. Then, the prover and verifier run the FRI IOPP with G_0. As an additional consistency check in the query phase, the verifier checks that $G_0(s_0) = \sum_{j=1}^B \xi^{j-1}\mathbf{G}_j(s_0)$, where – to recall – $s_0 \in \mathcal{L}_0$ is the random point that the FRI verifier samples in the query phase. The reader shall also recall the definition of the domains $\mathcal{L}_i$ and codes $\mathcal{C}_i = \mathcal{RS}[d_i, \mathcal{L}_i, \mathbb{F}]$ from Section 4. With this in mind, a transcript of the batched FRI IOPP has the form $T = (\mathbf{G}, \xi, G_0, \rho_1, G_0, \ldots, G_r, s_0)$, where $\rho_1, \ldots, \rho_r \in \mathbb{F}$ and $G_i \colon \mathcal{L}_i \to \mathbb{F}$.

Construction. To formally specify the IOPP, let P and V be the FRI prover and verifier from Section 4. We specify the batched FRI prover $\mathsf{P}_{\mathsf{Batch}}$ and the batched FRI verifier $\mathsf{V}_{\mathsf{Batch}}$ as follows, where $\xi \in \mathbb{F}$, $\rho_1, \ldots, \rho_r \in \mathbb{F}$, and $s_0 \in \mathcal{L}_0$:

- $\mathsf{P}_{\mathsf{Batch}}(\mathbf{G}, \xi) \to G_0$ for $i \in \{1, \ldots, r\}$:
  1. Set $G_0 \colon \mathcal{L}_0 \to \mathbb{F}$ where $G_0 := \sum_{j=1}^B \xi^{j-1}\mathbf{G}_j$.
- $\mathsf{P}_{\mathsf{Batch}}(\mathbf{G}, \xi, G_0, \rho_1, \ldots, \rho_i) \to G_i$ for $i \in \{1, \ldots, r\}$:
  1. Run $G_i \leftarrow \mathsf{P}(G_0, \rho_1, \ldots, \rho_i)$.
- $\mathsf{V}_{\mathsf{Batch}}^{\mathbf{G}, G_0, G_1, \ldots, G_r}(\xi, \rho_1, \ldots, \rho_r, s_0) \to b$:
  1. Run $b_0 := \mathsf{V}^{G_0, G_1, \ldots, G_r}(\rho_1, \ldots, \rho_r, s_0)$.
  2. If $G_0(s_0) = \sum_{j=1}^B \xi^{j-1}\mathbf{G}_j(s_0)$, set $b_1 := 1$. Otherwise, set $b_1 := 0$.
  3. Set $b := b_0 \wedge b_1$.

C.2 Analysis

The reader can easily verify completeness, and that batched FRI is non-adaptive (see Definition 7) and query-selectable (see Definition 8).

Proof of Opening-Consistency. Our goal is to prove opening-consistency for batched FRI, as stated in Theorem 4. For that, we first define a suitable lucky set $\mathsf{LuckyBatch}$ and a bad set $\mathsf{BadBatch}$. And then prove the properties needed for opening-consistency in Lemmata 14 to 17. Combining Lemmata 14 to 17, we get Theorem 4.

Definition 18 (Lucky Set for Batched FRI)**.** Consider the batched FRI IOPP and let $\delta^* = (1-\rho)/2$ be the unique decoding radius. We define the set $\mathsf{LuckyBatch} = \mathsf{Lucky}' \cup \mathsf{LuckyBatchColl} \cup \mathsf{LuckyBatchDist}$ of partial prover-turn transcripts as follows:

- **Inherited.** A partial prover-turn transcript $(\mathbf{G}, \xi, G_0, \rho_1, \ldots, G_{i-1}, \rho_i)$ is in Lucky', if and only if $(G_0, \rho_1, \ldots, G_{i-1}, \rho_i) \in \mathsf{Lucky}$, where Lucky is defined as in Definition 12.
- **Lucky Batch Collision.** A partial prover-turn transcript $(\mathbf{G}, \xi)$ with $\mathbf{G} = (\mathbf{G}_j)_{j=1}^B$ is in $\mathsf{LuckyBatchColl}$ if and only if the following two properties hold:
  1. $\mathbf{G}$ is within the unique decoding radius, i.e., $\delta\left(\mathbf{G}, {\mathcal{C}_0}^{\equiv B}\right) \leq \delta^*$. Let $\mathbf{G}^* = (\mathbf{G}_j^*)_{j=1}^B \in {\mathcal{C}_0}^{\equiv B}$ be the unique closest codeword.
  2. There is an $s_0 \in \mathcal{L}_0$ such that $(\mathbf{G}_j^*(s_0))_{j=1}^B \neq (\mathbf{G}_j(s_0))_{j=1}^B$ but
$$\sum_{j=1}^{B} \xi^{j-1} \mathbf{G}_j^*(s_0) = \sum_{j=1}^{B} \xi^{j-1} \mathbf{G}_j(s_0).$$
- **Lucky Batch Distortion.** A partial prover-turn transcript $(\mathbf{G}, \xi)$ with $\mathbf{G} = (\mathbf{G}_j)_{j=1}^B$ is in $\mathsf{LuckyBatchDist}$ if and only if (a) $\delta\left(\mathbf{G}, {\mathcal{C}_0}^{\equiv B}\right) > \delta^*$ and $\delta\left(\sum_{j=1}^B \xi^{j-1}\mathbf{G}_j, \mathcal{C}_0\right) \leq \delta^*$ or (b) $\delta\left(\sum_{j=1}^B \xi^{j-1}\mathbf{G}_j, \mathcal{C}_0\right) < \delta\left(\mathbf{G}, {\mathcal{C}_0}^{\equiv B}\right) \leq \delta^*$.

Definition 19 (Bad Set for Batched FRI)**.** Consider the batched FRI IOPP and let $\delta^* = (1-\rho)/2$ be the unique decoding radius. We define the set $\mathsf{BadBatch}$ of partial transcripts as follows. A partial transcript $(\mathbf{G}, \xi, G_0, \rho_1, G_0, \ldots, G_r)$ with $\mathbf{G} = (\mathbf{G}_j)_{j=1}^B$ is in Bad, if and only if no prover-turn prefix of it is in $\mathsf{LuckyBatch}$, and at least one of the following properties holds:

1. We have $(G_0, \rho_1, G_0, \ldots, G_r) \in \mathsf{Bad}$, where Bad is defined as in Definition 13, or
2. We have $\delta\left(\mathbf{G}, {\mathcal{C}_0}^{\equiv B}\right) > \delta^*$, or
3. We have $\delta\left(\mathbf{G}, {\mathcal{C}_0}^{\equiv B}\right) \leq \delta^*$ and $\delta^{\mathsf{B}}\left(G_0, \mathcal{C}_0\right) \leq \delta^*$, but $G_0^* \neq \sum_{j=1}^B \xi^{j-1}\mathbf{G}_j^*$, where $G_0^* \in \mathcal{C}_0$ and $\mathbf{G}^* = (\mathbf{G}_j^*)_{j=1}^B \in {\mathcal{C}_0}^{\equiv B}$ denote the unique closest codewords of G_0 and $\mathbf{G}$, respectively.

Lemma 14 (No Luck)**.** *The batched FRI IOPP satisfies the* no luck *property of opening-consistency (Definition 10) with* $\mathsf{LuckyBatch}$ *as in Definition 18 and* $\epsilon_1 \leq 2(\max\{F, B\} - 1)|\mathcal{L}_0|/|\mathbb{F}|$.

Proof. Consider a partial verifier-turn transcript T and the experiment of sampling a random verifier challenge. We have to give an upper bound on the probability that extending T with this challenge is in the lucky set $\mathsf{LuckyBatch}$ as in Definition 18. By definition of $\mathsf{LuckyBatch}$, we can consider two cases depending on the structure of T. Then, we can take the maximum of the two bounds.

Case 1: Inherited. In the first case, we have $T = (\mathbf{G}, \xi, G_0, \rho_1, \ldots, G_{i-1})$ and the challenge $\rho_i \xleftarrow{\$} \mathbb{F}$ is sampled. Then $T \circ \rho_i \in \mathsf{LuckyBatch}$ if and only if $T' \circ \rho_i \in \mathsf{Lucky}$, where $T' = (G_0, \rho_1, \ldots, G_{i-1})$ and Lucky is defined as in Definition 12. Using the analysis in Lemma 5, we get that the probability that the extension is in $\mathsf{LuckyBatch}$ is at most $2(F-1)|\mathcal{L}_0|/|\mathbb{F}|$.

Case 2: Batching Step. In the second case, we have $T = (\mathbf{G})$ and the challenge $\xi \xleftarrow{\$} \mathbb{F}$ is sampled, i.e., we consider the initial batching step. We have to bound the probability that $T \circ \xi$ is in $\mathsf{LuckyBatchColl}$ or $\mathsf{LuckyBatchDist}$. Using a union bound, we get

$$\begin{aligned}\Pr_{\xi}\left[(\mathbf{G}, \xi) \in \mathsf{LuckyBatchColl} \cup \mathsf{LuckyBatchDist}\right] \leq & \Pr_{\xi}\left[(\mathbf{G}, \xi) \in \mathsf{LuckyBatchColl}\right] \\ & + \Pr_{\xi}\left[(\mathbf{G}, \xi) \in \mathsf{LuckyBatchDist}\right].\end{aligned}$$

Claim. We have $\Pr_{\xi}\left[(\mathbf{G}, \xi) \in \mathsf{LuckyBatchColl}\right] \leq (B-1)|\mathcal{L}_0|/|\mathbb{F}|$.

Proof of Claim. If $\mathbf{G}$ is not within the unique decoding radius from ${\mathcal{C}_0}^{\equiv B}$, then we are done by definition of $\mathsf{LuckyBatchColl}$. So, assume $\delta\left(\mathbf{G}, {\mathcal{C}_0}^{\equiv B}\right) \leq \delta^*$. Let $\mathbf{G}^* = (\mathbf{G}_j^*)_{j=1}^B \in {\mathcal{C}_0}^{\equiv B}$ be the unique closest codeword. Now, we have to bound the probability that there is an $s_0 \in \mathcal{L}_0$ such that $(\mathbf{G}_j^*(s_0))_{j=1}^B \neq (\mathbf{G}_j(s_0))_{j=1}^B$ but $\sum_{j=1}^B \xi^{j-1}\mathbf{G}_j^*(s_0) = \sum_{j=1}^B \xi^{j-1}\mathbf{G}_j(s_0)$. We do this using a union bound over all $s_0 \in \mathcal{L}_0$. Hence, let $s_0 \in \mathcal{L}_0$ be fixed and assume that $(\mathbf{G}_j^*(s_0))_{j=1}^B \neq (\mathbf{G}_j(s_0))_{j=1}^B$. Now, consider the process of sampling $\xi \xleftarrow{\$} \mathbb{F}$ at random. As $(\mathbf{G}_j^*(s_0))_{j=1}^B \neq (\mathbf{G}_j(s_0))_{j=1}^B$, the expressions $\sum_{j=1}^B \xi^{j-1}\mathbf{G}_j^*(s_0)$ and $\sum_{j=1}^B \xi^{j-1}\mathbf{G}_j(s_0)$ are two different polynomials of degree at most $B-1$ in ξ, which means that their evaluation at a random ξ is equal with probability at most $(B-1)/|\mathbb{F}|$, finishing the proof of this claim.

Claim. We have $\Pr_\xi\left[(\mathbf{G}, \xi) \in \mathsf{LuckyBatchDist}\right] \leq (B-1)|\mathcal{L}_0|/|\mathbb{F}|$.

Proof of Claim. This follows directly from the correlated agreement lemma (Lemma 3) with $K := B$, code $\mathcal{V} := \mathcal{C}_0$, and functions $u_0 := \mathbf{G}_1, \ldots, u_{B-1} := \mathbf{G}_B$. □

Lemma 15 (Bad is Rejected)**.** *The batched FRI IOPP satisfies the* bad is rejected *property of opening-consistency (Definition 10) with* $\mathsf{BadBatch}$ *as in Definition 19 and* $\epsilon_2 \leq 1 - \delta^*$.

Proof. We consider a partial transcript $T = (\mathbf{G}, \xi, G_0, \rho_1, G_0, \ldots, G_r)$ such that $T \in \mathsf{BadBatch}$, with $\mathsf{BadBatch}$ as in Definition 19. For the random experiment of sampling $s_0 \xleftarrow{\$} \mathcal{L}_0$ and completing T with s_0, we need to upper bound the probability that the verifier $\mathsf{V}_{\mathsf{Batch}}$ accepts the completed transcript $T \circ s_0$. Let $T' := (G_0, \rho_1, G_0, \ldots, G_r)$ and let Bad be as defined as in Definition 13. According to the definition of $\mathsf{BadBatch}$, we can consider two cases, depending on whether T is in the bad set because of $T' \in \mathsf{Bad}$. In both cases, we show that the probability that the verifier accepts (over the random choice of s_0) is at most $1 - \delta^*$.

Case 1. $T' \in \mathsf{Bad}$. By definition, if the batched verifier $\mathsf{V}_{\mathsf{Batch}}$ accepts T, then in particular the verifier V accepts T'. Therefore, we can apply the *bad is rejected* property of the non-batched FRI (Lemma 8) to get the desired bound.

Case 2. $T' \notin \mathsf{Bad}$. For this case, we closely follow the proof strategy used to prove Lemma 7. To this end, we first fix notation. Let $G_0^* \in \mathcal{C}_0$ be the unique closest codeword for G_0, which exists as $T' \notin \mathsf{Bad}$. Write $\mathbf{G} = (\mathbf{G}_j)_{j=1}^B$. Further, define the following sets:

$$\mathsf{Dis} := \{s_0 \in \mathcal{L}_0 \mid G_0^*(s_0) \neq G_0(s_0)\}$$

and

$$\mathsf{Find} := \left\{ s_0 \in \mathcal{L}_0 \;\middle|\; \sum_{j=1}^B \xi^{j-1}\mathbf{G}_j(s_0) \neq G_0(s_0) \right\}.$$

Now, we show three claims which in combination give the desired bound.

Claim. We have $\delta\left(G_0^*, \sum_{j=1}^B \xi^{j-1}\mathbf{G}_j\right) \geq \delta^*$.

Proof of Claim. Recall that $T \in \mathsf{BadBatch}$ but $T' \notin \mathsf{Bad}$. By definition of $\mathsf{BadBatch}$, we know that no prefix of T is in $\mathsf{LuckyBatch}$ and we are in one of two cases holds: in the first case, we have $\delta\left(\mathbf{G}, {\mathcal{C}_0}^{\equiv B}\right) > \delta^*$. Then, by definition of $\mathsf{LuckyBatch}$ (specifically, $\mathsf{LuckyBatchDist}$), we have

$$\delta\left(\sum_{j=1}^B \xi^{j-1}\mathbf{G}_j, G_0^*\right) \geq \delta\left(\sum_{j=1}^B \xi^{j-1}\mathbf{G}_j, \mathcal{C}_0\right) > \delta^*,$$

finishing the proof of the claim for this case. In the second case, we have $\delta\left(\mathbf{G}, {\mathcal{C}_0}^{\equiv B}\right) \leq \delta^*$ but $G_0^* \neq \sum_{j=1}^B \xi^{j-1}\mathbf{G}_j^*$, where $\mathbf{G}^* = (\mathbf{G}_j^*)_{j=1}^B \in {\mathcal{C}_0}^{\equiv B}$ is the unique closest codeword of $\mathbf{G}$. As both $G_0^* \in \mathcal{C}_0$

and $\sum_{j=1}^{B} \xi^{j-1}\mathbf{G}_j^* \in \mathcal{C}_0$, we know that

$$\begin{aligned}1-\rho \leq \delta\left(G_0^*, \sum_{j=1}^{B} \xi^{j-1}\mathbf{G}_j^*\right) &\leq \delta\left(G_0^*, \sum_{j=1}^{B} \xi^{j-1}\mathbf{G}_j\right) + \delta\left(\sum_{j=1}^{B} \xi^{j-1}\mathbf{G}_j, \sum_{j=1}^{B} \xi^{j-1}\mathbf{G}_j^*\right) \\ &\leq \delta\left(G_0^*, \sum_{j=1}^{B} \xi^{j-1}\mathbf{G}_j\right) + \delta\left(\mathbf{G}, \mathbf{G}^*\right) \\ &\leq \delta\left(G_0^*, \sum_{j=1}^{B} \xi^{j-1}\mathbf{G}_j\right) + \delta^*.\end{aligned}$$

By rearranging this inequality and using $1-\rho-\delta^* = \delta^*$, we obtain the claimed bound.

Claim. We have $|\mathsf{Dis} \cup \mathsf{Find}|/|\mathcal{L}_0| \geq \delta\left(G_0^*, \sum_{j=1}^{B} \xi^{j-1}\mathbf{G}_j\right)$.

Proof of Claim. To prove the claim, we argue that every position $s \in |\mathcal{L}_0|$ for which G_0^* and $\sum_{j=1}^{B} \xi^{j-1}\mathbf{G}_j$ disagree must be in Dis or in Find. This can be seen as follows: if $s_0 \notin \mathsf{Dis}$ and $s_0 \notin \mathsf{Find}$ then by definition of these sets we have

$$G_0^*(s_0) = G_0(s_0) = \sum_{j=1}^{B} \xi^{j-1}\mathbf{G}_j(s_0).$$

Claim. We have $\Pr_{s_0 \xleftarrow{\boxtimes} \mathcal{L}_0}\left[\mathsf{V}_{\mathsf{Batch}}^{\mathbf{G},G_0,G_1,\ldots,G_r}(\xi,\rho_1,\ldots,\rho_r,s_0)=1\right] \leq 1-(|\mathsf{Dis}\cup\mathsf{Find}|/|\mathcal{L}_0|)$.

Proof of Claim. If the event we have to bound occurs, i.e., the verifier $\mathsf{V}_{\mathsf{Batch}}$ accepts, then clearly $s_0 \notin \mathsf{Find}$. Further, we claim that $s_0 \notin \mathsf{Dis}$. This is because we can invoke Lemma 6 with $i^* = 0$. All conditions of Lemma 6 are satisfied as we assume $T' \notin \mathsf{Bad}$. Now, Lemma 6 states that if $s_0 \notin \mathsf{Dis}$, then the verifier rejects. So, we have

$$\begin{aligned}\Pr_{s_0 \xleftarrow{\boxtimes} \mathcal{L}_0}\left[\mathsf{V}_{\mathsf{Batch}}^{\mathbf{G},G_0,G_1,\ldots,G_r}(\xi,\rho_1,\ldots,\rho_r,s_0)=1\right] &\leq \Pr_{s_0 \xleftarrow{\boxtimes} \mathcal{L}_0}\left[s_0 \notin \mathsf{Dis} \wedge s_0 \notin \mathsf{Find}\right] \\ &\leq 1 - \frac{|\mathsf{Dis}\cup\mathsf{Find}|}{|\mathcal{L}_0|}.\end{aligned}$$

□

Lemma 16 (Suitable is Close)**.** *The batched FRI IOPP satisfies the* suitable is close *property of opening-consistency (Definition 10) with* BadBatch *as in Definition 19 and* LuckyBatch *as in Definition 18.*

Proof. Consider a suitable transcript $T = (\mathbf{G}, \xi, G_0, \rho_1, G_0, \ldots, G_r)$ with respect to sets $\mathsf{BadBatch}$ as in Definition 19 and $\mathsf{LuckyBatch}$ as in Definition 18. To recall, according to the definition of suitable transcripts (Definition 9), this means that T is not in $\mathsf{BadBatch}$ and no prefix of T is in $\mathsf{LuckyBatch}$. By definition of $\mathsf{BadBatch}$, we therefore know that $\delta\left(\mathbf{G}, \mathcal{C}_0^{\equiv B}\right) \leq \delta^*$, which is exactly what we have to show. □

Lemma 17 (Inconsistent is Rejected)**.** *The batched FRI IOPP satisfies the* inconsistent is rejected *property of opening-consistency (Definition 10) with* BadBatch *as in Definition 19 and* LuckyBatch *as in Definition 18.*

Proof. We consider a suitable transcript $T = (\mathbf{G}, \xi, G_0, \rho_1, G_0, \ldots, G_r)$ with respect to sets $\mathsf{BadBatch}$ as in Definition 19 and $\mathsf{LuckyBatch}$ as in Definition 18. This means that T is not in $\mathsf{BadBatch}$ and no prefix of T is in $\mathsf{LuckyBatch}$. Recalling the definition of the *inconsistent is rejected* property (Definition 10), we need to consider completing T with $s_0 \in \mathcal{L}_0$ such that there is a query $x \in \mathcal{Q}_0(T \circ s_0) = \{s_0\}$ that the verifier issues for the complete transcript $T \circ s_0$ such that $\mathbf{G}^*$ and $\mathbf{G}$ differ on that query, where $\mathbf{G}^* \in \mathcal{C}_0^{\equiv B}$ is the unique closest codeword for $\mathbf{G}$. We have seen in Lemma 16 that $\mathbf{G}^*$ exists. Even more, by definition of $\mathsf{BadBatch}$, we also know that a unique closest codeword $G_0^* \in \mathcal{C}_0$ for G_0 exists. Writing $\mathbf{G}^* = (\mathbf{G}_j^*)_{j=1}^{B}$

and $\mathbf{G} = (\mathbf{G}_j)_{j=1}^B$, this means we assume that there is a $j \in [B]$ such that $\mathbf{G}_j^*(s_0) \neq \mathbf{G}_j^*(s_0)$. Now, we need to show that the verifier $\mathsf{V}_{\mathsf{Batch}}$ rejects the transcript. For that, we consider two cases.

Case 1. $G_0(s_0) \neq G_0^*(s_0)$. In this case, note that the transcript $(G_0, \rho_1, G_0, \ldots, G_r)$ is suitable with respect to the sets Bad as in Definition 13 and Lucky as in Definition 12 and satisfies the conditions to apply the *inconsistent is rejected* of the non-batched FRI. Therefore, we can apply Lemma 10 to finish the proof of this case.

Case 2. $G_0(s_0) = G_0^*(s_0)$. In this case, assume towards contradiction that the verifier $\mathsf{V}_{\mathsf{Batch}}$ accepts the transcript. Then, we have

$$\sum_{j=1}^{B} \xi^{j-1} \mathbf{G}_j(s_0) = G_0(s_0) = G_0^*(s_0) = \sum_{j=1}^{B} \xi^{j-1} \mathbf{G}_j^*(s_0),$$

where the first equality follows from the definition of the verifier, the second equality is because we are in Case 2, and the third equality is because T is not in $\mathsf{BadBatch}$. Note that this means that $(\mathbf{G}, \xi) \in \mathsf{LuckyBatchColl}$, a contradiction to the assumption that T is suitable. □

D Script for Parameter Computation

Here, we give Python scripts used to compute tables and graphs in Section 5. We have used the scripts in [HASW23] as a starting point.

Listing 1: Python script to compute the parameters for different codes. A discussion is given in Section 5.

```
from dataclasses import dataclass

import math

# Statistical Security Parameter for Soundness
SECPAR_SOUND = 40

@dataclass
class Code:
    size_msg_symbol: int  # size of one symbol in the message
    size_code_symbol: int # size of one symbol in the code
    msg_len: int          # number of symbols in the message
    codeword_len: int     # number of symbols in the codeword
    reception: int        # number of symbols needed to reconstruct (worst case)
    samples: int          # number of random samples to reconstruct with high probability

    def interleave(self, ell):
        return Code(
            size_msg_symbol = self.size_msg_symbol * ell,
            size_code_symbol = self.size_code_symbol * ell,
            msg_len = self.msg_len,
            codeword_len = self.codeword_len,
            reception = self.reception,
            samples = self.samples
        )

    def tensor(self, col):
        assert self.size_msg_symbol == col.size_msg_symbol
        assert self.size_code_symbol == col.size_code_symbol
        assert self.size_msg_symbol == self.size_code_symbol

        row_dist = self.codeword_len - self.reception + 1
        col_dist = col.codeword_len - col.reception + 1
        codeword_len = self.codeword_len * col.codeword_len

        '''
        Example:

        D D | o o
        D D | o o
        ----+----
        o o | o o
        o o | o o

        Where D is the data.
        The reception is 8, since 7 is not enough to reconstruct:

        o o | o x
        o o | o x
        ----+----
        o o | o x
        x x | x x

        Given the symbols marked with x, I cannot reconstruct the data.
        '''
        reception = codeword_len - row_dist * col_dist + 1
        '''
        To determine the number of samples, we have multiple options.
        we can use the minimum of all resulting number of samples

        Option 1: use reception and generalized coupon collector
        As reception is a "worst case bound", this may not be tight

        Option 2: use a more direct analysis.
```

```
        not being able to reconstruct
        -> there is a row we can not reconstruct
        -> union bound over all rows
        -> for fixed row, assume we can not reconstruct
        -> there is a set of t_r - 1 positions (t_r = reception in rows)
        such that all queries in that row are in that set
        -> we union bounding over all of these sets
        -> for each fixed set, the probability that
        all queries in that row are in that set is
        (1-((n_r - t_r + 1)/(n_r*n_c)))^{number of samples}
        so the total probability of not being able to reconstruct is at most
        n_c * (n_r choose t_r - 1) * (1-((n_r - t_r + 1)/(n_r*n_c)))^{number of samples}
        and (n_r choose t_r - 1) <= (n_r * e / (t_r - 1))^(t_r - 1)

        Option 3: same as Option 2 but reversed roles

        Asymptotic example: Tensor C: F^k -> F^{2k} with itself
        Option 1   -> Omega(k^2 + sec_par) samples
        Option 2/3 -> Omega(k^2 + sec_par * k) samples

        Concretely, Option 2/3 will be tighter, especially for large k
        '''
        samples_via_reception = samples_from_reception(SECPAR_SOUND, reception, codeword_len)

        loge = math.log2(math.e)
        lognc = math.log2(col.codeword_len)
        lognr = math.log2(self.codeword_len)
        logbinomr = (self.reception - 1) * (lognr + loge - math.log2(self.reception - 1))
        loginnerr = math.log2(1.0 - (self.codeword_len - self.reception + 1)/codeword_len)
        logbinomc = (col.reception - 1) * (lognc + loge - math.log2(col.reception - 1))
        loginnerc = math.log2(1.0 - (col.codeword_len - col.reception + 1)/codeword_len)

        samples_direct_via_rows = int(math.ceil(-(lognc + logbinomr + SECPAR_SOUND)/loginnerr))
        samples_direct_via_cols = int(math.ceil(-(lognr + logbinomc + SECPAR_SOUND)/loginnerc))

        samples_direct = min(samples_direct_via_rows, samples_direct_via_cols)
        samples = min(samples_direct, samples_via_reception)

        return Code(
            size_msg_symbol = self.size_msg_symbol,
            msg_len = self.msg_len * col.msg_len,
            size_code_symbol = self.size_code_symbol,
            codeword_len = codeword_len,
            reception = reception,
            samples = samples
        )

    def __eq__(self, other):
        return (
            self.size_msg_symbol == other.size_msg_symbol
            and self.size_code_symbol == other.size_code_symbol
            and self.msg_len == other.msg_len
            and self.codeword_len == other.codeword_len
            and self.reception == other.reception
        )

    def is_identity(self):
        return (
            self.size_msg_symbol == self.size_code_symbol
            and self.msg_len == self.codeword_len
        )


def samples_from_reception(sec_par, reception, codeword_len):
    '''
    Compute the number of samples needed to reconstruct
    data with probability at least 1-2^{-sec_par} based on
    the reception efficiency and a generalized coupon collector.
    Note: this may not be the tightest for all schemes (e.g. Tensor)
    '''
    # special case: if only one symbol is needed, we are done
    if reception == 1:
        return 1

    # special case: if all symbols are needed: just regular coupon collector
    if reception == codeword_len:
        n = codeword_len
        s = math.ceil((n / math.log(math.e, 2)) * (math.log(n, 2) + sec_par))
        return int(s)

    # generalized coupon collector
    delta = reception - 1
    c = delta/codeword_len
    s = math.ceil(-sec_par / math.log2(c) + (1.0-math.log(math.e,c))*delta)
    return int(s)

# Identity code
def makeTrivialCode(chunksize, k):
    return Code(
        size_msg_symbol = chunksize,
        msg_len = k,
        size_code_symbol = chunksize,
        codeword_len = k,
        reception = k,
        samples = samples_from_reception(SECPAR_SOUND, k, k)
    )

# Reed-Solomon Code
# Polynomial of degree k-1 over field with field element length fsize
# Evaluated at n points
def makeRSCode(fsize, k, n):
    assert k <= n
    assert 2**fsize >= n, 'no such reed-solomon code :('
    return Code(
        size_msg_symbol = fsize,
        msg_len = k,
        size_code_symbol = fsize,
        codeword_len = n,
        reception = k,
        samples = samples_from_reception(SECPAR_SOUND, k, n)
    )
```

```
# tests
assert makeRSCode(5, 2, 4).tensor(makeRSCode(5, 2, 4)).reception == 8
assert makeRSCode(5, 2, 4).reception == 2
```

Listing 2: Python script to compute the parameters for different data availability sampling schemes. A discussion is given in Section 5.

```
#!/usr/bin/env python

import math

# Some constants.
# Sizes of group elements, field elements, and hashes in bits
BLS_FE_SIZE = 48.0 * 8.0
BLS_GE_SIZE = 48.0 * 8.0

# Let's say we use the SECP256_k1 curve
PEDERSEN_FE_SIZE = 32.0 * 8.0
PEDERSEN_GE_SIZE = 33.0 * 8.0

# Let's say we use SHA256
HASH_SIZE = 256


from dataclasses import dataclass

from codes import *

@dataclass
class Scheme:
    code: Code            # code that is used
    com_size: int         # size of commitment in bits
    opening_overhead: int # overhead of opening a symbol in the encoding

    def samples(self):
        '''
        i.e. the number of random samples needed to collect
        enough symbols except with small probability
        '''
        return self.code.samples

    def total_comm(self):
        '''
        Compute the total communication in bits.
        '''
        return self.comm_per_query() * self.samples()

    def comm_per_query(self):
        '''
        Compute the communication per query in bits.
        '''
        return math.log2(self.code.codeword_len) + self.opening_overhead + self.code.size_code_symbol

    def encoding_size(self):
        '''
        Compute the size of the encoding in bits.
        '''
        return self.code.codeword_len * (self.opening_overhead + self.code.size_code_symbol)

    def reception(self):
        '''
        Compute the reception of the code.
        '''
        return self.code.reception

    def encoding_length(self):
        '''
        Compute the length of the encoding.
        '''
        return self.code.codeword_len

# Naive scheme
# Put all the data in one symbol, and let the commitment be a hash
def makeNaiveScheme(datasize):
    return Scheme(
        code = Code(
            size_msg_symbol = datasize,
            msg_len = 1,
            size_code_symbol = datasize,
            codeword_len = 1,
            reception = 1,
            samples = 1
        ),
        com_size = HASH_SIZE,
        opening_overhead = 0
    )

# Merkle scheme
# Take a merkle tree and the identity code
def makeMerkleScheme(datasize, chunksize=1024):
    k = math.ceil(datasize / chunksize)
    return Scheme(
        code = makeTrivialCode(chunksize, k),
        com_size = HASH_SIZE,
        opening_overhead = math.ceil(math.log(k, 2))*HASH_SIZE
    )


# KZG Commitment, interpreted as an erasure code commitment for the RS code
# The RS Code is set to have parameters k,n with n = invrate * k
def makeKZGScheme(datasize, invrate=4):
    k = math.ceil(datasize / BLS_FE_SIZE)
    return Scheme(
        code = makeRSCode(
            BLS_FE_SIZE,
            k,
```

```
            k * invrate
        ),
        com_size = BLS_GE_SIZE,
        opening_overhead = BLS_GE_SIZE,
    )


# Tensor Code Commitment, where each dimension is expanded with inverse rate invrate.
# That is, data is a k x k matrix, and the codeword is a n x n matrix, with n = invrate * k
# Both column and row code are RS codes.
def makeTensorScheme(datasize, invrate=2):
    m = math.ceil(datasize / BLS_FE_SIZE)
    k = math.ceil(math.sqrt(m))
    n = invrate * k

    rs = makeRSCode(BLS_FE_SIZE, k, n)

    return Scheme(
        code = rs.tensor(rs),
        com_size = BLS_GE_SIZE * k,
        opening_overhead = BLS_GE_SIZE,
    )


# Hash-Based Code Commitment, over field with elements of size fsize,
# parallel repetition parameters P and L. Data is treated as a k x k matrix,
# and codewords are k x n matrices, where n = k*invrate.
def makeHashBasedScheme(datasize, fsize=32, P=8, L=64, invrate=4):
    m = math.ceil(datasize / fsize)
    k = math.ceil(math.sqrt(m))
    n = invrate * k
    rs = makeRSCode(fsize, k, n)

    return Scheme(
        code = rs.interleave(k),
        com_size = n * HASH_SIZE + P * n * fsize + L * k * fsize,
        opening_overhead = 0,
    )

# Homomorphic Hash-Based Code Commitment
# instantiated with Pedersen Hash
# parallel repetition parameters P and L. Data is treated as a k x k matrix,
# and codewords are k x n matrices, where n = k*invrate.
def makeHomHashBasedScheme(datasize, P=2, L=2, invrate=4):
    m = math.ceil(datasize / PEDERSEN_FE_SIZE)
    k = math.ceil(math.sqrt(m))
    n = invrate * k
    rs = makeRSCode(PEDERSEN_FE_SIZE, k, n)

    return Scheme(
        code = rs.interleave(k),
        com_size = n * PEDERSEN_GE_SIZE + P * n * PEDERSEN_FE_SIZE + L * k * PEDERSEN_FE_SIZE,
        opening_overhead = 0,
    )
```

Listing 3: Python script to compute the parameters for our FRI-based data availability sampling scheme FRIDA. A discussion is given in Section 5.

```
#!/usr/bin/env python
import math
from schemes import *

GRINDING = 20
RO_QUERIES = 60
STATISTICAL_SECURITY = 40
FRI_SOUNDNESS = STATISTICAL_SECURITY + RO_QUERIES - GRINDING

# assume a Merkle tree that represents numleafs tuples of elements
# where each element has size fsize and one tuple contains tuplesize
# many elements. Each leaf of the tree contains one such tuple.
# size of one opening (i.e., Merkle path + element)
# of a Merkle tree that represents n elements
# and each element has size fsize.
def sizeMerkleOpening(numleafs, tuplesize, fsize):
    tupleItself =  tuplesize * fsize
    sibling = tuplesize * fsize
    treedepth = math.ceil(math.log2(numleafs))
    copath = (treedepth - 1) * HASH_SIZE
    return tupleItself + sibling + copath


# size of the information needed to open one position
# in the FRI base layer, including all Merkle paths
# domainsize is the number of elements in the base layer
# fsize is the field size, i.e., size of one element
def friAuthSize(domainsize, rate, fsize, batchsize, fanin, basedimension):
    size = 0
    # batching phase if FRI_BATCH_SIZE > 1:
    # we need to open one symbol ( = FRI_BATCH_SIZE field elements)
    # of the interleaved code, so one leaf of the Merkle tree
    # representing the batch
    if batchsize > 1:
        size += sizeMerkleOpening(domainsize, batchsize, fsize)

    # now assume that we have already opened the batching
    # i.e., it remains to open everything from oracle G_0
    # to oracle G_r. In every oracle, the FRI verifier
    # queries on a set of fanin positions.
    # We put this entire set into the same Merkle leaf.
    # We do not put the final oracle in a Merkle tree.
    # Instead, we put it in plain into the commitment.
    # This makes sense as we have to open it entirely.
    ncurr = domainsize
    while ncurr * rate > basedimension:
        numleafs = ncurr // fanin
        size += sizeMerkleOpening(numleafs, fanin, fsize)
        ncurr = numleafs
```

```
    return size


# k = number of field elements to represent the data --> number of rounds
def friNumRounds(mink, fanin, basedimension):
    # if we do no round, we can represent basedimension many elements
    # if we do one round, we can represent basedimension * fanin many elements
    # if we do r rounds, we can represent basedimension * (fanin**r) many elements
    dimension = basedimension
    rnd = 0
    while dimension < mink:
        dimension *= fanin
        rnd += 1
    return rnd



# rate, size of first evaluation domain LLL_0, field size
# --> number of repetitions of query phase
def friNumRepetitions(rate, domainsize, fsize, batchsize, fanin):
    # first make sure that the soundness error
    # induced by LuckySet (e.g., distortion) is small
    maxbf = max(fanin, batchsize)
    logeps1 = 1 + math.ceil(math.log2(domainsize * (maxbf - 1))) - fsize
    assert(logeps1 <= - FRI_SOUNDNESS)

    # now determine number of repetitions such that the
    # soundness error related to the query phase is small
    # recall: the soundness error for L repetitions is
    # (1-delta^*/F)^L, and we need to get it below 2^{-FRI_SOUNDNESS}
    deltastar = 0.5 * (1.0 - rate)
    base = 1.0 - deltastar
    logbase = math.log2(base)
    assert(logbase < 0)
    L = - FRI_SOUNDNESS / logbase
    return math.ceil(L)

def makeFRIScheme(datasize, invrate = 4, fsize = 128, verbose = False):
    # determine k. Should be "compatible" with the fan-in
    # we need k to be at least ceil(datasize / fsize)
    minfe = math.ceil(datasize / fsize)
    if verbose:
        print("Need at least dimension minfe = " + str(minfe) + " field elements to represent the data.")


    # call algorithm to find good batchsize, fanin, and base dimension
    (batchsize, fanin, basedimension) = friGoodParameters(minfe, fsize, invrate)

    mink = math.ceil(minfe / batchsize)
    if verbose:
        print("With batch size B = " + str(batchsize) + ", we need at least dimension mink = " + str(mink))
        print("Use fanin F = " + str(fanin) + " and base dimension = " + str(basedimension))
    # now determine the number of rounds to get at least
    # dimension mink in the base layer
    r = friNumRounds(mink, fanin, basedimension)
    if verbose:
        print("Need " + str(r) + " rounds.")
    # with that, we get the actual k and n
    k = basedimension * (fanin ** r)
    n = invrate * k
    rate =  1.0 / invrate
    if verbose:
        print("Need dimension k = " + str(k) + " and evaluation domain size n = " + str(n) + ".")
    # determine the number of repetitions we need
    # to get good soundness guarantees
    L = friNumRepetitions(rate, n, fsize, batchsize, fanin)
    if verbose:
        print("Need " + str(L) + " repetitions of the query phase.")

    # determine the size of one opening
    authsize = friAuthSize(n, rate, fsize, batchsize, fanin, basedimension)

    # now compile the scheme
    rs = makeRSCode(fsize, k, n)

    # we include all openings for the final layer in the commitment, and no Merkle root for it
    # if we do batching, we need one root more
    final = basedimension * fsize
    openings = L * authsize
    roots =  r * HASH_SIZE + (batchsize > 1) * HASH_SIZE

    return Scheme(
        com_size = roots + final + openings,
        code = rs.interleave(batchsize),
        opening_overhead = authsize-batchsize*fsize,
    )

#-------------------------------------------------------------------------#
#                           OPTIMIZATION SECTION                          #
#-------------------------------------------------------------------------#

# given the minimum number of field elements we need to represent (minfe),
# the field size (fsize), the inverse rate (invrate), the basedimension, and
# the fanin, this function computes a good batchsize. Good means that the
# batchsize minimizes (in a certain range) the size of a single opening
def friGoodBatchsize(minfe, fsize, invrate, basedimension, fanin):
    batchsizerange = range(1,257)
    batchsize = 1
    mink = math.ceil(minfe / batchsize)
    r = friNumRounds(mink, fanin, basedimension)
    minauthsize = friAuthSize(basedimension * (fanin**r) * invrate, 1.0 / invrate, fsize, batchsize, fanin, basedimension)
    for b in batchsizerange:
        mink = math.ceil(minfe / b)
        r = friNumRounds(mink, fanin, basedimension)
        currauthsize = friAuthSize(basedimension * (fanin**r) * invrate, 1.0 / invrate, fsize, b, fanin, basedimension)
        if currauthsize <= minauthsize:
            batchsize = b
            minauthsize = currauthsize
    return batchsize

# given the minimum number of field elements we need to represent (minfe),
```

```
# the field size (fsize) and the inverse rate (invrate), this function
# computes (batchsize, fanin, basedimension) for FRI that works reasonably
# well. This is for sure not always the optimal setting, especially if a
# specific metric should be optimized, e.g., communication per query
def friGoodParameters(minfe, fsize, invrate):

    # overall idea is to minimize the gap between the dimension on the largest layer
    # and the dimension we would actually need to represent minfe elements. That is,
    # we minimimize gap = basedimension * fanin^rounds - minfe / batchsize, ensuring
    # that gap >= 0. To do so, we try a few reasonable fanins and base dimensions
    faninrange = [4, 8, 16]
    basedimensionrange = [2, 4, 6, 8, 16, 32, 64, 128]

    # start minimazation loop. Iterate over all combinations (fanin, basedimension)
    optfanin = 0
    optbasedimension = 0
    optbatchsize = 0
    mingap = -1
    for fanin in faninrange:
        for basedimension in basedimensionrange:
            # if we want to compute the gap for the pair (fanin, basedimension),
            # we need to know a suitable batchsize first. To find it, we want
            # to minimize the size of an opening, i.e., minimize friAuthSize
            batchsize = friGoodBatchsize(minfe, fsize, invrate, basedimension, fanin)
            mink = math.ceil(minfe / batchsize)

            # determine the number of rounds that we need now
            r = friNumRounds(mink, fanin, basedimension)
            # compute gap for this fanin, basedimension, and batchsize
            gap = basedimension * (fanin**r) - mink
            # update if it is better
            if mingap == -1 or (gap >= 0 and gap <= mingap):
                mingap = gap
                optfanin = fanin
                optbasedimension = basedimension
                optbatchsize = batchsize

    return (optbatchsize, optfanin, optbasedimension)
```

Listing 4: Python script to compute the tables in Section 5.

```
#!/usr/bin/env python

import math
import sys
from tabulate import tabulate

from schemes import *
from fri import *

def makeRow(name,scheme,tex):
        comsize = '{:.2f}'.format(round(scheme.com_size/8000.0,2))
        encodingsize = '{:.2f}'.format(round(scheme.encoding_size() / 8000000.0,2))
        commpqsize = '{:.2f}'.format(round(scheme.comm_per_query() / 8000.0,2))
        reception = scheme.reception()
        encodinglength = scheme.encoding_length()
        samples = scheme.samples()
        commsize = '{:.2f}'.format(round(scheme.total_comm() / 8000000.0,2))
        if tex:
                row = ["\Inst"+name,comsize,encodingsize,commpqsize,commsize]
        else:
                row = [name,comsize,encodingsize,commpqsize,(reception,encodinglength),samples,commsize]
        return row

######################################################################

opts = [opt for opt in sys.argv[1:] if opt.startswith("-")]
args = [arg for arg in sys.argv[1:] if not arg.startswith("-")]

if len(args) == 0:
        print("Missing Argument: Datasize in Megabytes.")
        print("Hint: To print the table in LaTeX code, add the option -l.")
        sys.exit(-1)

datasize = int(args[0])*8000000


# Print to LaTeX
tex = "-l" in opts

if tex:
        table = [["Name","|com|","|Encoding|","Comm. p. Q.","Comm Total"]]
else:
        table = [["Name","|com| [KB]","|Encoding| [MB]","Comm. p. Q. [KB]","Reception","Samples","Comm Total [MB]"]]


scheme = makeNaiveScheme(datasize)
table.append(makeRow("Naive",scheme,tex))

scheme = makeMerkleScheme(datasize)
table.append(makeRow("Merkle",scheme,tex))

scheme = makeKZGScheme(datasize)
table.append(makeRow("RS",scheme,tex))

scheme = makeTensorScheme(datasize)
table.append(makeRow("Tensor",scheme,tex))

scheme = makeHashBasedScheme(datasize)
table.append(makeRow("Hash",scheme,tex))

scheme = makeHomHashBasedScheme(datasize)
table.append(makeRow("HomHash",scheme,tex))

scheme = makeFRIScheme(datasize)
table.append(makeRow("FRI",scheme,tex))
```

```
if tex:
        print(tabulate(table,headers='firstrow',tablefmt='latex_raw',disable_numparse=True))
else:
        print(tabulate(table,headers='firstrow',tablefmt='fancy_grid'))
```

Listing 5: Python script to compute the graphs in Section 5.

```
#!/usr/bin/env python

import math
import sys
import csv
import os

from schemes import *
from fri import *

DATASIZEUNIT = 8000*1000 # Megabytes
DATASIZERANGE = range(1,156,15)

def writeCSV(path,d):
        with open(path, mode="w") as outfile:
                writer = csv.writer(outfile, delimiter=',')
                for x in d:
                        writer.writerow([x,d[x]])


# Writes the graphs for a given scheme
# into a csv file
def writeScheme(name,makeScheme):

        commitment = {}
        commpq = {}
        commtotal = {}
        encoding = {}

        for s in DATASIZERANGE:
                datasize = s*DATASIZEUNIT
                scheme = makeScheme(datasize)
                commitment[s] = scheme.com_size / 8000000 # MB
                commpq[s] = scheme.comm_per_query() /8000 # KB
                commtotal[s] = scheme.total_comm() /8000000000 # GB
                encoding[s] = scheme.encoding_size() /8000000000 # GB

        if not os.path.exists("./csvdata/"):
                os.makedirs("./csvdata")

        writeCSV("./csvdata/"+name+"_com.csv",commitment)
        writeCSV("./csvdata/"+name+"_comm_pq.csv",commpq)
        writeCSV("./csvdata/"+name+"_comm_total.csv",commtotal)
        writeCSV("./csvdata/"+name+"_encoding.csv",encoding)


##########################################
writeScheme("rs",makeKZGScheme)
writeScheme("tensor",makeTensorScheme)
writeScheme("hash",makeHashBasedScheme)
writeScheme("homhash",makeHomHashBasedScheme)
writeScheme("fri",makeFRIScheme)
```