Advancing Scalability in Decentralized Storage: A Novel Approach to Proof-of-Replication via Polynomial Evaluation

Giuseppe Ateniese [1], Foteini Baldimtsi [1], Matteo Campanelli [2], Danilo Francati [3], and Ioanna Karantaidou [1]

[1] George Mason University
[2] Protocol Labs
[3] Aarhus University

October 11, 2023

Abstract

Proof-of-Replication (PoRep) plays a pivotal role in decentralized storage networks, serving as a mechanism to verify that provers consistently store retrievable copies of specific data. While PoRep's utility is unquestionable, its implementation in large-scale systems, such as Filecoin, has been hindered by scalability challenges. Most existing PoRep schemes, such as Fisch's (Eurocrypt 2019), face an escalating number of challenges and growing computational overhead as the number of stored files increases. This paper introduces a novel PoRep scheme distinctively tailored for expansive decentralized storage networks. At its core, our approach hinges on polynomial evaluation, diverging from the probabilistic checking prevalent in prior works. Remarkably, our design requires only a single challenge, irrespective of the number of files, ensuring both prover's and verifier's run-times remain manageable even as file counts soar. Our approach introduces a paradigm shift in PoRep designs, offering a blueprint for highly scalable and efficient decentralized storage solutions.

Keywords: proof of replication, proof of space, polynomial evaluation.

Contents

1 Introduction

In recent years, there has been a significant shift in the domain of consensus mechanisms. Traditional proof-of-work (PoW) systems [44, 33], though revolutionary in their own right, have been scrutinized for their significant energy consumption and potential centralization due to ASIC dominance. As a result, the cryptographic community has been driven to explore alternative methods that could provide similar security guarantees without the associated ecological or centralization concerns [30, 11, 30, 50, 6, 27, 25, 19].

Proof-of-Space (PoS) [30, 11] emerged as a compelling alternative to proof-of-work, with applications spanning from spam prevention and DDoS attack resistance, to serving as the backbone for novel Sybil-resistant blockchain consensus protocols. Notably, the allure of PoS stems from its eco-friendly nature and resistance to ASIC dominance. By leveraging available storage space instead of massive energy consumption, PoS positions itself as a more egalitarian and sustainable solution when compared to PoW.

Proof-of-Replication (PoRep) extends the concept of PoS by requiring the prover to store useful replicated data that can also be retrieved by the verifier. In a PoS protocol, the prover demonstrates to a verifier that it is dedicating a minimum amount of storage space, but the stored data can be arbitrary. In contrast, a PoRep scheme verifies that the prover is persistently storing *retrievable* copies of a *specific* data file or dataset. While a PoS only proves the size of storage used, a PoRep provides stronger guarantees that the prover is dedicating unique storage resources per replica of the data. PoRep has the useful side effect of providing decentralized and verifiable file storage, unlike PoS which wastes the dedicated space. PoRep can be conceptualized as a fusion of (1) PoS with application-specific beneficial data, and (2) the robust guarantees of Proof of Data Possession (PDP) [12] and Proof of Retrievability (PoR) [38], ensuring not only the existence but also the retrievability of stored data replicas.

This practicality and robustness of PoRep make it especially attractive for real-world applications. One notable implementation is Filecoin [1], a decentralized storage network built on top of the Interplanetary File System (IPFS). Filecoin not only capitalizes on the concept of PoRep but elevates it as a foundational pillar. In the Filecoin ecosystem, storage providers are incentivized through the native cryptocurrency, FIL, to reliably store users' files. To prove their reliability and earn these rewards, providers are *audited*: they must demonstrate through PoRep that they are consistently storing retrievable replicas of client data by answering a series of *challenges*. This ensures that data is not merely stored but is also readily available for retrieval, aligning with Filecoin's vision of offering a more efficient, decentralized, and resilient alternative to traditional data storage methods.

However, as the decentralized storage vision of Filecoin materializes, certain requirements emerge as paramount:

1. *Laconic challenges*: An efficient PoS and PoRep would necessitate compact challenges, ideally of constant size.

2. *Sublinear prover and verifier*: To ensure longevity and efficiency, the prover's query complexity on its local storage (or replica) should be minimized, preventing wear and tear from frequent audits [2]. Also, verification should be efficient to make these scheme applicable in large scale systems.

3. *Robust space gap*: To ensure maximal security and data fidelity, the gap between data deletion and its detection during audits must be minimized.

Fisch [31] introduces a novel construction of tight PoRep rooted in graph labeling, targeting an asymptotic proof size of $O(\log N/\eta)$, where η represents the space gap (i.e., the difference between the amount of space the prover claims to be using and the actual space they are using). Central to this, is the mechanism of *stacked Depth Robust Graphs (DRGs)*, wherein multiple fixed-degree DRGs are systematically layered. Such an arrangement is designed to ensure that a slight perturbation in one layer's data triggers a cascading recomputation in the preceding layers. The inherent interplay between the number of DRG layers and the degree of each graph directly impacts the efficacy of the construction. A potential imbalance in this delicate equation could lead to surging proof complexities, a considerable impediment, especially when accommodating arbitrary values of ϵ. In an alternative model, Fisch [31] proposes the *ZigZag Expander DRGs*. By amalgamating each DRG layer with a non-bipartite expander graph of constant degree and intertwining their dependencies in a "zig-zag" fashion, this design promises more streamlined data extraction. However, it calls for doubling the layer count to maintain analogous security guarantees, a compromise that might not be universally optimal.

In the context of systems like Filecoin, where vast numbers of files are stored, the methodology introduced by Fisch may encounter scalability challenges. Specifically, as the number of files escalates, the computational overhead inherent to the PoRep mechanism of [31] becomes increasingly pronounced, making it potentially expensive in terms of both challenge generation and runtime efficiency for provers and verifiers.

This poses the question:

> *How can we develop a PoRep scheme that retains the security and robustness of prior models, yet offers improved scalability and efficiency suitable for large-scale decentralized storage networks?*

Our primary objective is to design an optimized PoRep scheme specifically for large-scale decentralized storage networks, rectifying the shortcomings identified in existing models. Our goals encompass enhanced computational efficiency, improved challenge-response mechanisms, and a reduced space gap. Central to our approach is an auditing technique for proofs of space, anchored in polynomial evaluation. This guarantees that the prover's efficiency stays sublinear, being notably poly-logarithmic relative to the file size. To achieve this, we employ sophisticated polynomial evaluation methods together with polynomial pre-processing, a technique detailed in [40]. This pre-processing phase yields a data structure that ensures poly-logarithmic processing times for both provers and verifiers.

In Table 1, we showcase the performance of the state-of-the-art protocol [31] based on the criteria listed above.

For a single file, our scheme's performance might align with, or slightly lag behind, Fisch's design as presented in [31]. This is partly due to the overheads associated with our polynomial evaluations and pre-processing. However, our distinct advantage lies in our challenge mechanism: irrespective of the number of files, our polynomial-based encoding consistently demands only one challenge. As the file count grows, this feature becomes increasingly beneficial. In contrast, Fisch's design sees its number of challenges rise in direct proportion to the number of files. This inherent trait means that both the prover's and verifier's run-times in Fisch's approach grow quadratically as more files are added (see Table 1). It is important to note that our method, while benefiting from a data structure tailored for univariate polynomial evaluation, does introduce an "expansion factor" which increases the data footprint. Yet, this mild growth in data might be a reasonable trade-off for the significant computational gains it offers, particularly in extensive systems like Filecoin where processing speed often takes precedence over storage size. In addition, this factor is adjustable; it can be fine-tuned according to the prover's specific requirements, providing flexibility within the PoRep construction.

Table 1: This table contrasts our PoRep (Theorem 9 and Corollary 6) with [31] based on replication (i.e., memory guarantee). Here, $u \geq 1$ represents file count, $|m| = N = d \cdot z$ denotes each file's bit size, with d as block count and z as each block's bit size. We use $O_\lambda(f(\cdot))$ to symbolize $O(f(\cdot) \cdot \mathsf{poly}(\lambda))$. Columns 2-4 relate the challenge count to the malicious prover's required memory n (in bits) for verification. Column 5 depicts prover and verifier run-times relative to challenge count, while Column 6 presents the factor γ which modifies memory for the honest prover (i.e., memory used is $u \cdot N \cdot \gamma$). The 4th column's (η)-gap, defined as $(1-\eta)u \cdot N = n$, denotes the difference between file size $u \cdot N$ and adversary's memory bound; a smaller gap indicates a better memory guarantee. Rows one and two (labeled "generic statement") outline the construction's security. The table's final two rows, set by $|q| = \lambda$ and $z = \lambda^{1+\delta}$ (for any constant $\delta > 0$), contrast replication assurances for a specified malicious prover memory-bound (Column 3). Colored cells emphasize optimal parameters.

Scheme	**# Challenges**	**Adv.'s memory n in bits**	**(η)-gap where $\eta \in [0,1]$**	**Prover's and verifier's run-time**	**Honest prover's memory (γ)-expansion**
[31] (generic statement)	$1/\eta$ (of size $\log(d)$)	$u(N - \eta \cdot N)$	any $\eta \in [0,1]$ (η can depend on d and u)	$O_\lambda(u/\eta \cdot \log d)$	$O(1)$
Ours § 6.1 (generic statement)	1 (element from $\mathbb{Z}_q$)	$u \cdot N - d \cdot \lvert q \rvert - \lambda$	$(d \cdot \lvert q \rvert - \lambda) / (u \cdot N)$	$O_\lambda(u \cdot \mathsf{polylog}(d))$	$O_\lambda\left(\sqrt[c]{d}\right)$ (for arbitrary constant $c \geq 1$)
[31]	$O\left(u \cdot \lambda^{\delta_1}\right)$ (of size $\log(d)$)	$u \cdot N - (d+1)\lambda$	$O\left(\left(u \cdot \lambda^{\delta}\right)^{-1}\right)$	$O_\lambda(u^2 \cdot \log d)$	$O(1)$
Ours § 6.1	1 (of size λ)	$u \cdot N - (d+1)\lambda$	$O\left(\left(u \cdot \lambda^{\delta}\right)^{-1}\right)$	$O_\lambda(u \cdot \mathsf{polylog}(d))$	$O_\lambda\left(\sqrt[c]{d}\right)$ (for arbitrary constant $c \geq 1$)

1.1 Our Contributions

Our primary contribution lies in designing an optimized Proof-of-Replication (PoRep) scheme tailored for large-scale decentralized storage networks. By centering our methodology around polynomial evaluation, we ensure that as file sizes grow, our prover's efficiency remains consistent and manageable. Key benefits of our system include enhanced computational efficiency, a streamlined challenge-response mechanism, and a reduced space gap. Our methodology presents significant benefits—particularly as the number of files grows—improving on prior work like Fisch's [31], where challenges increase proportionally with file count. To realize this, we have introduced innovative solutions that tackle the limitations of current PoRep models:

- We introduce a novel auditing mechanism rooted in polynomial evaluation. This technique streamlines the proof verification process, especially beneficial for expansive datasets. Drawing inspiration from [14], our core method involves combining an identifier and a message to form a polynomial that appears random. This strategy allows our system to operate with just a single challenge, regardless of the number of files, providing a notable advantage in scalability.

- Building on Kedlaya and Umans [40], we have developed a method to achieve poly-logarithmic prover's running time for polynomial evaluation in decentralized storage networks. By using a RAM data structure, polynomial evaluations are expedited, reducing time complexities. This structure, however, enlarges memory by a multiplicative factor γ, which can be adjusted based on prover requirements. This balances efficient computation time with manageable memory overhead, making polynomial evaluations more efficient in our PoRep scheme.

- To achieve efficient PoRep verification, we leverage localized RAM computation, ensuring both the prover's and verifier's tasks remain poly-logarithmic in complexity. By using

Merkle trees on top of our data structure, a verifier can check the integrity of a prover's computation without having full access to the prover's data structure. While this approach emphasizes the auditing phase, it is crucial to ensure honest generation of the root digest, which can be reinforced using a SNARK proof during the encoding phase. Our resulting authenticated data structure for polynomial evaluation is of independent interest: it can be seen as a *succinct polynomial commitment where opening algorithm is sublinear in the degree of the polynomial*. This construction—which achieves this property trading additional storage—is to the best of our knowledge the first of its type. All the other constructions we are aware of require linear proving time (an incomplete list includes the works [39, 46, 41]).

1.2 Technical Overview

Given the intricate nature of our solution, we provide a high-level overview of our PoRep construction. It achieves laconic challenges, efficient proving and verification complexity, and a robust space gap (i.e., high memory guarantees) as the number of files u increases. For clarity, we focus on enforcing memory usage (the replication property), sidelining extractability. Note that extraction arises from polynomial interpolation; messages/files are encoded into polynomials, enabling extraction via interpolating multiple evaluations.

Initially, we detail the syntax and security guarantees of PoRep. Then, we outline our single-file approach (case $u = 1$). Towards the end, we delve into handling multiple files (case $u \geq 1$) and draw comparisons with Fisch's PoRep [31]. We assume all messages consist of d blocks of size $|p| = \log(p)$, where p is a prime in our construction; thus, $|m| = d \cdot |p|$.

Syntax and Security of PoRep Schemes. A PoRep scheme allows encoding highly *compressible* messages (e.g., a file) into *incompressible* strings that represent the messages. These schemes consist of five algorithms: Setup, Encode, Prove, Verify, and Decode. The setup algorithm generates three *public keys*: an encoding key ek, a proving key pk, and a verification key vk. Each key is used during a specific phase of the PoRep scheme. The Encode algorithm computes the incompressible encoding of a message m. Given the encoding key ek, a message m, and an identifier id for m, it outputs an encoding c and a digest h for later verification. After encoding (and the publication of the digest h), the auditing phase begins. This phase involves the execution of Prove (on the prover's side) and Verify (on the verifier's side). This phase primarily ensures the prover stores the encoding c. Specifically, given a random challenge chall, the prover runs $\mathsf{Prove}(\mathsf{ek}, \mathsf{chall}, \mathsf{c})$ to produce a proof π that verifies the storage of c. On the verifier's side, using the same challenge chall and proof π, the verifier executes $\mathsf{Verify}(\mathsf{vk}, \mathsf{h}, \mathsf{chall}, \pi)$ (where h is associated with the prover's encoding) to ensure the prover passed the auditing phase. Lastly, the Decode algorithm, when provided the encoding key ek, inverts an encoding c to retrieve the original message m.

Informally, a PoRep must ensure: (i) the prover utilizes significant memory and (ii) encoded messages are retrievable. These properties are termed *replication* and *extraction*.[1] For a single message (denoted by $u = 1$), PoRep's replication represents the minimum memory n a prover must use to pass verification. When $u > 1$, the replication concept remains, but a prover must produce u verification proofs, and *n can vary based on the number of messages $u > 1$*. Higher values of n indicate better replication. Additionally, *we aim to enforce a memory usage of size n that scales with the number of files u*. Such enforcement necessitates limiting the prover's runtime, a common trait in PoRep schemes [31]. Given the trapdoorless nature of our PoRep (no secret keys), certain constraints emerge. For instance, in a decentralized setting like

[1] The replication property corresponds to PoS (proof-of-space) in [31].

blockchains, a message m (chosen by the prover) can be highly compressible. Each block of $m = (\mathsf{F}(\mathsf{k}, 1), \mathsf{F}(\mathsf{k}, 2), \ldots)$ is generated by evaluating a PRF $\mathsf{F}(\mathsf{k}, \cdot)$, where $\mathsf{k} \in \{0,1\}^\lambda$ is a short key. To compute a proof π, the prover can regenerate blocks of m as needed. This approach uses minimal memory since the sizes of ek, pk, id, and k aren't related to the message's size. Thus, our PoRep employs a "slow" Encode algorithm. Its speed is adjustable using the time parameter t chosen during Setup, restricting the adversary to producing proofs more quickly than the execution time of Encode.

Conversely, PoRep's extraction property ensures all u messages are retrievable when a prover, holding the encodings, passes the verification phase for any number of files u.

Enforcing Space through Polynomial Evaluation. Our starting point is the work of Ateniese et al. [14], which leverages the evaluation of a random polynomial to build verifiable capacity-bound functions, a specific type of space-based primitive. Let $\mathbb{Z}_p$ be a field of order p from which the coefficients of the polynomial are sampled, and let $\mathbb{Z}_q \subseteq \mathbb{Z}_p$ be a field of order q (a subset of $\mathbb{Z}_p$, i.e., $q < p$) from which evaluation points are sampled. At a high level, [14] examines a setting where a (possibly malicious) evaluator receives a randomly sampled polynomial $f(X) \in \mathbb{Z}_p[X]$ of degree $d-1$, preprocesses $f(X)$ to compute a memory α smaller than $|f(X)|$ (i.e., by compressing $f(X)$ or pre-computing and storing some evaluations of $f(X)$ on some adversarially chosen points $(x_1, x_2, \ldots)$), and then attempts to compute $y = f(x)$ on a randomly chosen point $x \in \mathbb{Z}_q$ using only α (and not $f(X)$). The work in [14] formally shows that the evaluator's memory α cannot be smaller than $|\alpha| \approx d \cdot |p| - d \cdot |q|$ where $|p|$ is the size of a coefficient and $|q|$ is the size of the challenge point.[2] When $q \ll p$ (e.g., $|q|$ is sublinear in $|p|$), we find that $|\alpha|$ is close to the size of $f(X)$, which is $|f(X)| = d \cdot |p|$. Hence, evaluating $f(X)$ requires memory close to $|f(X)|$.

This result forms the core idea of our PoReps. The encoding c of a message $m \in \{0,1\}^{d \cdot |p|}$ (comprising d blocks each of size $|p|$) concerning an identifier id (i.e., the execution of $\mathsf{Encode}(\mathsf{ek}, m, \mathsf{id})$) involves combining id and m to derive a polynomial $f(X)$ that appears randomly sampled from $\mathbb{Z}_p[X]$. We achieve this by calculating $f(X) = r \oplus m$ (interpreting each block of $r \oplus m$ as a coefficient of $f(X)$) where $r = \mathsf{Eval}_{\mathsf{MHF}}(\mathsf{id})$ is the output of a memory-hard function (MHF) (denoted by $\mathsf{Eval}_{\mathsf{MHF}}$) that remains secure against input-dependent pre-processing.[3] This type of MHF abstracts functions that are "slow" to compute in the presence of an adversary that conserves storage by omitting some of the labels needed for output computation. An example of such functions are those based on either stacked DRG or ZigZag Expander DRG proposed by Fisch [31], which informally ensure that an adversary, omitting some labels associated with the last layer of the underlying DRG, will face a high sequential runtime to compute the correct output $r = \mathsf{Eval}_{\mathsf{MHF}}(\mathsf{id})$.[4]

Using this method, we determine that an evaluator utilizing memory of size at most $|\alpha| = n \approx \min\{n_{\mathsf{MHF}}, d \cdot |p| - d \cdot |q|\}$ (where $\approx d \cdot |p| - d \cdot |q|$ is the memory-bound provided by $f(X)$ and n_{MHF} is the one offered by the MHF) cannot compute $f(x)$ (on a randomly sampled x) in parallel time t_{MHF} where t_{MHF} is the time-bound offered by the MHF, dependent on n_{MHF}. This stems from the security guarantees of polynomial evaluation and MHF described earlier.[5] Setting $n_{\mathsf{MHF}} \lessapprox d \cdot |p| - d \cdot |q|$ (achievable by adjusting settings on the graph of [31] such as the

[2] To be precise, the memory size is $|\alpha| = d \cdot |p| - d \cdot |q| - \lambda$. We ignore the loss λ and we write $|\alpha| \approx d \cdot |p| - d \cdot |q|$ for clarity.

[3] Looking ahead, our construction will compute $f(X)$ as $f(X) = \mathsf{H}(r, \mathsf{id})$ where $r = \mathsf{Eval}_{\mathsf{MHF}}(\mathsf{id})$ and H is a hash function modeled as a random oracle (RO). In this section, we assume $f(X) = r \oplus m$ for clarity.

[4] Following our abstraction, $r = \mathsf{Eval}_{\mathsf{MHF}}(\mathsf{id})$ concatenates the random oracle labels associated with the last layer of the DRG as defined in [31].

[5] This requires selecting the minimum memory-bound $|\alpha| = n \approx \min\{n_{\mathsf{MHF}}, d \cdot |p| - d \cdot |q|\}$ provided by the two to ensure both hold simultaneously.

number of layers and nodes per layer of the underlying DRG), results in $|\alpha| = n \approx d \cdot |p| - d \cdot |q|$, close to $d \cdot |p| = |m|$ when $q \ll p$. Thus, requesting $y = f(x)$ will necessitate the evaluator to use memory $|\alpha| \approx |m|$ when restricted to a runtime shorter than t_{MHF}.

Though the above solution enforces significant space usage on the prover's side, which is essential for the PoReps' replication property, it presents the following challenges:

1. *How can the prover efficiently compute $y = f(x)$ to achieve sublinear prover's runtime?* Currently, evaluating the polynomial takes time linear in d (i.e., the number of coefficients).

2. *How can the verifier efficiently check that $y \overset{?}{=} f(x)$?* In other words, how can we make the above scheme verifiable in sublinear time?

We discuss solutions to these problems in the subsequent paragraphs.

Achieving poly-logarithmic prover's running time. As described in the previous paragraph, our approach asks a (possibly malicious) prover to compute $f(x)$ on a randomly sampled point in $\mathbb{Z}_q$ where $f(X) = m \oplus \mathsf{Eval}_{\mathsf{MHF}}(\mathsf{id})$. To decrease the prover's running time, we need to make the evaluation of a polynomial efficient. Several works [40, 43, 10, 17, 20, 53, 54, 37, 45, 35] have proposed different techniques to enable fast polynomial evaluation. This led to the work of Kedlaya and Umans [40] which proposed a RAM data structure D that allows computing $f(x)$ (for any $x \in \mathbb{Z}_p$) in time poly-logarithmic in the number of coefficients d of the polynomial $f(X) \in \mathbb{Z}_p[X]$ (recall that the number of coefficients d corresponds to the number of blocks of the encoded message). Formally, [40] shows the existence of an algorithm $\mathsf{GenData}$ that, on input $f(X) \in \mathbb{Z}_p$ and p, outputs a data structure D. Then, an evaluator can execute $\mathsf{Eval}(x, \mathsf{D})$ (i.e., Eval leverages the RAM access to D to read some blocks from D) to compute $y = f(x)$ in time $\mathsf{poly}(\log(d), |p|)$.

We note that, to achieve a $\mathsf{poly}(\log(d), |p|)$ evaluation time, the data structure D (output by $\mathsf{GenData}$) is larger than the size of $f(X)$, since $f(X)$ is pre-computed and manipulated. In particular, the D of [40] has a multiplicative overhead which we term (γ)-expansion, meaning the size of D is $|f(X)| \cdot \gamma$ (D is γ times larger than the size of the original polynomial). Specifically, the (γ)-expansion of [40] is $\gamma = \sqrt[c]{d} \cdot \log^{o(1)}(p)$ for any arbitrary constant $c > 1$. Thus, the running time of an honest prover can be made poly-logarithmic in d by increasing its memory by a multiplicative factor $\gamma = \sqrt[c]{d} \cdot \log^{o(1)}(p)$, which is sublinear in $|f(X)|$. On the bright side, the factor $\sqrt[c]{d}$ (of the expansion γ) can be made arbitrarily small by selecting a larger constant $c > 1$. This parameter can be chosen by the prover according to its needs: c can be dynamically increased or decreased as it is not fixed by the PoRep construction.

Poly-logarithmic verification through localized RAM computation. Until now, the technique we have described relies on requiring the prover (who stores m) to evaluate a polynomial $f(X)$, where $f(X) = m \oplus \mathsf{Eval}_{\mathsf{MHF}}(\mathsf{id})$ represents an encoding of m with respect to the identifier id, on a randomly chosen point $x \in \mathbb{Z}_q$. As noted at the beginning of this section, evaluating $f(X)$ suffices to verify that the prover is utilizing a memory α of size $|\alpha| \approx d \cdot |p| - d \cdot |q|$, which is approximately $d \cdot |p| = |m|$ when $|q| \ll |p|$. The remaining challenge is to enable a verifier to ascertain that $y \overset{?}{=} f(x)$.[6]

At first glance, one might assume that poly-logarithmic verification could be integrated using standard techniques for verifying computations, such as SNARKs. For instance, we could employ a SNARK (which permits efficient verification) to have the prover generate a proof

[6] If the verifier cannot validate $y \overset{?}{=} f(x)$, a prover could bypass the verification without using any memory by simply outputting a malicious $\tilde{y} \neq f(x)$.

π that demonstrates $y = f(x)$. Specifically, this could be achieved by validating that y was honestly computed using the data structure D (retained by the prover) outlined in the previous paragraph. However, this strategy is flawed because it would result in the prover's running time becoming linear in the size of the data structure D. Generating a proof with SNARKs demands time linear to the size of the witness (of the required relation), which corresponds to the data structure D. This would nullify the advantages gained by employing the efficient data structure.

Our solution to verification capitalizes on the observation that we target the same poly-logarithmic (in d) complexity for both the prover and the verifier. It might therefore be adequate for a verifier to replicate the prover's computation and verify that the derived result $f(x) = y'$ matches y, the value produced by the prover. A significant obstacle in implementing this strategy is that the verifier lacks access to D. To address this, we utilize Merkle trees (or any vector commitment) atop D. Specifically, the data structure $\mathsf{D} = (\mathsf{D}_1, \ldots, \mathsf{D}_\ell)$ is segmented into ℓ blocks, and h represents the root of the corresponding Merkle tree (with the tree's height being $\log(\ell)$). We recognize that, during the computation of y, the prover will access at most a poly-logarithmic number of blocks $\mathsf{D}' \subset \mathsf{D}$ from $\mathsf{D} = (\mathsf{D}_1, \ldots, \mathsf{D}_\ell)$ (recalling that D is a RAM data structure), making its running time poly-logarithmic in d. For verification, the prover merely needs to transmit y, the accessed blocks D', and the associated $|\mathsf{D}'|$ Merkle tree openings $(\pi'_1, \ldots, \pi'_{|\mathsf{D}'|})$. Thus, a verifier possessing the digest h can:

1. Using the openings $(\pi'_1, \ldots, \pi'_{|\mathsf{D}'|})$, verify that the received blocks D' align with the blocks of D.

2. Compute $y' = f(x)$ by running the evaluation algorithm Eval of the data structure solely with the received blocks D'.

3. Validate that $y' \stackrel{?}{=} y$, where y is the evaluation returned by the prover.

The ensuing verification process ensures the prover's running time remains poly-logarithmic in d while facilitating poly-logarithmic verification. We emphasize that h must be computed honestly to guarantee the soundness of the above verification procedure. Although this paper's contribution centers on the auditing phase, we implicitly presume that h is generated with integrity. Nonetheless, we stress that the integrity of h (furnished by the prover) can be assured by incorporating a SNARK proof during the encoding phase conducted by the prover. This doesn't impact the paper's conclusions since we don't set a specific limit on the running time of Encode.[7]

We term the act of conducting an accurate RAM computation using only the blocks involved in the computation (as performed by our verification algorithm) as "localized RAM computation". A thorough analysis is presented in Section 4, where we demonstrate that every RAM algorithm possesses its corresponding localized version, with the runtime being identical up to a logarithmic factor. While the foundational concept may appear straightforward, the formal proof is nuanced. For a comprehensive overview, we direct readers to Section 4.

Multiple files and comparison with Fisch's PoRep [31]. In the case of multiple messages or files (when $u > 1$), a prover must provide u verifying proofs to pass the auditing. As previously discussed, when a single message is stored, our PoRep ensures that a prover utilizes at least $n \approx d \cdot |p| - d \cdot |q|$ memory to pass the verification. Using a hybrid argument, we can assert that the memory requirement when $u \geq 1$ must be at least $n \approx u(d \cdot |p| - d \cdot |q|)$. A limitation of this memory-bound n is that the loss increases with the number of stored files. Specifically,

[7] Applying a SNARK at encoding time, replication can be ensured even under malicious executions of Encode. A variant of this method is employed in practice by Filecoin. See [1].

if a prover can save 1 GB with $u = 1$, then the same prover can save up to u GB when $u > 1$. This is naturally an undesirable outcome. A pertinent question arises: can we mitigate such a loss, potentially making it independent of the number of messages u?

We demonstrate that, in contrast to [31], this is achievable by expanding the analysis on the memory needed to evaluate a random polynomial (as outlined at the beginning of this section) to consider the case where $u > 1$ for polynomial evaluations. Our findings indicate that when evaluating u random polynomials $f_1(X), \ldots, f_u(X)$ at the *exact same* point $x \in \mathbb{Z}_q$ (i.e., the consistent PoRep's challenge), the memory required for the evaluation must be at least $|\alpha| = n \approx u \cdot d \cdot |p| - d \cdot |q|$, approximating $u \cdot |m|$ when $q \ll p$. Consequently, polynomial evaluations enable our PoRep scheme to *amortize* the memory loss when multiple polynomials are evaluated.

By integrating this insight with the method introduced at the start of this section, we derive a PoRep scheme that compels an evaluator (even if potentially malicious) operating in a time frame shorter than t_{MHF} (as defined by the underlying MHF) to allocate at least $n \approx u \cdot |m|$ memory to pass the auditing phase (i.e., computing u verifying proofs). The sole prerequisite is encoding the u messages $m_1, \ldots, m_u$ into u distinct random polynomials. Given that messages can be chosen with malice (e.g., they could all be identical), a unique identifier id_i for each m_i is mandatory. This is the sole alteration necessary to achieve the stated bound.

Regarding Fisch's PoRep construction [31], we note that, for $\eta \in [0, 1]$, the challenge count needed is $O(1/\eta)$ when the prover employs memory of size $|m| - \eta \cdot |m|$ (refer to Table 1). The memory-bound exhibits a loss that scales linearly with the message count u, meaning the challenge count must be $O(1/\eta)$ when $u \cdot |m| - u \cdot \eta \cdot |m|$. Hence, to ensure a loss independent of u, η must be inversely proportional to u, e.g., $\eta = \frac{1}{u} \cdot \eta'$ for some other $\eta' \in [0, 1]$. Nevertheless, this directly influences the challenge count of [31], causing it to rise linearly with the file count u, leading to a challenge count of $O(1/\eta) = O(u/\eta')$. This, in turn, affects the prover's operational time, making it quadratic in terms of message count, necessitating the prover to compute u proofs, each containing u openings.

Conversely, our PoRep demands a singular challenge (a point x), while achieving a memory-bound of $n \approx u \cdot |m|$, and a prover's operational time of $u \cdot \mathsf{poly}(\log(d), \log(p))$. For a juxtaposition between our approach and Fisch's PoRep security, we direct readers to Table 1.

2 Related Work

Proof-of-Space (PoS) protocols ensure provers allocate specific memory amounts, emphasizing efficient verification and communication. Some approaches, like the pebbling-based ones, delve into directed acyclic graphs to amplify space guarantees [30, 11, 4, 50, 6, 18]. Proof-of-Replication (PoRep) confirms storage providers are genuinely replicating data, thwarting them from storing unrelated content [48, 32, 26, 8, 24, 42, 47, 36, 23, 31]. Beyond PoS and PoRep, the cryptographic storage protocol landscape is varied and diverse. This section delves into other notable cryptographic storage mechanisms, highlighting their relevance and contributions to the field.[8]

Proof of Data Possession (PDP): PDP schemes are a cornerstone in cryptographic storage, enabling a storage provider to convince clients that their outsourced data remains intact and available. While they achieve the space-hardness goal of PoS when large, incompressible data is in play, they often entail significant communication costs, especially during the initial data transfer [12, 15, 52, 34, 9].

Proof of Retrievability (PoR): PoR is another fundamental protocol that permits clients to

[8] For a comprehensive overview, see https://proofofspace.org.

ensure the integrity and retrievability of their stored files on a server. It's equipped with an extractor that facilitates the reconstruction of the client's file from the provided proofs [38, 51, 28, 21].

Memory-Hard Functions (MHF): These are functions designed to require considerable memory/space for computation, primarily aimed at constructing ASIC-resistant proofs-of-work. While they demand continuous CPU utilization, their distinction from PoS is their ability to keep provers offline while still utilizing space-time [5, 7, 16, 3].

Proof of Secure Erasure (PoSE): This protocol ensures a prover's ability to confirm the erasure of specific memory portions. When integrated with PoS, it offers a holistic solution to secure storage and subsequent data erasure [4].

Proof of Transient Space (PoTS) and Proof of Persistent Space (PoPS): These protocols emphasize the temporal aspect of storage, with PoPS focusing on ensuring provers allocate space over time, verified through periodic audits. When integrated with PDP or PoR, they underline the prover's commitment to storage over extended periods [13, 49, 22].

3 Preliminaries

3.1 Notation

Bold capital letters (such as $\mathbf{X}$) are used to denote random variables, small letters (such as x) to denote concrete values, calligraphic letters (such as $\mathcal{X}$) to denote sets, serif letters (such as A) to denote algorithms. For a string $x \in \{0,1\}^*$, we let $|x|$ be its length; if $\mathcal{X}$ is a set, $|\mathcal{X}|$ represents the cardinality of $\mathcal{X}$. When x is chosen uniformly from a set $\mathcal{X}$, we write $x \leftarrow\!\!\$\, \mathcal{X}$. We use $\mathbf{U}_n$ to denote the uniform distribution over $\{0,1\}^n$. For an arbitrary distribution $\mathbf{X}$ (e.g., non-uniform) over a set $\mathcal{X}$, we write $x \leftarrow\!\!\$\, \mathbf{X}$ the act of sampling x from $\mathcal{X}$ according to the distribution $\mathbf{X}$. If A is a deterministic algorithm, we write $y = \mathsf{A}(x)$ to denote a run of A on input x and output y; if A is randomized, we write $y \leftarrow\!\!\$\, \mathsf{A}(x)$ (or $y = \mathsf{A}(x; r)$) to denote a run of A on input x and (uniform) randomness r, and output y. An algorithm A is *probabilistic polynomial-time* (PPT) if A is randomized and for any input $x, r \in \{0,1\}^*$ the computation of $\mathsf{A}(x; r)$ terminates in a polynomial number of steps (in the input size).

3.2 Memory-Hard Function with Input-dependent Pre-processing

A memory-hard function (MHF) with input space $\mathcal{X}$ and output space $\mathcal{Y}$, consists of the following polynomial-time algorithms:

$\mathsf{Setup}(1^\lambda, 1^t)$**:** On input the security parameter 1^λ and the time parameter 1^t, the randomized setup algorithm outputs the public parameter pp.

$\mathsf{Eval}(\mathsf{pp}, x)$**:** On input the public parameters pp and an input $x \in \mathcal{X}$, the deterministic evaluation algorithm outputs $y \in \mathcal{Y}$.

We are interested in secure MHF even in the presence of input-dependent pre-processing. Informally, such a flavor of MHF guarantees that it is infeasible to compute $y = \mathsf{Eval}(\mathsf{pp}, x)$ (for a random input $x \leftarrow\!\!\$\, \mathcal{X}$) in parallel time complexity σ with $O(\mathsf{poly}(t))$ processors (i.e., the computation of $y = \mathsf{Eval}(\mathsf{pp}, x)$ is non-parallelizable). This must hold even if the adversary pre-processes the MHF by producing a string α (of bounded size) conditioned to the challenged input x.

Definition 1 (Input-dependent pre-processing security of MHF)**.** *Let $\sigma(\lambda, t, n) = \sigma$ be a polynomial function that depends on the security parameter λ, time parameter t, and the memory* *bound n. A MHF scheme $\Pi = (\mathsf{Setup}, \mathsf{Eval})$ with message space $\mathcal{X}$ and output space $\mathcal{Y}$ is (ϵ, σ, n)-secure if for every valid PPT adversary $\mathsf{A} = (\mathsf{A}_1, \mathsf{A}_2)$, then*

$$\mathbb{P}\Big[\mathsf{Eval}(\mathsf{pp}, x) = \mathsf{A}_2(1^\lambda, 1^t, \mathsf{pp}, x, \alpha) \;\wedge\; |\alpha| \leq n\Big] \leq \epsilon$$

where $\mathsf{pp} \leftarrow\!\!\$\; \mathsf{Setup}(1^\lambda, 1^t)$, $x \leftarrow\!\!\$\; \mathcal{X}$, and $\alpha \leftarrow\!\!\$\; \mathsf{A}_1(1^\lambda, 1^t, \mathsf{pp}, x)$. An adversary $\mathsf{A} = (\mathsf{A}_1, \mathsf{A}_2)$ is called valid if A_2 runs in parallel time σ with $\mathsf{poly}(t)$ processors.

Remark 1 (On the output length of MHF with input-dependent pre-processing)**.** *In Definition 1, the auxiliary information α of size at most n (e.g., the memory state) can depend on the challenge input $x \leftarrow\!\!\$\; \mathcal{X}$. This implies that the output $y = \mathsf{Eval}(\mathsf{pp}, x)$ must be incompressible, i.e., it is infeasible to compress y (in polynomial-time) into a string of size at most n. Otherwise, A_1 can simply compute $y = \mathsf{Eval}(\mathsf{pp}, x)$ and output α which is the compression of y. In such a way, the second adversary A_2 may simply decompress α to recompute y, possibly bypassing the non-parallelizable computation enforced by the MHF scheme. This is analogous to the DRG constructions of [31] in which the output is set to the labels of the last layers of the DRG, and an adversary can delete a fraction of those labels.*

We will use the following corollary and instantiate Definition 1 from [31].

Corollary 1. *Let $\sigma(\lambda, t, n) = \sigma$ be a polynomial function in the security parameter λ, the time parameter, and the memory bound n (as defined in Definition 1). For every $\lambda \in \mathbb{N}$, for every $n \in \mathsf{poly}(\lambda)$, there exists a $(\mathsf{negl}(\lambda), \sigma, n)$-secure MHF with input space $\{0,1\}^\lambda$ in the parallel random oracle model (parallel ROM).*[9]

The work in [31] presents a construction of a *stacked depth-robust graph (DRG)*. The last layer in the "stack" contains the sinks of the graph. All nodes in the graph are labeled. Each of these labels is computed as a random oracle applied to the labels of the node's parents. The results in [31] can then be interpreted as: any adversary storing less than 80% of the sinks will need to perform a $\Omega(n)$ sequential computation where n is the number of blocks.

This allows us to instantiate Corollary 1 as follows. We define the evaluation of our MHF as the computation of the labels of the sinks in the graph. The labels of the source nodes are defined in terms of x, the input to the evaluation function. The setup of our MHF corresponds to the topology of the graph.

3.3 Vector Commitments and Merkle Trees

A vector commitment (VC) scheme with message space $\mathcal{M}$ is composed of the following polynomial-time algorithms:

$\mathsf{Setup}(1^\lambda)$**:** On input the security parameter 1^λ, the randomized setup algorithm outputs the public parameters pp.

$\mathsf{Commit}(\mathsf{pp}, (m_1, \ldots, m_\ell))$**:** On input the public parameters pp and a sequence of ℓ messages $(m_1, \ldots, m_\ell) \in \mathcal{M}^\ell$, the deterministic commit algorithm outputs a commitment c and an auxiliary information aux.

$\mathsf{Open}(\mathsf{pp}, m, i, \mathsf{aux})$**:** On input the public parameters pp, a message $m \in \mathcal{M}$, an index $i \in [\ell]$, and an auxiliary information aux, the deterministic open algorithm outputs a proof π.

[9] The parallel time complexity of A_2 corresponds to the number of parallel random oracle (parallel RO) queries submitted by A_2.

$\mathsf{Verify}(\mathsf{pp}, \mathsf{c}, m, i, \pi)$**:** On input the public parameters pp, a message $m \in \mathcal{M}$, and index $i \in [\ell]$, and a proof π, the deterministic verification algorithm outputs a decision bit $b \in \{0, 1\}$.

A VC scheme must satisfy the standard definitions of (perfect) correctness and position binding. Moreover, we focus on VC schemes where the running times of both Open and Verify are polylogarithmic in ℓ in the RAM model of computation.

Definition 2 (Efficiency of VC)**.** *A VC scheme* $\Pi = (\mathsf{Setup}, \mathsf{Commit}, \mathsf{Open}, \mathsf{Verify})$ *with message space* $\mathcal{M}$ *is efficient if both the open algorithm* Open *and the verification algorithm* Verify *have (worst-case) running time* $\mathsf{poly}(\lambda, \log(\ell))$ *where* ℓ *is the vector length parameter given as input to* Setup*. The running times of both* Open *and* Verify *are measured in the RAM model of computation.*

Definition 3 (Perfect correctness of VC)**.** *A VC scheme* $\Pi = (\mathsf{Setup}, \mathsf{Commit}, \mathsf{Open}, \mathsf{Verify})$ *with message space* $\mathcal{M}$ *is perfectly correct if* $\forall \lambda \in \mathbb{N}, \forall \ell \in \mathbb{N}, \forall (m_1, \dots, m_\ell) \in \mathcal{M}^\ell, \forall i \in [\ell]$*, the following probability holds:*

$$\mathbb{P}\left[\mathsf{Verify}(\mathsf{pp}, \mathsf{c}, m_i, i, \pi) = 1 : \begin{array}{c} \mathsf{pp} \leftarrow\!\!\$\, \mathsf{Setup}(1^\lambda), \\ (\mathsf{c}, \mathsf{aux}) = \mathsf{Commit}(\mathsf{pp}, (m_1, \dots, m_\ell)), \\ \pi = \mathsf{Open}(\mathsf{pp}, m_i, i, \mathsf{aux}) \end{array}\right] = 1$$

Definition 4 (Position binding of VC)**.** *A VC scheme* $\Pi = (\mathsf{Setup}, \mathsf{Commit}, \mathsf{Open}, \mathsf{Verify})$ *with message space* $\mathcal{M}$ *satisfies* (ϵ)*-position binding if for every PPT adversary* A*, we have:*

$$\mathbb{P}\left[\begin{array}{c} \mathsf{Verify}(\mathsf{pp}, \mathsf{c}, m, i, \pi) = 1\ \wedge \\ \mathsf{Verify}(\mathsf{pp}, \mathsf{c}, m', i, \pi') = 1\ \wedge \\ m \neq m' \end{array} : \begin{array}{c} \mathsf{pp} \leftarrow\!\!\$\, \mathsf{Setup}(1^\lambda), \\ (\mathsf{c}, m, m', i, \pi, \pi') \leftarrow\!\!\$\, \mathsf{A}(1^\lambda, \mathsf{pp}) \end{array}\right] \leq \epsilon$$

Merkles tree (implemented using collision-resistance hash functions) are efficient VCs.[10] Below, we report the formal corollary.

Corollary 2. *Assuming a collision-resistant hash function, there exists a* $(\mathsf{negl}(\lambda))$*-position binding and efficient VC scheme with message space* $\mathcal{M} = \{0, 1\}^z$ *(for any* $z \geq 1$*) and efficiency as defined in Definition 2. Moreover, we have that* $|\mathsf{c}| = \lambda$ *(i.e., commitments are succinct) and* $|\mathsf{aux}| = \ell \cdot z$*.*

3.4 Efficient Data Structure for Univariate Polynomial Evaluation

Next, we introduce the notion of data structures (DS) for univariate polynomial evaluation. At a high level, D is a DS for univariate polynomial evaluation if there exists a RAM algorithm that, given a point x, reads some blocks from D (using its RAM access to D) to compute $f(x)$ where $f(X) = \sum_{i=0}^{d} a_i \cdot X^i \in \mathbb{Z}_p[X]$ is the univariate polynomial taken into account.

More formally, let p be a prime and $\mathbb{Z}_p$ be a field. A (possibly efficient) data structure (DS) for evaluation of univariate polynomials is composed of the following polynomial-time algorithms:

$\mathsf{GenData}(f, p)$**:** On input a univariate polynomial $f(X) = \sum_{i=0}^{d} a_i \cdot X^i \in \mathbb{Z}_p[X]$ (of degree $d \in \mathbb{N}$) and a prime $p \in \mathbb{N}$, the deterministic data structure generation algorithm outputs a data structure D.

[10] In Merkle trees, the auxiliary information aux (output by Commit) corresponds to the intermediate hashes of the tree. Hence, computing the opening π (for some $m_i \in \mathcal{M}$ at position i) requires time $\lambda \cdot \log(\ell) + \log(|\mathcal{M}|)$ in the RAM model of computation since the evaluator needs to read $\log(\ell)$ intermediate hashes (from aux) each of size λ and a leaf (i.e., the sibling message of m_i) of size $\log(|\mathcal{M}|)$.

$\mathsf{Eval}(x, \mathsf{D})$**:** On input a point $x \in \mathbb{Z}_p$ and the data structure D, the deterministic evaluation algorithm outputs $y \in \mathbb{Z}_p$.

Intuitively, correctness says that, for every prime $p \in \mathbb{Z}_p$, for every $f(X) \in \mathbb{Z}_p[X]$, for every $x \in \mathbb{Z}_p$, the evaluation algorithm $\mathsf{Eval}(x, \mathsf{D})$ correctly computes $f(x)$ when $\mathsf{D} = \mathsf{GenData}(f, p)$.

Definition 5 (Perfect correctness of DS)**.** *A DS $\Pi = (\mathsf{GenData}, \mathsf{Eval})$ for evaluation of univariate polynomials is perfectly correct if $\forall$ prime $p \in \mathbb{N}$, $\forall f(X) \in \mathbb{Z}_p[X], \forall x \in \mathbb{Z}_p$, the following probability hold:*

$$\mathbb{P}[f(x) = \mathsf{Eval}(x, \mathsf{GenData}(f, p))] = 1.$$

Naturally, a trivial DS D corresponds to the coefficients of $f(X)$: evaluating $f(x)$ requires reading the d coefficients $a_i \in \mathbb{Z}_p$ from D. In this paper, we are interested in DS for univariate polynomial evaluation with non-trivial efficiency, i.e., the computation of $f(x)$ requires time $\mathsf{poly}(\log(d), \log(p))$ where p and d are the prime and the degree of the univariate polynomial $f(X) \in \mathbb{Z}_p[X]$ given in input to $\mathsf{GenData}$. We highlight that in order to obtain poly-logarithmic (or any sublinear) evaluation time, the data structure generation algorithm $\mathsf{GenData}$ may need to pre-process $f(X) \in \mathbb{Z}_p$. Hence, the data structure D (output by $\mathsf{GenData}$) may be bigger than $|f(X)| = (d+1) \cdot \log(p)$ where $\log(p)$ is the bit size of a coefficient $a_i \in \mathbb{Z}_p$. For this reason, the following definition of efficiency is parametrized by $\gamma = \frac{|\mathsf{D}|}{|f(X)|}$. Throughout the paper, we refer to γ as the expansion factor of DS. The formal definition follows.

Definition 6 (Efficiency with (γ)-expansion of DS)**.** *A DS $\Pi = (\mathsf{GenData}, \mathsf{Eval})$ for evaluation of univariate polynomials is efficient with (γ)-expansion if the following conditions hold:*

(γ)-expansion: *The size of D (output by $\mathsf{GenData}(f, p)$) is bounded by $|f(X)| \cdot \gamma = (d+1)\log(p) \cdot \gamma$ where d is the degree of $f(X) \in \mathbb{Z}_p[X]$ and $\log(p)$ is the size of a coefficient $a_i \in \mathbb{Z}_p$ of $f(X)$. The expansion parameter γ may depend on the degree d of $f(X)$ and bit length $\log(p)$ of p.*

Efficient evaluation: *The evaluation algorithm Eval has (worst-case) running time $\mathsf{poly}(\log(d), \log(p))$ where d is the degree of $f(X) \in \mathbb{Z}_p[X]$ and $\log(p)$ is the size of a coefficient $a_i \in \mathbb{Z}_p$ of $f(X)$ (recall that $f(X)$ and p are given as input to $\mathsf{GenData}$). The running time of Eval is measured in the RAM model of computation.*

Kedlaya and Umans [40] have proposed an efficient DS for univariate polynomial evaluation with expansion factor $\gamma = O(d^\delta \cdot \log^{o(1)}(p))$ where δ is an arbitrary positive constant. Below, we report the efficiency of their construction.

Corollary 3 ([40, Section 5] restated)**.** *For every positive constant $\delta > 0$, there exists an efficient DS for univariate polynomial evaluation with (γ)-expansion (Definition 6) defined as $\gamma = (d+1)^\delta \log^{o(1)}(p)$, where $\log(p)$ and d are the size of the prime p and the degree of $f(X) \in \mathbb{Z}_p$ (given in input to $\mathsf{GenData}$), respectively.*[11]

3.5 Incompressibility and Polynomial Evaluation

Next, we define the notion of incompressibility (w.r.t distributions) which defines how much a string x, sampled from a distribution $\mathbf{X}$, can be compressed.

[11] Observe that the expansion factor γ of $(d+1)^\delta \log^{o(1)}(p)$ implies that the overall size of D (output by $\mathsf{GenData}(f, p)$) is at most $(d+1)^{1+\delta} \log^{1+o(1)}(p)$. This follows by observing that $|\mathsf{D}| = |f(X)| \cdot \gamma = (d+1)\log(p)(d+1)^\delta \log^{o(1)}(p) = d^{1+\delta} \log^{1+o(1)}(p)$.

Definition 7 (Incompressibility)**.** *Let $\mathbf{X}$ be a distribution defined over $\{0,1\}^n$. We say that $\mathbf{X}$ is (c, ϵ)-incompressible if for every unbounded adversary $\mathsf{A} = (\mathsf{A}_1, \mathsf{A}_2)$, we have*

$$\mathbb{P}\Big[\mathsf{A}_2(1^\lambda, \alpha) = x \wedge |\alpha| \leq n - c : x \leftarrow\$\ \mathbf{X}_\lambda, \alpha \leftarrow\$\ \mathsf{A}_1(1^\lambda, x)\Big] \leq \epsilon.$$

Observe that the above definition considers adversaries with unbounded computation. It is known that the uniform distribution $\mathbf{U}_n$ over $\{0,1\}^n$ is $(c, \frac{1}{2^c})$-incompressible with respect to unbounded adversaries.

Theorem 1. *For every $n \in \mathbb{N}$ and for every $c \in \mathbb{N}$ such that $c \leq n$, the uniform distribution $\mathbf{U}_n$ over $\{0,1\}^n$ is $(c, \frac{1}{2^c})$-incompressible.*

We extend the above theorem to the setting of polynomial evaluation. In particular, we demonstrate that a randomly sampled polynomial $f(X) \leftarrow\$\ \mathbb{Z}_p[X]$ cannot be compressed in the following sense: *to compute $f(x)$ on a randomly sampled point x with sufficiently large probability, an adversary must have access to a sufficiently large string α (which encodes $f(X)$ or some pre-computed evaluations of $f(X)$).* Intuitively, this follows by observing that $d+1$ evaluations $(f(x_1), \dots, f(x_{d+1}))$ and their corresponding points $(x_1, \dots, x_{d+1})$ are an encoding of the random string $a = (a_0, \dots, a_d)$ composed of the $d+1$ coefficients of the polynomial $f(X) = \sum_{i=0}^{d} a_i \cdot X^i$.[12] This allows us to reduce the incompressibility of random polynomials to the incompressibility of random strings (Theorem 1). Below, we report the formal result whose proof appears in Appendix A.1. Also, we highlight that an analogous result has been demonstrated in [14, Section 3], but in a different setting, yielding a different bound.

Theorem 2. *Let p be a $(s_p + 1)$-bits prime and q be a (s_q)-bits prime where $q \leq p$. For every $u \in \mathbb{N}$, let $\mathbf{F}^u_{d-1,p}$ be a distribution (over the set of univariate polynomials of degree $d-1$ from $\mathbb{Z}_p[X]$) which samples u polynomials $f_1(X), \dots, f_u(X) \in \mathbb{Z}_p[X]$ as follows:*

- *Sample $(a_0, \dots, a_{u \cdot d - 1}) \leftarrow\$\ \mathbf{U}_{u \cdot d \cdot s_p}$ and return the u univariate polynomials $f_1(X), \dots, f_u(X)$ such that $f_j(X) = \sum_{i=0}^{d-1} a_{j \cdot d + i} \cdot X^i \in \mathbb{Z}_p[X]$ of degree $d-1$ for $j \in \{0\} \cup [u-1]$ (i.e., each a_i is interpreted as an element of $\mathbb{Z}_p$).*

For every $u \in \mathbb{N}$, for every $d \in \mathbb{N}$, for every $c \leq d(u \cdot s_p - s_q)$ and for every unbounded adversary $\mathsf{A} = (\mathsf{A}_1, \mathsf{A}_2)$, the following probability holds:

$$\mathbb{P}\left[\begin{array}{c} \mathsf{A}_2(1^\lambda, x, \alpha) = (f_1(x), \dots, f_u(x)) \wedge \\ |\alpha| \leq d(u \cdot s_p - s_q) - c \end{array} : \begin{array}{c} (f_i(X))_{i \in [u]} \leftarrow\$\ \mathbf{F}^u_{d-1,p}, \\ \alpha \leftarrow\$\ \mathsf{A}_1(1^\lambda, f_1, \dots, f_u), \\ x \leftarrow\$\ \mathbb{Z}_q \end{array}\right] \leq \frac{d-1}{|\mathbb{Z}_q|} + \frac{1}{2^c}.$$

On the best possible (asymptotic) security guarantees of Theorem 2. Fix $u = 1$ in Theorem 2 (i.e., the adversary must deal with a single random polynomial). We observe that the bound of $|\alpha|$ presents a loss proportional to $d \cdot s_q$ which depends on the size of the (single) polynomial (i.e., the degree d).[13] This loss exactly corresponds to the length of d points from $\mathbb{Z}_q \subseteq \mathbb{Z}_p$. Intuitively, this is because the adversary A_1 can compute α such that it will later allow A_2 to answer correctly only to some adversarially chosen points that may be correlated to the random polynomial $f(X) \leftarrow\$\ \mathbf{F}^1_{d-1,p}$ (this is because A_1 computes α while knowing $f(X)$).[14] The reason behind this loss is also discussed in [14, Sec. 3].

[12] From $(f(x_1), \dots, f(x_{d+1}))$ and $(x_1, \dots, x_{d+1})$ it is possible to reconstruct $a = (a_0, \dots, a_d)$ through Lagrange interpolation.

[13] For the sake of clarity, we ignore c that does not depend on either d or u.

[14] By looking at the proof of Theorem 2 in Appendix A.1, this loss is due to the need of encoding $(x_1, \dots, x_d)$ (which may be arbitrary and correlated to $f(X)$) in the string α output by the reduction. These points $(x_1, \dots, x_d)$ are essentially the ones on which the adversary A_2 correctly computes $(f(x_1), \dots, f(x_d))$. This is required in order to allow the reduction to correctly reconstruct $a = (a_0, \dots a_{d-1})$ and contradicts the $(c, \frac{1}{2^c})$-incompressibility of $\mathbf{U}_{d \cdot s_p}$ (recall we are assuming that $u = 1$).

In addition, we notice that the best possible security guarantee offered by Theorem 2 is when the adversarial advantage is small and the upper-bound of $|\alpha|$ is maximized (note that the upper-bound on $|\alpha|$ cannot go beyond $d \cdot u \cdot s_p$). In such a way, we are guaranteed that the adversary does not win even when it has access to a large pre-computed information (i.e., the string α). To this end, we choose to state Theorem 2 while taking into account that s_q (which defines the size of the (challenge) point space $\mathbb{Z}_q \subseteq \mathbb{Z}_p$) can be both (i) significantly smaller than s_p (which defines the size of a coefficient of the polynomials) to get a reasonably high upper-bound on $|\alpha|$, and (ii) large enough to get a significantly small (possibly negligible) upper-bound on the adversary's advantage.

The following corollary shows that this is possible asymptotically. In particular, we can set the upper-bound of the adversary's advantage to be exponentially small in λ while keeping the bound on $|\alpha|$ to be asymptotically close to the optimal value $u \cdot d \cdot s_p$ (i.e., the best possible security that can be achieved) by setting $s_p = s_p(\lambda) = \lambda^{1+\delta}$ (where δ is a positive constant) and $s_q = s_q(\lambda) = \lambda$. Below, we report the formal corollary whose proof is in Appendix A.2.

Corollary 4. *For every $\lambda \in \mathbb{N}$, for every $u = u(\lambda) \in \mathsf{poly}(\lambda)$, for every $d = d(\lambda) \in \mathsf{poly}(\lambda)$, for every $(\lambda^{1+\delta}+1)$-bits prime p where $\delta > 0$ is a constant, for every (λ)-bits prime q (note that $q < p$ by definition), we have:*

$$\mathbb{P}\left[\begin{array}{c} \mathsf{A}_2(1^\lambda, x, \alpha) = (f_1(x), \ldots, f_u(x)) \wedge \\ |\alpha| \leq d \cdot u \cdot \lambda^{1+\delta} - (d+1) \cdot \lambda \end{array} : \begin{array}{c} (f_i(X))_{i \in [u]} \leftarrow\$ \mathbf{F}^u_{d-1,p}, \\ \alpha \leftarrow\$ \mathsf{A}_1(1^\lambda, f_1, \ldots, f_u), \\ x \leftarrow\$ \mathbb{Z}_q \end{array}\right] \leq O\left(\frac{1}{2^\lambda}\right).$$

We stress that there are other combinations of parameters for s_p and s_q that allow for a negligible adversarial advantage. For example, it is sufficient to set $s_q \in \omega(\log(\lambda))$ and $s_p = \lambda$. However, these choices do not allow us to set the adversary's advantage to be exponentially small in λ as achieved in Corollary 4. Thus, we choose $s_q = s_q(\lambda) = \lambda$ and $s_p = s_p(\lambda) = \lambda^{1+\delta}$ to set the advantage to be at most $O(\frac{1}{2^\lambda})$ while getting an upper-bound on $|\alpha|$ which is asymptotically close to $d \cdot u \cdot s_p$ (i.e., the best possible security that can be achieved).

3.6 Pseudorandom Functions

A pseudorandom function (PRF) scheme $\Pi = (\mathsf{KGen}, \mathsf{F})$ with input space $\mathcal{X}$ and output space $\mathcal{Y}$ is composed of the following polynomial-time algorithms:

$\mathsf{KGen}(1^\lambda)$**:** The randomized key generation algorithm takes as input the security parameter 1^λ and outputs a key k.

$\mathsf{F}(\mathsf{k}, x)$**:** The deterministic function evaluation algorithm takes as input a key k and an input $x \in \mathcal{X}$, it outputs a value $y \in \mathcal{Y}$.

A PRF Π is considered secure (i.e., pseudorandom) if its output distribution is indistinguishable from the one of a truly random function.

Definition 8 (Security of PRF)**.** *A PRF Π with input space $\mathcal{X}$ and output space $\mathcal{Y}$ is (ϵ)-secure if for every PPT adversary A, we have:*

$$\left|\mathbb{P}\left[\mathsf{A}^{\mathsf{F}(\mathsf{k},\cdot)}(1^\lambda) = 1\right] - \mathbb{P}\left[\mathsf{A}^{\mathsf{F}_{\mathsf{rnd}}(\cdot)}(1^\lambda) = 1\right]\right| \leq \epsilon,$$

where $\mathsf{k} \leftarrow\$ \mathsf{KGen}(1^\lambda)$ and $\mathsf{F}_{\mathsf{rnd}} : \mathcal{X} \to \mathcal{Y}$ is a truly random function over $\mathcal{X}$ and $\mathcal{Y}$.

4 Localized (deterministic) RAM algorithms

Consider an input $x = (x_1, \dots, x_n)$ composed of n blocks where each x_i can be accessed in constant time in the RAM model of computation. Also, consider a RAM algorithm T with RAM access to x that, on input $y \in \{0,1\}^*$,[15] performs a computation, reading only a subset $\mathcal{X} \subset \{x_1, \dots, x_n\}$ of the blocks of x. In this section, we formally demonstrate that the exact same computation can be executed with only having y and the blocks $\mathcal{X}$ effectively read by T (i.e., the unread blocks are unnecessary) when T is deterministic.[16] To this end, we start by introducing the oracle notation for RAM algorithms which will allow us to formally demonstrate the above statement.

Oracle Notation for Deterministic algorithms in the RAM model of computation. We focus *only* on deterministic RAM algorithms. Let x be a *RAM accessible* input composed of multiple blocks $x = (x_1, \dots, x_n)$. We denote with $\mathsf{T}^{[x]}$ a RAM algorithm T with *read-only* RAM access to x. Moreover, let $y \in \{0,1\}^*$ be an arbitrary binary string. We denote with $\mathsf{T}^{[x]}(y)$ the deterministic execution of a RAM algorithm T with read-only RAM access to x but not to y, i.e., T must read y in its entirety. Below, we formally define how T interacts with its RAM accessible input.

Definition 9 (Oracle abstraction for deterministic RAM algorithms)**.** *A deterministic RAM algorithm* T *is an algorithm that performs computations by leveraging its RAM access to (some of) its inputs. Let* $x = (x_1, \dots, x_n)$ *be a read-only RAM accessible input* x *composed of* n *blocks and* $y \in \{0,1\}^*$ *be an arbitrary binary input string. The execution* $\mathsf{T}^{[x]}(y)$ *performs computations over* x *and* y *where* T *can read parts of* x *by interacting with the oracle* $[x]$ *as follows:*

- T *can send a* (read, i) *read command (for* $i \in [n]$*) to* $[x]$*. As a result,* T *receives the* i*-th block* x_i *from* $[x]$*.*

Definition 10 (Indexes read during a RAM computation)**.** *Let* T *be a deterministic RAM algorithm,* $x = (x_1, \dots, x_n)$ *be a read-only RAM accessible input* x *composed of* n *blocks and* $y \in \{0,1\}^*$ *be an arbitrary binary input string. We say that* $\mathcal{I}_{x,y} \subseteq [n]$ *is the* ordered *set of indexes read from* x *during the computation* $\mathsf{T}^{[x]}(y)$ *if the following conditions hold:*

Completeness: *Let* $(\mathsf{read}, i_1), \dots, (\mathsf{read}, i_{n'})$ *be the read commands submitted by* T *to* $[x]$ *during the RAM computation* $\mathsf{T}^{[x]}(y)$ *(note that* n' *may be greater than* n *since* T *can read a block multiple times). Then, we have that* $i_j \in \mathcal{I}_{x,y}$ *for every* $j \in [n']$ *where* $\mathcal{I}_{x,y}$ *is a set (i.e., no duplicate indexes).*

Ordering: *The set* $\mathcal{I}_{x,y} = \{i_1, \dots, i_k\}$ *is ordered, i.e.,* $\forall j \in [k-1]$ *we have* $i_j < i_{j+1}$ *where* $i_j, i_{j+1} \in \mathcal{I}_{x,y}$*.*

Localized (deterministic) RAM algorithms. The fact that the deterministic RAM computation $\mathsf{T}^{[x]}(y)$ computes output v by only reading the indexes $\mathcal{I}_{x,y}$ of the input $x = (x_1, \dots, x_n)$ implies that it is possible to compute v (in the RAM model) even without the blocks $(x_i)_{i \in [n] \setminus \mathcal{I}_{x,y}}$ where $x = (x_1, \dots, x_n)$. To this end, we define the notion of *localized RAM algorithm.* Intuitively, the localized version $\mathsf{Local.T}$ of the deterministic RAM algorithm T is, in turn, a deterministic RAM algorithm that is able to recompute the output of $\mathsf{T}^{[x]}(y)$ by taking as input the string y and the blocks $x_{i_1}, \dots, x_{i_k}$ read by $\mathsf{T}^{[x]}(y)$ from x.

[15] In this section it is sufficient to interpret y as an arbitrary binary string that needs to be read in its entirety, i.e., RAM access to y does not give any benefit to T.

[16] Otherwise, the final output and the blocks read from x may also depend on the random coins of T.

Somewhat more formally, if $\mathsf{T}^{[x]}(y) = v$ then $\mathsf{Local.T}^{[x'],[\mathsf{map}]}(y) = v$, where $\mathcal{I}_{x,y}$ is the ordered set of indexes read from x during the computation $\mathsf{T}^{[x]}(y)$ and $x' = (x'_1, \dots, x'_k) = (x_{i_1}, \dots, x_{i_k})$ is a read-only RAM accessible input. We note that the localized algorithm $\mathsf{Local.T}$ has access to an additional read-only RAM accessible input map which is essentially the memory mapping between $x' = (x'_1, \dots, x'_k)$ and $x = (x_1, \dots, x_n)$. In particular, map is defined as $\mathsf{map} = (i_j)_{i_j \in \mathcal{I}_{x,y}}$ and its required to let $\mathsf{Local.T}^{[x'],[\mathsf{map}]}(y)$ know that its j-th block x'_j of x' corresponds to the i_j-th block x_{i_j} of x (held by the original computation $\mathsf{T}^{[x]}(y)$).[17] We formalize the notion of localized RAM algorithms below.

Definition 11 (Localized RAM algorithms). *We say that a deterministic RAM algorithm* $\mathsf{Local.T}$ *is the* localized version *of the deterministic RAM algorithm* T *if the following conditions hold:*

Perfect correctness: *For every read-only RAM accessible input $x = (x_1, \dots, x_n)$, for every arbitrary binary input $y \in \{0,1\}^*$, let $\mathcal{I}_{x,y} = \{i_1, \dots, i_k\}$ be the* ordered *set of indexes read from x during the RAM computation $\mathsf{T}^{[x]}(y)$. Then, for every $k' \geq k$, for every memory mapping $\mathsf{map} = (i'_1, \dots, i'_{k'}) \subseteq [n]$, for every read-only RAM accessible input $x' = (x'_1, \dots, x'_{k'})$ such that*

- map *is ordered, i.e., $\forall j \in [k'-1]$ then $i'_j < i'_{j+1}$,*
- *$\forall i_j \in \mathcal{I}_{x,y}$ then $i_j \in \mathsf{map}$,*
- *$\forall i'_j \in \mathsf{map}$, if $i'_j \in \mathcal{I}_{x,y}$ then $x'_j = x_{i'_j}$,*

we have $\mathbb{P}\Big[\mathsf{T}^{[x]}(y) = \mathsf{Local.T}^{[x'],[\mathsf{map}]}(y)\Big] = 1.$

Invalid mapping: *For every read-only RAM accessible input $x = (x_1, \dots, x_n)$ and for every arbitrary binary input $y \in \{0,1\}^*$, let $\mathcal{I}_{x,y} = \{i_1, \dots, i_k\}$ be the ordered set of indexes read from x during the RAM computation T. Then, for every memory mapping $\mathsf{map} = (i'_1, \dots, i'_{k'}) \subseteq [n]$, for every read-only RAM accessible input $x' = (x'_1, \dots, x'_{k'})$ such that*

- map *is ordered, i.e., $\forall j \in [k'-1]$ then $i'_j < i'_{j+1}$,*
- *$\exists i_j \in \mathcal{I}_{x,y}$ such that $i_j \notin \mathsf{map}$,*
- *$\forall i'_j \in \mathsf{map}$ then $x'_j = x_{i'_j}$,*

we have $\mathbb{P}\Big[\mathsf{Local.T}^{[x'],[\mathsf{map}]}(y) = \perp\Big] = 1.$

Intuitively, *perfect correctness* says that $\mathsf{Local.T}^{[x'],[\mathsf{map}]}(y)$ performs the same computation of $\mathsf{T}^{[x]}(y)$ when map provides the correct mapping between x' and x for all the indexes $\mathcal{I}_{x,y}$ read by $\mathsf{T}^{[x]}(y)$ (observe that Definition 11 allows x' and map to additionally include blocks and indexes not read by $\mathsf{T}^{[x]}$. Still, this do not affect the computation of $\mathsf{Local.T}^{[x'],[\mathsf{map}]}(y)$).

On the other hand, *invalid mapping* says that $\mathsf{Local.T}^{[x'],[\mathsf{map}]}(y)$ outputs $\perp$ when map does not contain an index $i_j \notin \mathsf{map}$ which, instead, is read by $\mathsf{T}^{[x]}(y)$. Looking ahead, this property is fundamental to prove security of our verifiable data structure for univariate polynomial evaluation (Section 5).

The following theorem states (whose proof appears in Appendix A.3) that any deterministic RAM algorithm has its localized deterministic RAM algorithm. Moreover, the theorem also explicates the running time of the localized RAM algorithm in terms of the running time of the original one.

[17] This is essentially identical to how virtual memories work in practice.

Theorem 3. *If there exists a deterministic RAM algorithm* T*, then there exists a deterministic RAM algorithm* $\mathsf{Local.T}$ *which is the localized version of* T *(Definition 11). In addition, for every read-only RAM accessible input* x*, arbitrary binary input* y*, read-only RAM accessible input* x'*, and read-only RAM accessible memory mapping* map*, the running time of* $\mathsf{Local.T}^{[x'],[\mathsf{map}]}(y)$ *is at most* $t \cdot \log(|\mathsf{map}|)$ *where* t *is the running time of* $\mathsf{T}^{[x]}(y)$*. The running times of both* T *and* $\mathsf{Local.T}$ *are measured in the RAM model of computation.*

For the sake of clarity, in the remaining sections we drop the oracle notation $[x]$ used to denote a RAM accessible input. Thus, we will write $\mathsf{Local.T}(y, x)$ (instead of $\mathsf{Local.T}^{[x]}(y)$) when it is clear that x is the RAM accessible input.

5 Verifiable DS for Univariate Polynomial Evaluation

We extend the notion of DS for univariate polynomial evaluation (introduced in Section 3.4) by making it verifiable, i.e., making it possible to check that $y = f(x)$. Intuitively, the syntax of verifiable DS (VDS, in short) for univariate polynomial evaluation is analogous to that of (non-verifiable) DS except that the evaluation algorithm produces a proof π that can later be verified by the corresponding verification algorithm Verify. To make the verification process work, a VDS also has some public parameters pp (taken as input by all algorithms) and a digest h that is a succinct representative value of the data structure D.

Formally, a VDS for evaluation of univariate polynomials is composed of the following polynomial-time algorithms:

$\mathsf{Setup}(1^\lambda)$**:** On input the security parameter 1^λ, the randomized setup algorithm outputs the public parameters pp.

$\mathsf{GenData}(\mathsf{pp}, f, p)$**:** On input the public parameters pp, a univariate polynomial $f(X) = \sum_{i=0}^{d} a_i \cdot X^i \in \mathbb{Z}_p[X]$ (of degree $d \in \mathbb{N}$) and a prime $p \in \mathbb{N}$, the deterministic data structure generation algorithm outputs a data structure D, a digest h (of the data structure D), and auxiliary information aux (required to compute proofs of correctness).

$\mathsf{Eval}(\mathsf{pp}, x, \mathsf{D}, \mathsf{aux})$**:** On input the public parameters pp, a point $x \in \mathbb{Z}_p$, a data structure D, and auxiliary information aux, the deterministic evaluation algorithm outputs $y \in \mathbb{Z}_p$ and a proof π.

$\mathsf{Verify}(\mathsf{pp}, \mathsf{h}, x, y, \pi)$**:** On input the public parameters pp, a digest h, a point $x \in \mathbb{Z}_p$, a value $y \in \mathbb{Z}_p$ and a proof π, the deterministic verification algorithm outputs a decision bit $b \in \{0, 1\}$.

We assume that p is a prime (thus, $\mathbb{Z}_p$ is a field of prime order) for simplicity, since our PoRep will leverage such fields.

We require a VDS to satisfy the standard notions of correctness and completeness. The former says that VDS allows one to correctly compute $y = f(x)$, whereas the latter says that honestly generated proofs always verify. Differently from non-verifiable DS, a VDS additionally needs to satisfy soundness in order to be considered secure. At a high level, it is infeasible for a malicious evaluator to produce a proof π that verifies with respect to an incorrect output $y \neq f(x)$.

Definition 12 (Perfect correctness of VDS)**.** *A VDS* $\Pi = (\mathsf{Setup}, \mathsf{GenData}, \mathsf{Eval}, \mathsf{Verify})$ *for evaluation of univariate polynomials is perfectly correct if* $\forall \lambda \in \mathbb{N}$*,* $\forall$ *prime* $p \in \mathbb{N}$*,* $\forall f(X)$

$\in \mathbb{Z}_p[X]$, $\forall x \in \mathbb{Z}_p$, *the following probability holds:*

$$\mathbb{P}\left[y = f(x) : \begin{array}{c} \mathsf{pp} \leftarrow_{\$} \mathsf{Setup}(1^\lambda) \\ (\mathsf{D}, \mathsf{h}, \mathsf{aux}) = \mathsf{GenData}(\mathsf{pp}, f, p) \\ (y, \pi) = \mathsf{Eval}(\mathsf{pp}, x, \mathsf{D}, \mathsf{aux}) \end{array} \right] = 1$$

Definition 13 (Perfect completeness of VDS)**.** *A VDS* $\Pi = (\mathsf{Setup}, \mathsf{GenData}, \mathsf{Eval}, \mathsf{Verify})$ *for evaluation of univariate polynomials is perfectly complete if* $\forall\lambda \in \mathbb{N}$, $\forall$ *prime* $p \in \mathbb{N}$, $\forall f(X) \in \mathbb{Z}_p[X]$, $\forall x \in \mathbb{Z}_p$, *the following probability holds:*

$$\mathbb{P}\left[\mathsf{Verify}(\mathsf{pp}, \mathsf{h}, x, y, \pi) = 1 : \begin{array}{c} \mathsf{pp} \leftarrow_{\$} \mathsf{Setup}(1^\lambda) \\ (\mathsf{D}, \mathsf{h}, \mathsf{aux}) = \mathsf{GenData}(\mathsf{pp}, f, p) \\ (y, \pi) = \mathsf{Eval}(\mathsf{pp}, x, \mathsf{D}, \mathsf{aux}) \end{array} \right] = 1$$

Definition 14 (Soundness of VDS)**.** *A VDS* $\Pi = (\mathsf{Setup}, \mathsf{GenData}, \mathsf{Eval}, \mathsf{Verify})$ *for evaluation of univariate polynomials* (ϵ)*-sound if for every valid PPT adversary* A*, the following probability holds:*

$$\mathbb{P}\left[\mathsf{Verify}(\mathsf{pp}, \mathsf{h}, x, y, \pi) = 1 \wedge y \neq f(x) : \begin{array}{c} \mathsf{pp} \leftarrow_{\$} \mathsf{Setup}(1^\lambda) \\ (x, y, \pi, f, p) \leftarrow_{\$} \mathsf{A}(1^\lambda, \mathsf{pp}) \\ (\mathsf{D}, \mathsf{h}, \mathsf{aux}) = \mathsf{GenData}(\mathsf{pp}, f, p) \end{array} \right] \leq \epsilon.$$

An adversary A *is called valid if* $p \in \mathbb{N}$ *is a prime and* $f(X) \in \mathbb{Z}_p[X]$.[18]

Observe that the above definition is fully adaptive even if A does not take as input the tuple $(\mathsf{D}, \mathsf{h}, \mathsf{aux})$. This is because $\mathsf{GenData}$ is deterministic.

Lastly, we extend the notion of efficiency with (γ)-expansion of DS (see Definition 6) to the setting of VDS. The only difference is that (i) we consider a doubly-efficient VDS where both evaluation and verification run in time poly-logarithmic in the degree d of the polynomial, and (ii) the expansion is defined as $\gamma = \frac{|\mathsf{D}|+|\mathsf{aux}|}{|f(X)|}$, i.e., we consider as expansion any information (that depends on $f(X)$) required for evaluation which is capable of computing $y = f(x)$ and its corresponding proof π (observe that the computation of π requires knowledge of aux).[19] We do not include the public parameters pp in the expansion factor since they only depend on the security parameter.

Definition 15 (Double-efficiency with (γ)-expansion of VDS)**.** *A VDS* $\Pi = (\mathsf{Setup}, \mathsf{GenData}, \mathsf{Eval}, \mathsf{Verify})$ *for evaluation of univariate polynomials is doubly-efficient with* (γ)*-expansion if the following conditions hold:*

(γ)-expansion: *The size of* $(\mathsf{D}, \mathsf{aux})$ *(output by* $\mathsf{GenData}(\mathsf{pp}, f, p)$*) is bounded by* $|f(X)| \cdot \gamma = (d+1)\log(p) \cdot \gamma$ *where* d *is the degree of* $f(X) \in \mathbb{Z}_p[X]$ *and* $\log(p)$ *is the size of a coefficient* $a_i \in \mathbb{Z}_p$ *of* $f(X)$*. The expansion parameter* γ *may depend on the security parameter* λ*, the degree* d *of* $f(X)$*, and bit length* $\log(p)$ *of* p*.*

Efficient evaluation and verification: *Both* Eval *and* Verify *have (worst-case) running time* $\mathsf{poly}(\lambda, \log(d), \log(p))$ *where* d *is the degree of* $f(X) \in \mathbb{Z}_p[X]$ *and* $\log(p)$ *is the size of a coefficient* $a_i \in \mathbb{Z}_p$ *of* $f(X)$ *(recall that* $f(X)$ *and* p *are given in input to* $\mathsf{GenData}$*). The running time of both* Eval *and* Verify *is measured in the RAM model of computation.*

[18] We assume that p (output by A) is a prime only because we will leverage fields of prime order when building our PoRep scheme. Hence, this definition can be extended to any field (e.g., composite p).

[19] Recall that in DS the only additional information was the expansion of D introduced to handle fast evaluation. See Definition 15.

5.1 (Doubly-efficient) VDS from DS and VC

We build a doubly-efficient VDS (Section 5 and Definition 15) from any efficient DS for evaluation of univariate polynomials (Section 3.4 and Definition 6) and efficient VC schemes (Section 3.3 and Definition 2). At a high level, the construction leverages the fact that a verifier can check $y \overset{?}{=} f(x)$ by using $\mathsf{Local.Eval}_{\mathsf{DS}}$ which is the (deterministic) localized RAM version of the (deterministic) RAM evaluation algorithm of DS (see Definition 11) as described in the technical overview (Section 1.2). The formal construction follows.

Construction 1. *Consider the following ingredients:*

1. *A DS scheme $\Pi_{\mathsf{DS}} = (\mathsf{GenData}_{\mathsf{DS}}, \mathsf{Eval}_{\mathsf{DS}})$ for evaluation of univariate polynomials. Without loss of generality, we assume that the output space of $\mathsf{GenData}_{\mathsf{DS}}$ is $\{0,1\}^{\ell \cdot z}$, i.e., the (read-only RAM accessible) data structure $\mathsf{D} = (\mathsf{D}_1, \ldots, \mathsf{D}_\ell)$ is composed of $\ell = \ell(d)$ blocks each of size z (for some arbitrary $z \in \mathbb{N}$).*[20] *Observe that the degree d of $f(X)$ (the polynomial given in input to $\mathsf{GenData}_{\mathsf{DS}}$) affects the length of the data structure $\mathcal{D}$ (see Corollary 3) and, for this reason, $\ell = \ell(d)$ is a function of d.*

2. *A deterministic RAM algorithm $\mathsf{Local.Eval}_{\mathsf{DS}}$ that is the localized version of the deterministic RAM algorithm $\mathsf{Eval}_{\mathsf{DS}}$ of Π_{DS} (Definition 11).*

3. *A VC scheme $\Pi_{\mathsf{VC}} = (\mathsf{Setup}_{\mathsf{VC}}, \mathsf{Commit}_{\mathsf{VC}}, \mathsf{Open}_{\mathsf{VC}}, \mathsf{Verify}_{\mathsf{VC}})$ with message space $\{0,1\}^z$ where z is the block size of the output space of $\mathsf{GenData}_{\mathsf{DS}}$ (as defined in Item 1).*

We build a VDS scheme Π for evaluation of univariate polynomials as follows:

$\mathsf{Setup}(1^\lambda)$**:** *On input the security parameter 1^λ, the randomized setup algorithm outputs $\mathsf{pp} = \mathsf{pp}_{\mathsf{VC}} \leftarrow\$ \mathsf{Setup}_{\mathsf{VC}}(1^\lambda)$.*

$\mathsf{GenData}(\mathsf{pp}, f, p)$**:** *On input the public parameters $\mathsf{pp} = \mathsf{pp}_{\mathsf{VC}}$, a univariate polynomial $f(X) = \sum_{i=0}^{d} a_i \cdot X^i \in \mathbb{Z}_p[X]$ of degree d, and a prime $p \in \mathbb{N}$, the deterministic data structure generation algorithm computes $\mathsf{D} = (\mathsf{D}_1, \ldots, \mathsf{D}_\ell) = \mathsf{GenData}_{\mathsf{DS}}(f, p)$ and $(\mathsf{c}, \mathsf{aux}) = \mathsf{Commit}_{\mathsf{VC}}(\mathsf{pp}_{\mathsf{VC}}, (\mathsf{D}_1, \ldots, \mathsf{D}_\ell))$ (recall that $\ell = \ell(d)$). Finally, it outputs the data structure D, the digest $\mathsf{h} = \mathsf{c}$, and the auxiliary information aux.*

$\mathsf{Eval}(\mathsf{pp}, x, \mathsf{D}, \mathsf{aux})$**:** *On input the public parameters $\mathsf{pp} = \mathsf{pp}_{\mathsf{VC}}$, a point $x \in \mathbb{Z}_p$, a data structure D, and auxiliary information aux, the deterministic evaluation algorithm proceeds as follows:*

1. *Execute $\mathsf{Eval}_{\mathsf{DS}}(x, \mathsf{D}) = y$ and let $\mathcal{I}_{x,\mathsf{D}} = \{i_1, \ldots, i_k\}$ be the ordered set of indexes read from D during the computation $\mathsf{Eval}_{\mathsf{DS}}(x, \mathsf{D})$ (recall that $\mathsf{Eval}_{\mathsf{DS}}$ is a RAM algorithm. See Definitions 9 and 10).*
2. *For $j \in \mathcal{I}_{x,\mathsf{D}}$, compute $\pi_j = \mathsf{Open}_{\mathsf{VC}}(\mathsf{pp}_{\mathsf{VC}}, \mathsf{D}_j, j, \mathsf{aux})$.*

Finally, it outputs $y \in \mathbb{Z}_p$ and $\pi = (\mathcal{I}_{x,\mathsf{D}}, \{\mathsf{D}_j\}_{j \in \mathcal{I}_{x,\mathsf{D}}}, \{\pi_j\}_{j \in \mathcal{I}_{x,\mathsf{D}}})$.

$\mathsf{Verify}(\mathsf{pp}, \mathsf{h}, x, \pi)$**:** *On input the public parameters $\mathsf{pp} = \mathsf{pp}_{\mathsf{VC}}$, a digest $\mathsf{h} = \mathsf{c}$, a point $x \in \mathbb{Z}_p$, a value $y \in \mathbb{Z}_p$, and a proof $\pi = (\mathcal{I}_{x,\mathsf{D}}, \{\mathsf{D}_j\}_{j \in \mathcal{I}_{x,\mathsf{D}}}, \{\pi_j\}_{j \in \mathcal{I}_{x,\mathsf{D}}})$, the deterministic verification algorithm proceeds as follows:*

1. *Check that $|\mathcal{I}_{x,\mathsf{D}}| \leq \ell$ and $\mathcal{I}_{x,\mathsf{D}} = \{i_1, \ldots, i_k\}$ is ordered. If not, return 0.*
2. *For $j \in \mathcal{I}_{x,\mathsf{D}}$, compute $\mathsf{Verify}_{\mathsf{VC}}(\mathsf{pp}_{\mathsf{VC}}, \mathsf{c}, \mathsf{D}_j, j, \pi_j) = b_j$.*

[20] This means that a z-bits size block of D can be read in constant time in the RAM model of computation.

3. *Execute the localized algorithm* $\mathsf{Local.Eval}_{\mathsf{DS}}(x, (\mathsf{D}_{i_1}, \dots, \mathsf{D}_{i_k}), \mathsf{map}) = y'$ *where* $\mathsf{map} = (i_1, \dots, i_k)$.[21]

Finally, the verification algorithm outputs 1 *if* $y = y'$ *and* $b_j = 1$ *for every* $j \in \mathcal{I}_{x,\mathsf{D}}$. *Otherwise, it outputs* 0.

Below, we report the results regarding correctness, completeness, and soundness of Construction 1. The formal proofs appear in Appendices A.4 and A.5.

Theorem 4. *Let* Π_{DS}, $\mathsf{Local.Eval}_{\mathsf{DS}}$, *and* Π_{VC} *as defined in Construction 1.*

1. *If* Π_{DS} *is perfectly correct (Definition 5) then* Π *of Construction 1 is perfectly correct (Definition 12).*
2. *If* Π_{VC} *is perfectly correct (Definition 3) and* $\mathsf{Local.Eval}_{\mathsf{DS}}$ *is perfectly correct (Definition 11) then* Π *of Construction 1 is perfectly complete (Definition 13).*

Theorem 5. *Let* Π_{DS}, $\mathsf{Local.Eval}_{\mathsf{DS}}$, *and* Π_{VC} *as defined in Construction 1. If* Π_{DS} *is perfectly correct (Definition 5),* $\mathsf{Local.Eval}_{\mathsf{DS}}$ *satisfies the invalid mapping property (Definition 11),* Π_{VC} *is perfectly correct (Definition 3) and* (ϵ_{VC})*-position binding (Definition 4), then* Π *from Construction 1 is* $(\ell \cdot \epsilon_{\mathsf{VC}})$*-sound.*

Construction 1 is doubly-efficient if both the underlying DS and VC scheme are efficient. Moreover, the expansion factor of Construction 1 depends of the expansion factor of DS and the size of aux generated by the VC scheme (observe that aux is needed to correctly compute a proof π). Below, we state the formal result whose proof is deferred to Appendix A.7.

Theorem 6. *If* Π_{DS} *is efficient with* (γ_{DS})*-expansion (Definition 6) and* Π_{VC} *is efficient (Definition 2) then* Π *of Construction 1 is doubly-efficient with* (γ)*-expansion for* $\gamma = \gamma_{\mathsf{DS}} + \frac{|\mathsf{aux}|}{|f(X)|}$, *where* $f(X)$ *is the univariate polynomial taken into account and* aux *is the auxiliary information generated by* Π_{VC}.

The following corollary is obtained by combining Theorem 6 and Corollaries 2 and 3 (see Appendix A.6 for the formal proof).

Corollary 5. *Under the collision resistant hash function assumption, for every positive constant* $\delta > 0$, *there exists a VDS for evaluation of univariate polynomials that is* $(\mathsf{negl}(\lambda))$*-sound and doubly-efficient with* (γ)*-expansion for* $\gamma = 2(d+1)^{\delta} \log^{o(1)}(p)$ *where* $\log(p)$ *and* d *are the size of the prime* p *and the degree of* $f(X) \in \mathbb{Z}_p$ *(given in input to* $\mathsf{GenData}$*), respectively. Moreover, we have that* $|\mathsf{h}| = \lambda$, *i.e., digests are succinct.*

6 Proof-of-Replication

A proof-of-replication (PoRep) scheme allows a verifier to efficiently check that a prover is using a significant amount of space to store an arbitrary message $m \in \mathcal{M}$.[22] The space required to store m must be sufficiently large even if m is highly compressible. Moreover, PoRep guarantees that m can be retrieved if the prover passes the verification process. PoRep was previously proposed in [31]. Our syntax and security definitions below generally capture the same properties, but they are tailored to reflect the objectives and contributions of this work.

[21] Recall that $\mathsf{Local.Eval}_{\mathsf{DS}}(x, (\mathsf{D}_{i_1}, \dots, \mathsf{D}_{i_k}), \mathsf{map})$ corresponds to $\mathsf{Local.Eval}_{\mathsf{DS}}^{[\mathsf{D}_{i_1}, \dots, \mathsf{D}_{i_k}], [\mathsf{map}]}(x)$ using the oracle abstraction introduced in Section 4.

[22] We use the term "message" to refer to a file that needs to be stored.

Formally, we define a PoRep scheme with message space $\mathcal{M}$, identifier space $\mathcal{I}$, and challenge space $\mathcal{C}$ to consist of the following polynomial-time algorithms:[23]

$\mathsf{Setup}(1^\lambda, 1^t)$**:** On input the security parameter 1^λ and the time parameter 1^t, the randomized setup algorithm outputs a public encoding key ek, a public proving key pk, and a public verification key vk.

$\mathsf{Encode}(\mathsf{ek}, m, \mathsf{id})$**:** On input the public encoding key ek, a message $m \in \mathcal{M}$, and an identifier $\mathsf{id} \in \mathcal{I}$ (for the message m), the deterministic encoding algorithm outputs an encoding c (of the message m with associated identifier id) and a digest h (of the encoding Encode).

$\mathsf{Prove}(\mathsf{pk}, \mathsf{chall}, \mathsf{c})$**:** On input the public proving key pk, a challenge $\mathsf{chall} \in \mathcal{I}$, and an encoding c, the deterministic proving algorithm outputs a proof π.

$\mathsf{Verify}(\mathsf{vk}, \mathsf{h}, \mathsf{chall}, \pi)$**:** On input the public verification key vk, a digest h, a challenge chall, and a proof π, the deterministic algorithm outputs $b \in \{0, 1\}$.

$\mathsf{Decode}(\mathsf{ek}, \mathsf{c}, \mathsf{id})$**:** On input the public encoding key ek, an encoding c, and an identifier $\mathsf{id} \in \mathcal{I}$, the deterministic decoding algorithm outputs $m \in \mathcal{M}$.

We require a PoRep scheme to satisfy the standard notions of correctness and completeness. The former says that an honest execution (of PoRep's algorithms) allows to correctly decode the message whereas the latter says that honest proofs always verify.

Definition 16 (Perfect correctness of PoRep)**.** *A PoRep* $\Pi = (\mathsf{Setup}, \mathsf{Encode}, \mathsf{Prove}, \mathsf{Verify}, \mathsf{Decode})$ *with message space* $\mathcal{M}$*, identifier space* $\mathcal{I}$*, and challenge space* $\mathcal{C}$ *is perfectly correct if* $\forall \lambda \in \mathbb{N}$*,* $\forall t \in \mathbb{N}$*,* $\forall m \in \mathcal{M}$*,* $\forall \mathsf{id} \in \mathcal{I}$*, the following probability holds:*

$$\mathbb{P}\left[\mathsf{Decode}(\mathsf{ek}, \mathsf{c}, \mathsf{id}) = m : \begin{array}{c} (\mathsf{ek}, \mathsf{pk}, \mathsf{vk}) \leftarrow\!\$\ \mathsf{Setup}(1^\lambda, 1^t) \\ (\mathsf{c}, \mathsf{h}) = \mathsf{Encode}(\mathsf{ek}, m, \mathsf{id}) \end{array}\right] = 1.$$

Definition 17 (Perfect completeness of PoRep)**.** *A PoRep* $\Pi = (\mathsf{Setup}, \mathsf{Encode}, \mathsf{Prove}, \mathsf{Verify}, \mathsf{Decode})$ *with message space* $\mathcal{M}$*, identifier space* $\mathcal{I}$*, and challenge space* $\mathcal{C}$ *is perfectly complete if* $\forall \lambda \in \mathbb{N}$*,* $\forall t \in \mathbb{N}$*,* $\forall m \in \mathcal{M}$*,* $\forall \mathsf{id} \in \mathcal{I}$*,* $\forall \mathsf{chall} \in \mathcal{C}$*, the following probability holds:*

$$\mathbb{P}\left[\mathsf{Verify}(\mathsf{vk}, \mathsf{h}, \mathsf{chall}, \pi) = 1 : \begin{array}{c} (\mathsf{ek}, \mathsf{pk}, \mathsf{vk}) \leftarrow\!\$\ \mathsf{Setup}(1^\lambda, 1^t) \\ (\mathsf{c}, \mathsf{h}) = \mathsf{Encode}(\mathsf{ek}, m, \mathsf{id}) \\ \pi = \mathsf{Prove}(\mathsf{pk}, \mathsf{chall}, \mathsf{Encode}) \end{array}\right] = 1.$$

In addition, we are interested in PoRep protocols that are doubly-efficient, i.e., the running times of both Eval and Verify are poly-logarithmic in the size $|m|$ of m in the RAM model of computation. Analogously to VDS, the encoding c may be bigger than m in order to achieve the above double-efficiency property. Thus, we extend the notion of (γ)-expansion to PoRep except that γ is defined with respect to $|m|$, i.e., $\gamma = \frac{|\mathsf{c}|}{|m|}$ (this means that $|c| = |m| \cdot \gamma$).

Definition 18 (Double-efficiency with (γ)-expansion of PoRep)**.** *A PoRep* $\Pi = (\mathsf{Setup}, \mathsf{Encode}, \mathsf{Prove}, \mathsf{Verify}, \mathsf{Decode})$ *with message space* $\mathcal{M}$*, identifier space* $\mathcal{I}$*, and challenge space* $\mathcal{C}$ *is doubly-efficient with* (γ)*-expansion if the following conditions hold:*

[23] Following [31], if needed, one can consider PoRep scheme with an additional message preprocessing algorithm (e.g., encryption of the message) executed by the data owner, or a polling algorithm when challenges are structured.

(γ)-expansion: *The size of* c *(output by* $\mathsf{Encode}(\mathsf{ek}, m, \mathsf{id})$*) is bounded by* $|m| \cdot \gamma$ *where* m *is the encoded message. The expansion parameter* γ *may depend on the security parameter* λ *and the size* $|m|$ *of* m*.*

Efficient proving and verification: *Both* Prove *and* Verify *have (worst-case) running time* $\mathsf{poly}(\lambda, \log(|m|))$ *where* $|m|$ *is the size of the encoded message* $m \in \mathcal{M}$ *(recall that* m *is given as input to* Encode*). The running time of both* Prove *and* Verify *are measured in the RAM model of computation.*

We now turn on security. A PoRep must satisfy two notions, named *replication* and *extraction*, which we formally define in the remainder of this section.

```
$\mathbf{G}^{\mathsf{replicate}}_{\mathsf{A},\Pi}(\lambda, t, n, u)$

$(\mathsf{ek}, \mathsf{pk}, \mathsf{vk}) \leftarrow\!\!\$\ \mathsf{Setup}(1^\lambda, 1^t)$
$(m_1, \ldots, m_u, \mathsf{state}) \leftarrow\!\!\$\ \mathsf{A}_1(1^\lambda, \mathsf{ek}, \mathsf{pk}, \mathsf{vk})$
$(\mathsf{id}_1, \ldots, \mathsf{id}_u) \leftarrow\!\!\$\ \mathcal{I}^u$
$\forall i \in [u],\ (\mathsf{c}_i, \mathsf{h}_i) = \mathsf{Encode}(\mathsf{ek}, m_i, \mathsf{id}_i)$
$\alpha \leftarrow\!\!\$\ \mathsf{A}_2(1^\lambda, (\mathsf{id}_i, \mathsf{h}_i, \mathsf{c}_i)_{i \in [u]}, \mathsf{state})$
$\mathsf{chall} \leftarrow\!\!\$\ \mathcal{C}$
$(\pi_1, \ldots, \pi_u) \leftarrow\!\!\$\ \mathsf{A}_3(1^\lambda, \mathsf{ek}, \mathsf{pk}, \mathsf{vk}, (\mathsf{id}_i)_{i \in [u]}, \mathsf{chall}, \alpha)$
**If** $\forall i \in [u],\ \mathsf{Verify}(\mathsf{vk}, \mathsf{h}_i, \mathsf{chall}, \pi_i) = 1$ : **return** 1
**Otherwise** : **return** 0
```

Figure 1: Experiment defining (ϵ, σ, n, u)-replication of PoRep.

Replication of PoRep. In a nutshell, a PoRep must force a prover to use a memory of size n (to store $m \in \mathcal{M}$) in order to produce a proof that verifies. For any adversarially chosen message m, an adversary cannot compress an honestly computed encoding c (output by $\mathsf{Encode}(\mathsf{ek}, m, \mathsf{id})$) into a string α (i.e., the memory) of size n while passing the verification process. The same guarantee must hold even if the adversary is required to store $u > 1$ (possibly identical) messages $(m_1, \ldots, m_u)$ and pass the verification for each of those messages. Naturally, in this case, we have that the memory bound $n = n(u)$ is a function of the number of messages u (optimally we would like to have that n scales linearly in u).

As already discussed in the technical overview, we consider trapdoorless PoRep. Thus, the above notion cannot be achieved without restricting the behavior of the adversary (see the PRF-based attack described in Section 1.2).

For this reason, we consider PoRep with a "slow" encoding algorithm Encode (the slowness of Encode can be tuned by setting the time parameter t chosen on Setup), and we restrict the adversary to produce a valid proof in less time than the one required to execute Encode. In the trapdoorless setting, several works [30, 29, 11, 50, 6, 48, 31] have considered adversaries with restricted running time. The formal definition follows.

Definition 19 (Replication of PoRep)**.** *Let* $\sigma(\lambda, t, n) = \sigma$ *be a polynomial function of the security parameter* λ*, the time parameter* t*, and the memory bound* n*, and* $n(\lambda, u) = n$ *be a function that depends on the security parameter* λ *and the number of messages* $u \in \mathbb{N}$*. A PoRep* $\Pi = (\mathsf{Setup}, \mathsf{Encode}, \mathsf{Prove}, \mathsf{Verify}, \mathsf{Decode})$ *with message space* $\mathcal{M}$*, identifier space* $\mathcal{I}$*, and challenge space* $\mathcal{C}$ *satisfies* (ϵ, σ, n, u)*-replication if for every valid PPT adversary* $\mathsf{A} = (\mathsf{A}_1, \mathsf{A}_2, \mathsf{A}_3)$*, then* $\mathbb{P}\left[\mathbf{G}^{\mathsf{replicate}}_{\mathsf{A},\Pi}(\lambda, t, n, u) = 1\right] \leq \epsilon$ *where* $\mathbf{G}^{\mathsf{replicate}}_{\mathsf{A},\Pi}$ *is depicted in Figure 1.*

An adversary $\mathsf{A} = (\mathsf{A}_1, \mathsf{A}_2, \mathsf{A}_3)$ *is called valid if* $|\alpha| \leq n$ *and* A_3 *runs in parallel time* σ *with* $\mathsf{poly}(t)$ *processors.*

Observe that both A_1 and A_2 are unrestricted; thus, they can perform any polynomial-time computation (even running Encode over multiple adversarially chosen messages and identifiers). Also, in addition to α, A_3 takes as input anything that is not produced by Encode. This means that A_2 only needs to encode in α (in an compressed fashion) Encode's output. Finally, we associate a random but public identifier id only after A_1 has committed on the challenge message m (note that both A_2 and A_3 take id as input). This allows Encode to have some randomness (i.e., the identifier) which is not correlated to the chosen message m, allowing it to produce an incompressible encoding c which is fundamental in order to achieve Definition 19.[24] Practical implementations of random identifiers in decentralized systems (such as blockchain systems) are random beacons, hashing the last blocks of a blockchains, or hashing the message $\mathsf{id} = \mathsf{H}(m)$.

Remark 2 (On the honest execution of Encode)**.** *In Definition 19, we assume that* Encode *is honestly executed. This is because the digest* h *is fundamental in order to have a sound verification process (*h *provides a binding guarantee on the original* m*). However, this setting is not compatible with decentralized scenarios (e.g., blockchain systems) in which the prover (which can be malicious) is entitled to run* Encode*. Still, we highlight that standard techniques for verifying computations (such a SNARKs) can be used to efficiently verify that* h *has been honestly computed (note that* h*'s computation does not require any secret, making verification easier). For the sake of exposition, we choose not to deal with malicious executions of* Encode *since the main objective of this paper is to propose a novel approach for PoRep verification phase.*

Memory gap of PoRep. Let u be the number of messages to be stored. The memory gap of PoRep is defined as the distance between the sizes $|m_1| + \ldots + |m_u|$ of the u messages to be stored and the maximum parameter n for which the PoRep satisfies replication (with respect to u messages) with negligible adversarial advantage (formally, $(\mathsf{negl}(\lambda), \sigma, n, u)$-replication for some σ). The notion of gap is useful for comparison between different PoReps schemes.

Definition 20 ((η)-gap of PoRep)**.** *A PoRep* $\Pi = (\mathsf{Setup}, \mathsf{Encode}, \mathsf{Prove}, \mathsf{Verify}, \mathsf{Decode})$ *with message space* $\mathcal{M}$*, identifier space* $\mathcal{I}$*, and challenge space* $\mathcal{C}$ *has* (η)*-gap if* Π *satisfies* $(\mathsf{negl}(\lambda), n, \sigma, u)$*-replication and* $n = (1 - \eta)(u \cdot \log(|\mathcal{M}|))$ *where* $\log(|\mathcal{M}|)$ *is the length of the messages supported by* Π*. The gap parameter* $\eta = \eta(\lambda, u, \log(|\mathcal{M}|))$ *can depend on the security parameter* λ*, the number of encoded messages* u*, and the length of the supported messages* $\log(|\mathcal{M}|)$*.*

Intuitively, the smaller the gap, the better is the robustness of the PoRep scheme. This is because an adversary, to pass the verification with non-negligible probability, is forced to use a memory which is close to the sizes of the u messages which it is entitled to store (where closeness is defined by the gap parameter η).

Extractability of PoRep. We now turn on extraction. PoRep is extractable if there is a universal extractor Ext that, given $(\mathsf{ek}, \mathsf{vk}, \mathsf{h}, \mathsf{id})$ and oracle access to the adversary, is able to extract the encoded message m. Naturally, this must hold only when the adversary is able to produce verifying proofs with respect to h, i.e., the digest of the encoding of m. We use a standard definition of extraction, we consider (i) universal PPT extractors, and (ii) adversary with noticeable (i.e., non-negligible) probability in producing verifying proofs.[25]

[24] If m is correlated to id, then c has no chance to be incompressible, Encode is deterministic.

[25] If the adversary passes the verification with negligible probability then extraction cannot be guaranteed. Indeed, any adversary (even the one that does not know m) can pass the verification with negligible probability

```
$\mathbf{G}^{\mathsf{extract}}_{\mathsf{A},\mathsf{Ext},\Pi}(\lambda, t, u)$

$(\mathsf{ek}, \mathsf{pk}, \mathsf{vk}) \leftarrow\!\!\$\ \mathsf{Setup}(1^\lambda, 1^t)$
$(m_1, \ldots, m_u, \mathsf{state}) \leftarrow\!\!\$\ \mathsf{A}_1(1^\lambda, \mathsf{ek}, \mathsf{pk}, \mathsf{vk})$
$(\mathsf{id}_1, \ldots, \mathsf{id}_u) \leftarrow\!\!\$\ \mathcal{I}^u$
$\forall i \in [u],\ (\mathsf{c}_i, \mathsf{h}_i) = \mathsf{Encode}(\mathsf{ek}, m_i, \mathsf{id}_i)$
$\alpha \leftarrow\!\!\$\ \mathsf{A}_2(1^\lambda, (\mathsf{id}_i, \mathsf{h}_i, \mathsf{c}_i)_{i \in [u]}, \mathsf{state})$
$(m'_1, \ldots, m'_u) \leftarrow\!\!\$\ \mathsf{Ext}^{\mathsf{A}_3(\cdot)}(1^\lambda, \mathsf{ek}, \mathsf{vk}, (\mathsf{h}_i, \mathsf{id}_i)_{i \in [u]})$
If $\forall i \in [u],\ m'_i = m_i$ : return $1$
Otherwise : return $0$
```

Figure 2: Experiment defining extractability of PoRep. The extractor Ext of $\mathbf{G}^{\mathsf{extract}}_{\mathsf{A},\mathsf{Ext},\Pi}(\lambda, t, u)$ has oracle access to $\mathsf{A}_3(\cdot)$ which is defined as $\mathsf{A}_3(\cdot) = \mathsf{A}_3(1^\lambda, \mathsf{ek}, \mathsf{pk}, \mathsf{vk}, (\mathsf{id}_i)_{i \in [u]}, \cdot, \alpha)$, i.e., Ext can only submit challenges $\mathsf{chall} \in \mathcal{C}$ to $\mathsf{A}_3(\cdot)$.

Below, we report the formal definition of extraction of PoRep. For the sake of clarity, we directly define extraction in the asymptotic setting since the definition only depends on λ (i.e., we do not put any time and memory restriction on the adversary except from being polynomial in λ).

Definition 21 (Extractability of PoRep)**.** *A PoRep $\Pi = (\mathsf{Setup}, \mathsf{Encode}, \mathsf{Prove}, \mathsf{Verify}, \mathsf{Decode})$ with message space $\mathcal{M}$, identifier space $\mathcal{I}$, and challenge space $\mathcal{C}$ is extractable if there exists an universal PPT extractor Ext such that $\forall \lambda \in \mathbb{N}$, $\forall t \in \mathsf{poly}(\lambda)$, $\forall u \in \mathsf{poly}(\lambda)$, $\forall n \in \mathsf{poly}(\lambda)$, and for every PPT adversary $\mathsf{A} = (\mathsf{A}_1, \mathsf{A}_2, \mathsf{A}_3)$, the following condition holds:*

$$\mathbb{P}\Big[\mathbf{G}^{\mathsf{replicate}}_{\mathsf{A},\Pi}(\lambda, t, n, u) = 1\Big] \geq \frac{1}{\mathsf{poly}(\lambda)} \Longrightarrow$$
$$\mathbb{P}\big[\mathbf{G}^{\mathsf{extract}}_{\mathsf{A},\mathsf{Ext},\Pi}(\lambda, t, u) = 1\big] \geq 1 - \mathsf{negl}(\lambda), \qquad (1)$$

where the both experiments $\mathbf{G}^{\mathsf{replicate}}_{\mathsf{A},\Pi}(\lambda, t, n, u)$ and $\mathbf{G}^{\mathsf{extract}}_{\mathsf{A},\mathsf{Ext},\Pi,d}(\lambda, t, u)$ are depicted in Figure 2. We stress that A does not need to be valid with respect to experiment $\mathbf{G}^{\mathsf{replicate}}_{\mathsf{A},\Pi}(\lambda, t, n, u)$ as defined in Definition 19 (i.e., Ext is able to extract independently from the running time and the memory used by the adversary).

We highlight that the head of the implication (Equation (1)) implies that the adversary passes the verification with non-negligible probability (see experiment $\mathbf{G}^{\mathsf{replicate}}_{\mathsf{A},\Pi}(\lambda, t, n, u)$ in Figure 2). Moreover, Ext of $\mathbf{G}^{\mathsf{extract}}_{\mathsf{A},\mathsf{Ext},\Pi,d}(\lambda, t, u)$ (Figure 2) has only oracle access to A_3 and no a-priori knowledge about $(m_1, \ldots, m_u)$ (the messages that Ext needs to extract).

6.1 PoRep from (input-dependent pre-processing) MHF and VDS

Next, we build a PoRep scheme from MHF (with input-dependent pre-processing), VDS, and an hash function H modelled as a RO.

Construction 2. *Consider the following ingredients:*

1. *A prime p of $(s_p + 1)$-bits and a prime q of (s_q)-bits where $q \leq p$ (by definition $\mathbb{Z}_q \subseteq \mathbb{Z}_p$ when $q \leq p$), $s_p(\lambda) = s_p$ and $s_q(\lambda) = s_q$ two polynomials in the security parameter. We assume that s_q is at least $\omega(\log(\lambda))$ w.l.o.g.*

by simply guessing the verifying proof π.

2. *A MHF scheme* $\Pi_{\mathsf{MHF}} = (\mathsf{Setup}_{\mathsf{MHF}}, \mathsf{Eval}_{\mathsf{MHF}})$ *with input space* $\mathcal{X}_{\mathsf{MHF}}$ *and output space* $\mathcal{Y}_{\mathsf{MHF}}$.

3. *A VDS* $\Pi_{\mathsf{VDS}} = (\mathsf{Setup}_{\mathsf{VDS}}, \mathsf{GenData}_{\mathsf{VDS}}, \mathsf{Eval}_{\mathsf{VDS}}, \mathsf{Verify}_{\mathsf{VDS}})$ *for evaluation of univariate polynomials.*

4. *A hash function* $\mathsf{H} : \mathcal{Y}_{\mathsf{MHF}} \times \mathcal{X}_{\mathsf{MHF}} \to \{0,1\}^{d \cdot s_p}$ *modeled as a RO.*

We build a PoRep scheme Π *with message space* $\mathcal{M} = \{0,1\}^{d \cdot s_p}$ *(for any* $d < q$*), identifier space* $\mathcal{I} = \mathcal{X}_{\mathsf{MHF}}$*, and challenge space* $\mathcal{C} = \mathbb{Z}_q$*, as follows:*

$\mathsf{Setup}(1^\lambda, 1^t)$**:** *On input the security parameter* 1^λ *and the time parameter* 1^t*, the randomized setup algorithm outputs a public encoding key* $\mathsf{ek} = (\mathsf{pp}_{\mathsf{MHF}}, \mathsf{pp}_{\mathsf{VDS}})$*, a public proving key* $\mathsf{pk} = \mathsf{pp}_{\mathsf{VDS}}$*, and a public verification key* $\mathsf{vk} = \mathsf{pp}_{\mathsf{VDS}}$ *where* $\mathsf{pp}_{\mathsf{MHF}} \leftarrow\!\!\$\ \mathsf{Setup}_{\mathsf{MHF}}(1^\lambda, 1^t)$ *and* $\mathsf{pp}_{\mathsf{VDS}} \leftarrow\!\!\$\ \mathsf{Setup}_{\mathsf{VDS}}(1^\lambda)$.

$\mathsf{Encode}(\mathsf{ek}, m, \mathsf{id})$**:** *On input the public encoding key* $\mathsf{ek} = (\mathsf{pp}_{\mathsf{MHF}}, \mathsf{pp}_{\mathsf{VDS}})$*, a message* $m \in \{0,1\}^{d \cdot s_p}$*, and an identifier* $\mathsf{id} \in \mathcal{I}$ *(for the message* m*), the deterministic encoding algorithm computes* $v = \mathsf{Eval}_{\mathsf{MHF}}(\mathsf{pp}_{\mathsf{MHF}}, \mathsf{id})$ *and* $r = \mathsf{H}(v, \mathsf{id})$*. In addition, it computes* $r \oplus m = f(X) \in \mathbb{Z}_p[X]$ *(i.e.,* $r \oplus m$ *is interpreted as a random polynomial* $f(X)$ *of degree* $d-1$ *from* $\mathbb{Z}_p[X]$*) and* $(\mathsf{h}, \mathsf{D}, \mathsf{aux}) = \mathsf{GenData}_{\mathsf{VDS}}(\mathsf{pp}_{\mathsf{VDS}}, f, p)$*. Finally, it outputs the encoding* $\mathsf{c} = (\mathsf{D}, \mathsf{aux})$ *and the digest* h.

$\mathsf{Prove}(\mathsf{pk}, \mathsf{chall}, \mathsf{c})$**:** *On input the public proving key* $\mathsf{pk} = \mathsf{pp}_{\mathsf{VDS}}$*, a challenge* $\mathsf{chall} = x \in \mathcal{C}$*, and the encoding* $\mathsf{c} = (\mathsf{D}, \mathsf{aux})$*, the deterministic proving algorithm outputs a proof* $\pi = (y, \pi') = \mathsf{Prove}_{\mathsf{VDS}}(\mathsf{pp}_{\mathsf{VDS}}, x, \mathsf{D}, \mathsf{aux})$.

$\mathsf{Verify}(\mathsf{vk}, \mathsf{h}, \mathsf{chall}, \pi)$**:** *On input the public verification key* $\mathsf{vk} = \mathsf{pp}_{\mathsf{VDS}}$*, a digest* h*, a challenge* $\mathsf{chall} = x \in \mathcal{C}$*, and a proof* $\pi = (y, \pi')$*, the deterministic verification algorithm outputs* $b = \mathsf{Verify}_{\mathsf{VDS}}(\mathsf{pp}_{\mathsf{VDS}}, \mathsf{h}, x, y, \pi')$.

$\mathsf{Decode}(\mathsf{ek}, \mathsf{c}, \mathsf{id})$**:** *On input the public encoding key* $\mathsf{ek} = (\mathsf{pp}_{\mathsf{MHF}}, \mathsf{pp}_{\mathsf{VDS}})$*, an encoding* $\mathsf{c} = (\mathsf{D}, \mathsf{aux})$*, and an identifier* $\mathsf{id} \in \mathcal{I}$*, the deterministic decoding algorithm proceeds as follows:*

- *For every* $i \in [d]$*, compute* $(y_i, \pi_i) = \mathsf{Prove}_{\mathsf{VDS}}(\mathsf{pp}_{\mathsf{VDS}}, i, \mathsf{D}, \mathsf{aux})$ *(recall that* $d < q$ *thus* $d \in \mathbb{Z}_q = \mathcal{C}$*).*[26]
- *Compute a polynomial* $f'(X) \in \mathbb{Z}_p[X]$ *of degree* $d-1$ *by running Lagrange interpolation on the points* $(1, \ldots, d)$ *and the evaluations* $(y_1, \ldots, y_d)$.

Finally, it outputs $m = \mathsf{H}(\mathsf{Eval}_{\mathsf{MHF}}(\mathsf{pp}_{\mathsf{MHF}}, \mathsf{id}), \mathsf{id}) \oplus f'(X)$.

Correctness, completeness, and double-efficiency trivially follow from that of VDS, so we omit the proof.

Theorem 7. *Let* Π_{VDS} *as defined in Construction 2. If* Π_{VDS} *is perfectly correct (Definition 12) and doubly-efficient with* (γ)*-expansion (Definition 15) then* Π *of Construction 2 is perfectly correct, perfectly complete, and doubly-efficient with* (γ)*-expansion.*

As for replication and extraction, we establish the following results (Theorems 8 and 9 and Corollary 6) whose proofs appear in Appendices A.8 to A.10.

[26] We set $d < q$ to guarantee that the challenge space $\mathcal{C}$ contains at least d challenges required for re-computing $f(X)$ using Langrange interpolation.

Theorem 8. *Let Π_{VDS} as defined in Construction 2. If Π_{VDS} is $(\mathsf{negl}(\lambda))$-sound (Definition 4) then Π of Construction 2 is extractable.*

Theorem 9. *Let $s_p(\lambda) = s_p$, $s_q(\lambda) = s_q$, p, q, Π_{MHF}, Π_{VDS}, and H as defined in Construction 2. For every $\lambda \in \mathbb{N}$, $d \in \mathbb{N}$, $u \in \mathbb{N}$, $c \in \mathbb{N}$ such that $c < d(u \cdot s_p - s_q)$, and under the following conditions:*

- *existence of an $(\epsilon_{\mathsf{PRF}})$-secure $\Pi_{\mathsf{PRF}} = (\mathsf{KGen}_{\mathsf{PRF}}, \mathsf{F}_{\mathsf{PRF}})$ scheme with input space $\mathcal{Y}_{\mathsf{MHF}}$ and output space $\{0,1\}^{d \cdot s_p}$ (Definition 8),*

- *$(\epsilon_{\mathsf{MHF}}, \sigma_{\mathsf{MHF}}, n_{\mathsf{MHF}})$-security of Π_{MHF} (Definition 1),*

- *$(\epsilon_{\mathsf{VDS}})$-soundness of Π_{VDS} (Definition 14),*

then Π of Construction 2 satisfies $(\epsilon, \sigma_{\mathsf{MHF}}, n, u)$-replication (Definition 19) in the ROM for $n = \min\{n_{\mathsf{MHF}}, d(u \cdot s_p - s_q) - c\}$, and $\epsilon = \epsilon_{\mathsf{PRF}} + u \cdot q_{\mathsf{H}} \cdot \epsilon_{\mathsf{MHF}} + \frac{u}{|\mathcal{X}_{\mathsf{MHF}}|} + u \cdot \epsilon_{\mathsf{VDS}} + \frac{d-1}{|\mathbb{Z}_q|} + \frac{1}{2^c}$ where q_{H} is the number of queries submitted to the RO H.

Remark 3 (On the Need of Assuming PRFs in Theorem 9)**.** *Surprisingly, Theorem 9 holds under the existence of a secure PRF even if Construction 2 does not involve a PRF at all. The reason behind the need of a PRF is due to the combination of the ROM with experiments (defining the security of a primitive) in which a multi-stage adversary is restricted to sharing a state of bounded size. Examples of such experiments are Definitions 1 and 19 and Theorem 2 in which there is a first adversary that passes to a second adversary a pre-computed state α (i.e., the memory) which is bounded by some parameter $n \geq |\alpha|$. For the sake of concreteness, consider a multi-stage reduction $\mathsf{A}' = (\mathsf{A}'_1, \mathsf{A}'_2)$ that wins against the MHF experiment of Definition 1 by using (in a black-box fashion) a multi-stage algorithm $\mathsf{A} = (\mathsf{A}_1, \mathsf{A}_2, \mathsf{A}_3)$ against the replication experiment of Definition 19 (this is exactly Lemma 11 of the proof of Theorem 9). In order to correctly simulate A's view, A' must answers to all the RO queries consistently. To do so, A must share a state which contains the mapping between the inputs/outputs of the simulated RO. However, A'_1 and A'_2 can only share a state $|\alpha| \leq n$ whereas A may submit any a-priori unknown number of RO queries which may require more than n bits to be stored. This means that A'_2 (the one that receives $|\alpha|$ from A'_1) may fail in replying consistently to the RO queries if A submits a particular query twice (submit the same query to both A'_1 and A'_2). To overcome this problem, we use a PRF that allows A' to compress the inputs/outputs of the RO by sharing a short PRF key k between A'_1 and A'_2. In this way, A'_1 and A'_2 can answer consistently by replying with $\mathsf{F}(\mathsf{k}, x)$ when they receive a RO query x. This is exactly why we need a PRF to prove Theorem 9 (see Lemma 11 of Theorem 9 for more details about the proof).*

By combining Corollaries 1, 4 and 5 with Theorems 8 and 9, we obtain the following corollary (see Appendix A.10 for the formal proof).

Corollary 6. *Under the collision resistant hash function, for every positive constants $\delta_1, \delta_2 > 0$, for every $d \in \mathsf{poly}(\lambda)$, for every $u \in \mathsf{poly}(\lambda)$, there exists a PoRep scheme with message space $\{0,1\}^{d \cdot \lambda^{1+\delta_1}}$, identifier space $\{0,1\}^{\lambda}$, and challenge space $\{0,1\}^{\lambda}$ that satisfies $(\mathsf{negl}(\lambda), \sigma, n, u)$-replication, extraction, and double-efficiency with (γ)-expansion (Definitions 18, 19 and 21), in the parallel ROM where $n = d \cdot u \cdot \lambda^{1+\delta_1} - (d+1) \cdot \lambda$, $\gamma = 2 \cdot d^{\delta_2} \cdot \lambda^{o(1)(1+\delta_1)}$, and $\sigma(\lambda, t, n) = \sigma$ be the upper-bound of (possibly parallel) running time under which the underlying MHF scheme remains secure. Moreover, the (η)-gap of the aforementioned PoRep scheme is defined as $\eta = O\left(\frac{1}{u \cdot \lambda^{\delta_1}}\right)$.*

Acknowledgements

The authors thank Rosario Gennaro for useful discussions and observations about the authenticated data structure. Foteini Baldimtsi is supported by NSF #2143287 and #2247304; Danilo Francati is supported by the Carlsberg Foundation under the Semper Ardens Research Project CF18-112 (BCM); Ioanna Karantaidou is supported by a Protocol Labs Fellowship.

References

[1] Filecoin whitepaper (2017), https://filecoin.io/filecoin.pdf

[2] Effects of chia mining on hard drives (2023), https://platinumdatarecovery.com/blog/chia-crypto-mining-can-kill-your-ssd-or-hard-drive

[3] Abadi, M., Burrows, M., Manasse, M., Wobber, T.: Moderately hard, memory-bound functions. ACM Transactions on Internet Technology (TOIT) **5**(2), 299–327 (2005)

[4] Abusalah, H., Alwen, J., Cohen, B., Khilko, D., Pietrzak, K., Reyzin, L.: Beyond hellman's time-memory trade-offs with applications to proofs of space. In: Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part II 23. pp. 357–379. Springer (2017)

[5] Alwen, J., Blocki, J.: Efficiently computing data-independent memory-hard functions. In: Annual International Cryptology Conference. pp. 241–271. Springer (2016)

[6] Alwen, J., Chen, B., Kamath, C., Kolmogorov, V., Pietrzak, K., Tessaro, S.: On the complexity of scrypt and proofs of space in the parallel random oracle model. In: Advances in Cryptology–EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II 35. pp. 358–387. Springer (2016)

[7] Alwen, J., Serbinenko, V.: High parallel complexity graphs and memory-hard functions. In: Proceedings of the forty-seventh annual ACM symposium on Theory of computing. pp. 595–603 (2015)

[8] Armknecht, F., Barman, L., Bohli, J.M., Karame, G.O.: Mirror: Enabling proofs of data replication and retrievability in the cloud. In: 25th USENIX security symposium (USENIX security 16). pp. 1051–1068 (2016)

[9] Armknecht, F., Bohli, J.M., Karame, G.O., Liu, Z., Reuter, C.A.: Outsourced proofs of retrievability. In: Ahn, G.J., Yung, M., Li, N. (eds.) ACM CCS 2014. pp. 831–843. ACM Press (Nov 2014). https://doi.org/10.1145/2660267.2660310

[10] Arnold, A., Giesbrecht, M., Roche, D.S.: Faster sparse multivariate polynomial interpolation of straight-line programs. Journal of Symbolic Computation **75**, 4–24 (2016)

[11] Ateniese, G., Bonacina, I., Faonio, A., Galesi, N.: Proofs of space: When space is of the essence. In: Security and Cryptography for Networks: 9th International Conference, SCN 2014, Amalfi, Italy, September 3-5, 2014. Proceedings 9. pp. 538–557. Springer (2014)

[12] Ateniese, G., Burns, R., Curtmola, R., Herring, J., Kissner, L., Peterson, Z., Song, D.: Provable data possession at untrusted stores. In: Proceedings of the 14th ACM conference on Computer and communications security. pp. 598–609 (2007)

[13] Ateniese, G., Chen, L., Etemad, M., Tang, Q.: Proof of storage-time: Efficiently checking continuous data availability. NDSS (2020)

[14] Ateniese, G., Chen, L., Francati, D., Papadopoulos, D., Tang, Q.: Verifiable capacity-bound functions: A new primitive from kolmogorov complexity: (revisiting space-based security in the adaptive setting). In: Boldyreva, A., Kolesnikov, V. (eds.) Public-Key Cryptography – PKC 2023. vol. 13941 LNCS, pp. 63–93. Springer Nature Switzerland, Cham (2023). https://doi.org/10.1007/978-3-031-31371-4_3

[15] Ateniese, G., Di Pietro, R., Mancini, L.V., Tsudik, G.: Scalable and efficient provable data possession. In: Proceedings of the 4th international conference on Security and privacy in communication netowrks. pp. 1–10 (2008)

[16] Biryukov, A., Khovratovich, D.: Tradeoff cryptanalysis of memory-hard functions. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 633–657. Springer (2015)

[17] Bluestein, L.: A linear filtering approach to the computation of discrete fourier transform. IEEE Transactions on Audio and Electroacoustics **18**(4), 451–455 (1970). https://doi.org/10.1109/TAU.1970.1162132

[18] Boneh, D., Bonneau, J., Bünz, B., Fisch, B.: Verifiable delay functions. In: Annual international cryptology conference. pp. 757–788. Springer (2018)

[19] Borge, M., Kokoris-Kogias, E., Jovanovic, P., Gasser, L., Gailly, N., Ford, B.: Proof-of-personhood: Redemocratizing permissionless cryptocurrencies. In: 2017 IEEE European Symposium on Security and Privacy Workshops, EuroS&P Workshops 2017, Paris, France, April 26-28, 2017. pp. 23–26. IEEE (2017). https://doi.org/10.1109/EuroSPW.2017.46

[20] Bostan, A., Schost, É.: Polynomial evaluation and interpolation on special sets of points. Journal of Complexity **21**(4), 420–446 (2005)

[21] Cash, D., Küpçü, A., Wichs, D.: Dynamic proofs of retrievability via oblivious ram. Journal of Cryptology **30**, 22–57 (2017)

[22] Cohen, B., Pietrzak, K.: Simple proofs of sequential work. In: Annual international conference on the theory and applications of cryptographic techniques. pp. 451–467. Springer (2018)

[23] Cook, S.A.: An observation on time-storage trade off. In: Proceedings of the fifth annual ACM symposium on Theory of computing. pp. 29–33 (1973)

[24] Curtmola, R., Khan, O., Burns, R., Ateniese, G.: Mr-pdp: Multiple-replica provable data possession. In: 2008 the 28th international conference on distributed computing systems. pp. 411–420. IEEE (2008)

[25] Daian, P., Pass, R., Shi, E.: Snow white: Robustly reconfigurable consensus and applications to provably secure proof of stake. In: Goldberg, I., Moore, T. (eds.) FC 2019. LNCS, vol. 11598, pp. 23–41. Springer, Heidelberg (Feb 2019). https://doi.org/10.1007/978-3-030-32101-7_2

[26] Damgård, I., Ganesh, C., Orlandi, C.: Proofs of replicated storage without timing assumptions. In: Boldyreva, A., Micciancio, D. (eds.) CRYPTO 2019, Part I. LNCS, vol. 11692, pp. 355–380. Springer, Heidelberg (Aug 2019). https://doi.org/10.1007/978-3-030-26948-7_13

[27] David, B., Gazi, P., Kiayias, A., Russell, A.: Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In: Nielsen, J.B., Rijmen, V. (eds.) EUROCRYPT 2018, Part II. LNCS, vol. 10821, pp. 66–98. Springer, Heidelberg (Apr / May 2018). https://doi.org/10.1007/978-3-319-78375-8_3

[28] Dodis, Y., Vadhan, S., Wichs, D.: Proofs of retrievability via hardness amplification. In: Theory of Cryptography: 6th Theory of Cryptography Conference, TCC 2009, San Francisco, CA, USA, March 15-17, 2009. Proceedings 6. pp. 109–127. Springer (2009)

[29] Dwork, C., Naor, M., Wee, H.: Pebbling and proofs of work. In: Advances in Cryptology–CRYPTO 2005: 25th Annual International Cryptology Conference, Santa Barbara, California, USA, August 14-18, 2005. Proceedings 25. pp. 37–54. Springer (2005)

[30] Dziembowski, S., Faust, S., Kolmogorov, V., Pietrzak, K.: Proofs of space. In: Annual Cryptology Conference. pp. 585–605. Springer (2015)

[31] Fisch, B.: Tight proofs of space and replication. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019, Part II. LNCS, vol. 11477, pp. 324–348. Springer, Heidelberg (May 2019). https://doi.org/10.1007/978-3-030-17656-3_12

[32] Fisch, B., Bonneau, J., Benet, J., Greco, N.: Proofs of replication using depth robust graphs. Blockchain Protocol Analysis and Security Engineering **2018** (2018)

[33] Garay, J.A., Kiayias, A., Leonardos, N.: The bitcoin backbone protocol: Analysis and applications. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015, Part II. LNCS, vol. 9057, pp. 281–310. Springer, Heidelberg (Apr 2015). https://doi.org/10.1007/978-3-662-46803-6_10

[34] Hanling, M., Anthoine, G., Dumas, J.G., Maignan, A., Pernet, C., Roche, D.S.: Poster: Proofs of retrievability with low server storage. In: Cavallaro, L., Kinder, J., Wang, X., Katz, J. (eds.) ACM CCS 2019. pp. 2601–2603. ACM Press (Nov 2019). https://doi.org/10.1145/3319535.3363266

[35] Van der Hoeven, J.: The truncated fourier transform and applications. In: Proceedings of the 2004 international symposium on Symbolic and algebraic computation. pp. 290–296 (2004)

[36] Hopcroft, J., Paul, W., Valiant, L.: On time versus space and related problems. In: 16th Annual Symposium on Foundations of Computer Science (sfcs 1975). pp. 57–64. IEEE (1975)

[37] Huang, X., Pan, V.Y.: Fast rectangular matrix multiplication and applications. Journal of complexity **14**(2), 257–299 (1998)

[38] Juels, A., Kaliski Jr., B.S.: Pors: proofs of retrievability for large files. In: Ning, P., De Capitani di Vimercati, S., Syverson, P.F. (eds.) ACM CCS 2007. pp. 584–597. ACM Press (Oct 2007). https://doi.org/10.1145/1315245.1315317

[39] Kate, A., Zaverucha, G.M., Goldberg, I.: Constant-size commitments to polynomials and their applications. In: Abe, M. (ed.) ASIACRYPT 2010. LNCS, vol. 6477, pp. 177–194. Springer, Heidelberg (Dec 2010). https://doi.org/10.1007/978-3-642-17373-8_11

[40] Kedlaya, K.S., Umans, C.: Fast polynomial factorization and modular composition. SIAM Journal on Computing **40**(6), 1767–1802 (2011)

[41] Lee, J.: Dory: Efficient, transparent arguments for generalised inner products and polynomial commitments. In: Nissim, K., Waters, B. (eds.) TCC 2021, Part II. LNCS, vol. 13043, pp. 1–34. Springer, Heidelberg (Nov 2021). https://doi.org/10.1007/978-3-030-90453-1_1

[42] Lengauer, T., Tarjan, R.E.: Asymptotically tight bounds on time-space trade-offs in a pebble game. Journal of the ACM (JACM) **29**(4), 1087–1130 (1982)

[43] Lin, W.K., Mook, E., Wichs, D.: Doubly efficient private information retrieval and fully homomorphic ram computation from ring lwe. In: Proceedings of the 55th Annual ACM Symposium on Theory of Computing. pp. 595–608 (2023)

[44] Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (May 2009), http://www.bitcoin.org/bitcoin.pdf

[45] Nüsken, M., Ziegler, M.: Fast multipoint evaluation of bivariate polynomials. In: European Symposium on Algorithms. pp. 544–555. Springer (2004)

[46] Papamanthou, C., Shi, E., Tamassia, R.: Signatures of correct computation. In: Sahai, A. (ed.) TCC 2013. LNCS, vol. 7785, pp. 222–242. Springer, Heidelberg (Mar 2013). https://doi.org/10.1007/978-3-642-36594-2_13

[47] Paul, W.J., Tarjan, R.E.: Time-space trade-offs in a pebble game. Acta Informatica **10**, 111–115 (1978)

[48] Pietrzak, K.: Proofs of catalytic space. Cryptology ePrint Archive (2018)

[49] Rabaninejad, R., Abdolmaleki, B., Malavolta, G., Michalas, A., Nabizadeh, A.: storna: Stateless transparent proofs of storage-time. Cryptology ePrint Archive (2023)

[50] Ren, L., Devadas, S.: Proof of space from stacked expanders. In: Theory of Cryptography: 14th International Conference, TCC 2016-B, Beijing, China, October 31-November 3, 2016, Proceedings, Part I 14. pp. 262–285. Springer (2016)

[51] Shacham, H., Waters, B.: Compact proofs of retrievability. Journal of cryptology **26**(3), 442–483 (2013)

[52] Shi, E., Stefanov, E., Papamanthou, C.: Practical dynamic proofs of retrievability. In: Sadeghi, A.R., Gligor, V.D., Yung, M. (eds.) ACM CCS 2013. pp. 325–336. ACM Press (Nov 2013). https://doi.org/10.1145/2508859.2516669

[53] Van Der Hoeven, J., Lecerf, G.: On the bit-complexity of sparse polynomial and series multiplication. Journal of Symbolic Computation **50**, 227–254 (2013)

[54] Van Der Hoeven, J., Schost, É.: Multi-point evaluation in higher dimensions. Applicable Algebra in Engineering, Communication and Computing **24**(1), 37–52 (2013)

A Supporting Proofs

A.1 Proof of Theorem 2

Fix $u \in \mathbb{N}$ and $c \leq d(u \cdot s_p - s_q)$. Assume there exists a PPT adversary $\mathsf{A} = (\mathsf{A}_1, \mathsf{A}_2)$ such that

$$\mathbb{P}\left[\begin{array}{c} \mathsf{A}_2(1^\lambda, x, \alpha) = (f_1(x), \ldots, f_u(x)) \wedge \\ |\alpha| \leq d(u \cdot s_p - s_q) - c \end{array} : \begin{array}{c} (f_1(X), \ldots, f_u(X)) \leftarrow\!\!\$\, \mathbf{F}^u_{d-1,p}, \\ \alpha \leftarrow\!\!\$\, \mathsf{A}_1(1^\lambda, f_1, \ldots, f_u), \\ x \leftarrow\!\!\$\, \mathbb{Z}_q \end{array}\right] \tag{2}$$
$$> \frac{d-1}{|\mathbb{Z}_q|} + \frac{1}{2^c} = \epsilon.$$

Fix $(f_1(X), \ldots, f_u(X)) \leftarrow\!\!\$\, \mathbf{F}^u_{d-1,p}$, $\alpha \leftarrow\!\!\$\, \mathsf{A}_1(1^\lambda, f_1, \ldots, f_u)$, and the random coins $r_2 \in \{0,1\}^*$ of A_2. Let $\mathcal{X}_{f_1,\ldots,f_u,\alpha,r_2}$ be the set of points for which the adversary A_2, on input $(1^\lambda, \alpha)$ and random coins r_2, is able to correctly compute $(f_1(x), \ldots, f_u(x))$, i.e.,

$$\mathcal{X}_{f_1,\ldots,f_u,\alpha,r_2} \overset{\text{def}}{=} \{x : x \in \mathbb{Z}_q \text{ such that } (f_1(x), \ldots, f_u(x)) = \mathsf{A}_2(1^\lambda, x, \alpha; r_2)\}.$$

It is easy to see that $\frac{|\mathcal{X}_{f_1,\ldots,f_u,\alpha,r_2}|}{|\mathbb{Z}_q|} > \epsilon$; otherwise, Equation (2) does not hold. Hence, we have that $|\mathcal{X}_{f_1,\ldots,f_u,\alpha,r_2}| > \epsilon \cdot |\mathbb{Z}_q| = d - 1 + \frac{|\mathbb{Z}_q|}{2^c} > d - 1$. This means that $\mathcal{X}_{f_1,\ldots,f_u,\alpha,r_2} \subseteq \mathbb{Z}_q$ contains at least d distinct points with probability ϵ where the probability is taken over choices of $(f_1(X), \ldots, f_u(X)) \leftarrow\!\!\$\, \mathbf{F}^u_{d-1,p}$, $\alpha \leftarrow\!\!\$\, \mathsf{A}_1(1^\lambda, f_1, \ldots, f_u)$, and random coins $r_2 \leftarrow\!\!\$\, \{0,1\}^*$.

We leverage this observation to build an adversary $\mathsf{A}' = (\mathsf{A}'_1, \mathsf{A}'_2)$ that contradicts the $(c, \frac{1}{2^c})$-incompressibility of $\mathbf{U}_{u \cdot d \cdot s_p}$ (Theorem 1). Without loss of generality, we assume that both A'_1 and A'_2 have hardcoded $r_2 \leftarrow\!\!\$\, \{0,1\}^*$. $\mathsf{A}' = (\mathsf{A}'_1, \mathsf{A}'_2)$ are defined as follows:

$\mathsf{A}'_1(1^\lambda, a)$: On input the security parameter 1^λ and string $a = (a_0, \ldots, a_{u \cdot d - 1}) \in \{0,1\}^{u \cdot d \cdot s_p}$, A'_1 proceeds as follows:

1. Compute $\alpha \leftarrow\!\!\$\, \mathsf{A}_1(1^\lambda, f_1, \ldots, f_u)$ where $f_j(X) = \sum_{i=0}^{d-1} a_{j \cdot d + i} \cdot X^i \in \mathbb{Z}_p[X]$ for every $j \in \{0\} \cup [u-1]$.
2. Compute (in time $|\mathbb{Z}_q|$) the set $\mathcal{X}_{f_1,\ldots,f_u,\alpha,r_2}$. Note that this is possible since A'_1 is unbounded and has r_2 hardcoded.
3. Output $\alpha' = (\alpha, x_1, \ldots, x_d)$ where $x_1, \ldots, x_d \in \mathcal{X}_{f_1,\ldots,f_u,\alpha,r_2}$.

$\mathsf{A}'_2(1^\lambda, \alpha')$: On input the security parameter 1^λ and $\alpha' = (\alpha, x_1, \ldots, x_d)$, A'_2 proceeds as follows:

1. For every $i \in [d]$, compute $(y_{i,1}, \ldots, y_{i,u}) \leftarrow\!\!\$\, \mathsf{A}_2(1^\lambda, x_i, \alpha; r_2)$.
2. For every $j \in \{0\} \cup [u-1]$, compute $(a_{j \cdot d}, \ldots, a_{j \cdot d + d - 1})$ using Langrange interpolation over the evaluations $y_{1,j}, \ldots, y_{d,j}$ and the points $x_1, \ldots, x_d$ (recall that p is of size $(s_p + 1)$. This guarantees that each a_i of size s_p can be correctly reconstructed).
3. Output $a = (a_0, \ldots a_{u \cdot d - 1})$.

First, it is easy to see that A'_1 outputs a string α' of the correct size since

$$|\alpha'| = |\alpha| + |x_1| + \ldots + |x_d| \leq d(u \cdot s_p - s_q) - c + d \cdot s_q = u \cdot d \cdot s_p - c.$$

Second, we know that $\mathcal{X}_{f_1,\ldots,f_u,\alpha,r_2}$ contains d points with probability ϵ and, by definition of $\mathcal{X}_{f_1,\ldots,f_u,\alpha,r_2}$, the evaluations $(y_{1,j}, \ldots, y_{d,j})_{j \in [u]}$ are correct, i.e., $y_{i,j} = f_j(x_i)$ for every $i \in [d]$ and for every $j \in [u]$. Since $f_1(X), \ldots, f_u(X)$ are all univariate polynomials of degree $d-1$, the Lagrange interpolation correctly reconstructs the corresponding coefficients of each polynomial. In turn, this implies that A'_2 outputs the correct string $a = (a_0, \ldots, a_{u \cdot d - 1})$. Hence, $\mathsf{A}' = (\mathsf{A}'_1, \mathsf{A}'_2)$ retains the same advantage of $\mathsf{A} = (\mathsf{A}_1, \mathsf{A}_2)$ which is greater than $\frac{1}{2^c}$ (independently from the choices of d and q. See Equation (2)). This concludes the proof.

A.2 Proof of Corollary 4

The corollary follows by setting $c = c(\lambda) = \lambda$, $s_p = s_p(\lambda) = \lambda^{1+\delta}$, $s_q = s_q(\lambda) = \lambda$, and observing that $\frac{d-1}{|\mathbb{Z}_q|} \leq \frac{\mathsf{poly}(\lambda)}{2^\lambda} \in O(\frac{1}{2^\lambda})$, $\frac{1}{2^c} = \frac{1}{2^\lambda}$, and

$$\begin{aligned}|\alpha| \leq d(u \cdot s_p - s_q) - c = d \cdot u \cdot s_p - (d \cdot s_q + c) = d \cdot u \cdot \lambda^{1+\delta} - (d \cdot \lambda + \lambda)\\ = d \cdot u \cdot \lambda^{1+\delta} - (d+1) \cdot \lambda.\end{aligned}$$

A.3 Proof of Theorem 3

This proof relies on the oracle abstraction defined in Definition 9. This is because, the localized version $\mathsf{Local.T}^{[x'],[\mathsf{map}]}(y)$ of a deterministic RAM algorithm $\mathsf{T}^{[x]}(y)$ is essentially the execution of $\mathsf{T}^{[\cdot]}(y)$ where the (oracle) RAM accessible input (denoted as $[\cdot]$) is simulated by $\mathsf{Local.T}^{[x'],[\mathsf{map}]}(y)$ using its oracle access to both x' and map, and intercepting the all (read, i) read commands issued by $\mathsf{T}^{[\cdot]}(y)$ (as defined in Definition 9).

Formally, consider the following construction.

Construction 3. *Let* T *be a deterministic RAM algorithm. We build the localized version* $\mathsf{Local.T}$ *of* T *as follows.*

$\mathsf{Local.T}^{[x'],[\mathsf{map}]}(y)$**:** *On input an arbitrary binary string* $y \in \{0,1\}^*$, *a RAM accessible input* $x' = (x'_1, \ldots, x'_{k'})$, *and a RAM accessible memory mapping* $\mathsf{map} = (i'_1, \ldots, i'_{k'})$, *the localized deterministc RAM algorithm* $\mathsf{Local.T}$ *executes* $\mathsf{T}^{[\cdot]}(y)$ *(the notation* $[\cdot]$ *indicates that* $\mathsf{Local.T}$ *will simulate the access to the read-only RAM input) and, until* T *stops or it outputs a value* v*, it proceeds as follows:*

1. *When* T *submits a read command* (read, i_j), $\mathsf{Local.T}$ *leverages its read-only RAM access to* map *to execute a binary search over* map *to identify the index* c *which is the location of* i_j *into* map*. If such an index* c *does not exist (i.e.,* $i_j \notin \mathsf{map}$*),* $\mathsf{Local.T}$ *outputs* $\perp$ *and terminates.*
2. $\mathsf{Local.T}$ *sends the read command* (read, c) *to* $[x']$ *and receives* x'_c *as a result. Then,* $\mathsf{Local.T}$ *returns* x'_c *to* T *as the answer of the read command* (read, i_j)*.*
3. $\mathsf{Local.T}$ *waits for the next read command of* T *and re-executes Items 1 to 3.*

Finally, $\mathsf{Local.T}$ *outputs the same value* v *output by* $\mathsf{T}^{[\cdot]}(y)$*.*

We now demonstrate that $\mathsf{Local.T}$ of Construction 3 satisfies perfect correctness and invalid mapping. Lastly, we demonstrate the running time of $\mathsf{Local.T}$.

Lemma 1. *The localized version* $\mathsf{Local.T}$ *(Construction 3) of the deterministic RAM algorithm* T *is perfectly correct (Definition 11).*

Proof. Fix the read-only accessible input $x = (x_1, \ldots, x_n)$ and an arbitrary binary input of T. Also, consider $\mathcal{I}_{x,y}$, k', map, and $x' = (x'_1, \ldots, x'_{k'})$ as defined in the perfect correctness property of Definition 11. Then, we have the following conditions:

1. map is ordered, i.e., $\forall j \in [k']$ then $i'_j < i'_{j+1}$,
2. $\forall i_j \in \mathcal{I}_{x,y}$ then $i_j \in \mathsf{map}$,
3. $\forall i'_j \in \mathsf{map}$ if $i'_j \in \mathcal{I}_{x,y}$ then $x'_j = x_{i'_j}$.

From Item 1, we conclude that $\mathsf{Local.T}$ of Construction 3 can correctly execute the binary search on map. Moreover, by definition, T will submit only read commands (read, i_j) such that $i_j \in \mathcal{I}_{x,y}$. In turn, by combining the above with Item 2, we conclude that $\mathsf{Local.T}$ will never return $\perp$ since i_j will be in map. Let c be the index of i_j (of the read command $(read, i_j)$) indentified by $\mathsf{Local.T}$ after running the binary search on map. By leveraging, Item 3 we have that $x'_c = x_{i_j}$ which is the correct value expected by T. Hence, it must be that $\mathsf{T}^{[x]}(y) = v = \mathsf{Local.T}^{[x'],[\mathsf{map}]}(y)$. This concludes the proof. □

Lemma 2. *The localized version* $\mathsf{Local.T}$ *(Construction 3) of the deterministic RAM algorithm* T *satisfies the invalid mapping property (Definition 11).*

Proof. Fix the read-only accessible input $x = (x_1, \ldots, x_n)$ and an arbitrary binary input of T. Also, consider $\mathcal{I}_{x,y}$, k', map, and $x' = (x'_1, \ldots, x'_{k'})$ as defined in the invalid mapping property of Definition 11. Then, we have the following conditions:

1. map is ordered, i.e., $\forall j \in [k']$ then $i'_j < i'_{j+1}$,
2. $\exists i_j \in \mathcal{I}_{x,y}$ such that $i_j \notin \mathsf{map}$,
3. $\forall i'_j \in \mathsf{map}$ then $x'_j = x_{i'_j}$.

As usual, Item 1 implies that $\mathsf{Local.T}$ of Construction 3 can correctly execute the binary search on map. Moreover, by definition, T will submits only read commands (read, i_j) such that $i_j \in \mathcal{I}_{x,y}$. Now, fix a generic m-th read command (read, i_j) submitted by T. We can identify the following two cases:

- If $i_j \in \mathsf{map}$ then $\mathsf{Local.T}$ will return x'_c to T as the answer of the read command (read, i_j) where c is the index of i_j indentified by $\mathsf{Local.T}$ (after running the binary search on map). By leveraging, Item 3, we have $x'_c = x_{i_j}$ which is precisely the value that T expects to see. Hence, the computation of T and $\mathsf{Local.T}$ can continue as expected.
- If $i_j \notin \mathsf{map}$ (note that $i_j \in \mathcal{I}_{x,y}$ also holds since, by definition, T will only submit read commands (read, i_j) such that $i_j \in \mathcal{I}_{x,y}$) then the binary search, executed by $\mathsf{Local.T}$, will not find an index c. In turn, $\mathsf{Local.T}$ will output $\perp$ and terminate. Observe that this case must happen (i.e., there exists an m'-th read command query (read, i_j) such that $i_j \notin \mathsf{map}$) since we know that $\exists i_j \in \mathcal{I}_{x,y}$ such that $i_j \notin \mathsf{map}$ (Item 2).

By combining the above observations, we conclude that $\mathsf{Local.T}^{[x']}(y)$ will output $\perp$. This concludes the proof. □

Lemma 3. *For every read-only RAM accessible input x, arbitrary binary input y, read-only RAM accessible input x', and read-only RAM accessible memory mapping* map*, the running time of* $\mathsf{Local.T}^{[x'],[\mathsf{map}]}(y)$ *(Construction 3) is at most $t \cdot \log(|\mathsf{map}|)$ where t is the running time of* $\mathsf{T}^{[x]}(y)$*. Running times of both* T *and* $\mathsf{Local.T}$ *are measured in the RAM model of computation.*

Proof. Assume that the running time of $\mathsf{T}^{[x]}(y)$ is t in the RAM model of computation. This implies that $\mathsf{T}^{[x]}(y)$ submits at most t read command queries to its read-only RAM accessible input x. Without loss of generality, assume $\mathsf{T}^{[x]}(y)$ only submits read command queries and does not perform any internal computation (this is fine to assume since $\mathsf{Local.T}^{[x'],[\mathsf{map}]}(y)$ will produce an overhead only when a read command query is submitted by $\mathsf{T}^{[x]}(y)$). For every read command query submitted by $\mathsf{T}^{[x]}(y)$, $\mathsf{Local.T}^{[x'],[\mathsf{map}]}(y)$ executes a binary search over map which requires (worst-case) time $\log(|\mathsf{map}|)$ in the RAM model of computation. Hence, the overall running time of $\mathsf{Local.T}^{[x'],[\mathsf{map}]}(y)$ will be at most $t \cdot \log(|\mathsf{map}|)$. □

Theorem 3 follows by combining Lemmas 1 to 3.

A.4 Proof of Theorem 4

First, regarding perfect correctness of Π, the value y, output by $\mathsf{Eval}(\mathsf{pp}, x, \mathsf{D}, \mathsf{aux})$, is computed as $y = \mathsf{Eval}_{\mathsf{DS}}(x, \mathsf{D})$ where $\mathsf{D} = \mathsf{GenData}_{\mathsf{DS}}(f, p)$. As a consequence, perfect correctness of Π simply follows from the perfect correctness of Π_{DS}.

Second, regarding perfect completeness of Π, fix an honestly generated output (y, π) produced by $\mathsf{Eval}(\mathsf{pp}, x, \mathsf{D}, \mathsf{aux})$ where $\pi = (\mathcal{I}_{x,\mathsf{D}_{x,\mathsf{D}}}, \{\pi_j\}_{j \in \mathcal{I}_{x,\mathsf{D}}})$ as defined in Construction 1 (recall that $\mathcal{I}_{x,\mathsf{D}}$ is the ordered set of indexes read from the data structure D during the RAM computation $\mathsf{Eval}_{\mathsf{DS}}(x, \mathsf{D})$. See Item 1). On verification, $\mathsf{Verify}(\mathsf{pp}, \mathsf{h}, x, \pi)$ proceeds as follows:

1. It checks $|\mathcal{I}_{x,\mathsf{D}}| \leq \ell$ and $\mathcal{I}_{x,\mathsf{D}} = \{i_1, \ldots, i_k\}$ is an ordered set. If $\mathsf{Eval}(\mathsf{pp}, x, \mathsf{D}, \mathsf{aux})$ is honestly executed then these conditions hold by definition (observe that D is composed of ℓ blocks so $\mathcal{I}_{x,\mathsf{D}}$ cannot be bigger than ℓ).

2. It computes $\mathsf{Verify}_{\mathsf{VC}}(\mathsf{pp}_{\mathsf{VC}}, \mathsf{c}, \mathsf{D}_j, j, \pi_j) = b_j$ for $j \in \mathcal{I}_{x,\mathsf{D}}$. Since Π_{VC} is perfectly correct, we have that $b_j = 1$ for every $j \in \mathcal{I}_{x,\mathsf{D}}$.

3. It executes $\mathsf{Local.Eval}_{\mathsf{DS}}(x, (\mathsf{D}_{i_1}, \ldots, \mathsf{D}_{i_k}), \mathsf{map}) = y'$ where $\mathcal{I}_{x,\mathsf{D}} = \{i_1, \ldots, i_k\}$ and $\mathsf{map} = (i_1, \ldots, i_k)$. By definition, we have that $\mathcal{I}_{x,\mathsf{D}} = \{i_1, \ldots, i_k\}$ is the ordered set of indexes read from x during the computation $\mathsf{Eval}_{\mathsf{DS}}(\mathsf{pp}, x, \mathsf{D})$. Hence, we conclude that $\mathsf{Local.Eval}_{\mathsf{DS}}(x, (\mathsf{D}_{i_1}, \ldots, \mathsf{D}_{i_k}), \mathsf{map}) = y' = y = \mathsf{Eval}_{\mathsf{DS}}(x, \mathsf{D})$ by leveraging the perfect correctness of the localized computation $\mathsf{Local.Eval}_{\mathsf{DS}}$ of $\mathsf{Eval}_{\mathsf{DS}}$ (see Definition 11).

Observe that the final output of $\mathsf{Verify}(\mathsf{pp}, \mathsf{h}, x, \pi)$ is 1 if $y' = y$ and $b_j = 1$ for every $j \in \mathcal{I}_{x,\mathsf{D}}$. Hence, by combining the two above observations, we conclude that Π is perfectly complete.

A.5 Proof of Theorem 5

Suppose that Π is not $(\ell \cdot \epsilon_{\mathsf{VC}})$-sound, i,e, there exists a valid PPT adversary A such that

$$\mathbb{P}\left[\begin{array}{c} \mathsf{Verify}(\mathsf{pp}, \mathsf{h}, \widetilde{x}, \widetilde{y}, \widetilde{\pi}) = 1 \wedge \\ \widetilde{y} \neq f(\widetilde{x}) \end{array} : \begin{array}{c} \mathsf{pp} \leftarrow\!\!\$\ \mathsf{Setup}(1^\lambda) \\ (\widetilde{x}, \widetilde{y}, \widetilde{\pi}, f, p) \leftarrow\!\!\$\ \mathsf{A}(1^\lambda, \mathsf{pp}) \\ (\mathsf{D}, \mathsf{h}, \mathsf{aux}) = \mathsf{GenData}(\mathsf{pp}, f, p) \end{array}\right] > \ell \cdot \epsilon_{\mathsf{VC}}. \qquad (3)$$

Fix the output $(\widetilde{x}, \widetilde{y}, \widetilde{\pi})$ of A where $\widetilde{\pi} = (\widetilde{\mathcal{I}}, \{\widetilde{\mathsf{D}}_1, \ldots \widetilde{\mathsf{D}}_n\}, \{\widetilde{\pi}_1, \ldots, \widetilde{\pi}_n\})$ and $\widetilde{\mathcal{I}} = \{\widetilde{i_1}, \ldots, \widetilde{i_n}\}$. Through the proof, we assume the following:

1. $n \leq \ell \in \mathbb{N}$ and $\widetilde{\mathcal{I}}$ is an ordered set; otherwise, the verification $\mathsf{Verify}(\mathsf{pp}, \mathsf{h}, \widetilde{x}, \widetilde{y}, \widetilde{\pi})$ would output 0, contradicting Equation (3) (see Construction 1).

2. $p \in \mathbb{N}$ is a prime and $f(X) \in \mathbb{Z}_p[X]$. This is because A is valid with respect to Definition 14.

Lemma 4. *For every $j \in [n]$, the following probability holds:*

$$\mathbb{P}\left[\begin{array}{c} \mathsf{Verify}_{\mathsf{VC}}(\mathsf{pp}_{\mathsf{VC}}, \mathsf{c}, \widetilde{\mathsf{D}}_j, \widetilde{i_j}, \widetilde{\pi}_j) = 1 \wedge \\ \mathsf{D}_{\widetilde{i_j}} \neq \widetilde{\mathsf{D}}_j \end{array} : \begin{array}{c} \mathsf{pp} \leftarrow\!\!\$\ \mathsf{Setup}(1^\lambda) \\ (\widetilde{x}, \widetilde{y}, \widetilde{\pi}, f, p) \leftarrow\!\!\$\ \mathsf{A}(1^\lambda, \mathsf{pp}) \\ (\mathsf{D}, \mathsf{h}, \mathsf{aux}) = \mathsf{GenData}(\mathsf{pp}, f, p) \end{array}\right] \leq \epsilon_{\mathsf{VC}},$$

where $\widetilde{\pi} = (\widetilde{\mathcal{I}}, \{\widetilde{\mathsf{D}}_1, \ldots \widetilde{\mathsf{D}}_n\}, \{\widetilde{\pi}_1, \ldots, \widetilde{\pi}_n\})$, $\widetilde{\mathcal{I}} = \{\widetilde{i_1}, \ldots, \widetilde{i_n}\}$, $\mathsf{h} = \mathsf{c}$, $\mathsf{pp} = \mathsf{pp}_{\mathsf{VC}}$, and $\mathsf{D} = (\mathsf{D}_1, \ldots, \mathsf{D}_\ell)$.

Proof. Suppose that the above probability does not hold. We build a PPT adversary A_{VC} that breaks the (ϵ_{VC})-positional binding of Π_{VC}. A_{VC} proceeds as follows:

1. Receive $\mathsf{pp}_{\mathsf{VC}}$ from the challenger.
2. Send $\mathsf{pp}_{\mathsf{VC}}$ to A.
3. Receive $(\widetilde{x}, \widetilde{y}, \widetilde{\pi}, f, p)$ from A where $\widetilde{\pi} = (\widetilde{\mathcal{I}}, \{\widetilde{\mathsf{D}}_1, \dots \widetilde{\mathsf{D}}_n\}, \{\widetilde{\pi}_1, \dots, \widetilde{\pi}_n\})$ and $\widetilde{\mathcal{I}} = \{\widetilde{i_1}, \dots, \widetilde{i_n}\}$.
4. Compute $(\mathsf{D}, \mathsf{h}, \mathsf{aux}) = \mathsf{GenData}(\mathsf{pp}_{\mathsf{VC}}, f, p)$ where $\mathsf{D} = (\mathsf{D}_1, \dots, \mathsf{D}_\ell)$.
5. Output $(\mathsf{h}, \mathsf{D}_{\widetilde{i_j}}, \widetilde{\mathsf{D}}_j, \widetilde{i_j}, \pi_{\widetilde{i_j}}, \widetilde{\pi}_j)$ where $\pi_{\widetilde{i_j}} = \mathsf{Open}_{\mathsf{VC}}(\mathsf{pp}_{\mathsf{VC}}, \mathsf{D}_{\widetilde{i_j}}, \widetilde{i_j}, \mathsf{aux})$.

It is easy to see that A_{VC} correctly simulates the view of A. Conditioned to a correct simulation, we observe the following:

1. $\mathsf{Verify}_{\mathsf{VC}}(\mathsf{pp}_{\mathsf{VC}}, \mathsf{h}, \mathsf{D}_{\widetilde{i_j}}, \widetilde{i_j}, \pi_{\widetilde{i_j}}) = 1$ where $\mathsf{h} = \mathsf{c} = \mathsf{Commit}_{\mathsf{VC}}(\mathsf{pp}_{\mathsf{VC}}, (\mathsf{D}_1, \dots, \mathsf{D}_\ell))$. This follows from the perfect correctness of Π_{VC}.
2. $\mathsf{Verify}_{\mathsf{VC}}(\mathsf{pp}_{\mathsf{VC}}, \mathsf{h}, \widetilde{\mathsf{D}}_j, \widetilde{i_j}, \widetilde{\pi}_j) = 1$ and $\mathsf{D}_{\widetilde{i_j}} \neq \widetilde{\mathsf{D}}_j$ with probability at least ϵ_{VC}. This because, by definition, A has an advantage ϵ_{VC} in outputting $(\widetilde{x}, \widetilde{y}, \widetilde{\pi}, f, p)$ such that $\mathsf{Verify}_{\mathsf{VC}}(\mathsf{pp}_{\mathsf{VC}}, \mathsf{c}, \widetilde{\mathsf{D}}_j, \widetilde{i_j}, \widetilde{\pi}_j) = 1$ and $\mathsf{D}_{\widetilde{i_j}} \neq \widetilde{\mathsf{D}}_j$.

By combining the above observations, we conclude that A_{VC} breaks positional binding with probability greater than ϵ_{VC}. This concludes the proof. □

Lemma 5. *The following probability holds:*

$$\mathbb{P}\left[\ \mathsf{Local.Eval}_{\mathsf{DS}}(\widetilde{x}, (\mathsf{D}'_1, \dots, \mathsf{D}'_n), \mathsf{map}) = \bot \ : \begin{array}{c} \mathsf{pp} \leftarrow\!\!\$\ \mathsf{Setup}(1^\lambda), \\ (\widetilde{x}, \widetilde{y}, \widetilde{\pi}, f, p) \leftarrow\!\!\$\ \mathsf{A}(1^\lambda, \mathsf{pp}), \\ (\mathsf{D}, \mathsf{h}, \mathsf{aux}) = \mathsf{GenData}(\mathsf{pp}, f, p), \\ \exists i_j \in \mathcal{I}_{\mathsf{D},\widetilde{x}} \ s.t.\ i_j \notin \mathsf{map} \end{array} \right] = 1,$$

where $\mathcal{I}_{\mathsf{D},\widetilde{x}} = \{i_1, \dots, i_k\}$ *is the set of ordered indexes read from* x *during the (honestly executed) deterministic RAM computation* $\mathsf{Eval}_{\mathsf{DS}}(\widetilde{x}, \mathsf{D})$*,* $\widetilde{\pi} = (\widetilde{\mathcal{I}}, \{\widetilde{\mathsf{D}}_1, \dots, \widetilde{\mathsf{D}}_n\}, \{\widetilde{\pi}_1, \dots, \widetilde{\pi}_n\})$*,* $\widetilde{\mathcal{I}} = \{\widetilde{i_1}, \dots, \widetilde{i_n}\}$*,* $\mathsf{h} = \mathsf{c}$*,* $\mathsf{pp} = \mathsf{pp}_{\mathsf{VC}}$*,* $\mathsf{map} = (\widetilde{i_1}, \dots, \widetilde{i_n})$*, and* $(\mathsf{D}'_1, \dots, \mathsf{D}'_n) = (\mathsf{D}_{\widetilde{i_1}}, \dots, \mathsf{D}_{\widetilde{i_n}})$.

Proof. Observe that the following conditions hold:

- map is ordered since it is computed from $\widetilde{\mathcal{I}}$ which in turn is an ordered set; otherwise, $\mathsf{Verify}(\mathsf{pp}, \mathsf{h}, \widetilde{x}, \widetilde{y}, \widetilde{\pi})$ would output 0, contradicting Equation (3).
- By definition, Lemma 5 assumes that $\exists i_j \in \mathcal{I}_{\mathsf{D},\widetilde{x}}$ such that $i_j \notin \mathsf{map}$.
- By definition, Lemma 5 assumes that $\forall \widetilde{i_j} \in \mathsf{map}$ we have $\mathsf{D}'_j = \mathsf{D}_{\widetilde{i_j}}$.

By leveraging the above conditions, the Lemma 5 follows by using the invalid mapping property (Definition 11) of the localized deterministic RAM algorithm $\mathsf{Local.Eval}_{\mathsf{DS}}$ of $\mathsf{Eval}_{\mathsf{DS}}$. □

Lemma 6. *The following probability holds:*

$$\mathbb{P}\left[\mathsf{Local.Eval}_{\mathsf{DS}}(\widetilde{x}, (\mathsf{D}'_1, \dots, \mathsf{D}'_n), \mathsf{map}) = f(\widetilde{x}) : \begin{array}{c} \mathsf{pp} \leftarrow\!\!\$\ \mathsf{Setup}(1^\lambda) \\ (\widetilde{x}, \widetilde{y}, \widetilde{\pi}, f, p) \leftarrow\!\!\$\ \mathsf{A}(1^\lambda, \mathsf{pp}) \\ (\mathsf{D}, \mathsf{h}, \mathsf{aux}) = \mathsf{GenData}(\mathsf{pp}, f, p) \\ \nexists i_j \in \mathcal{I}_{\mathsf{D},\widetilde{x}} \ s.t.\ i_j \notin \mathsf{map} \end{array} \right] = 1,$$

where $\mathcal{I}_{\mathsf{D},\widetilde{x}} = \{i_1, \dots, i_k\}$ *is the set of ordered indexes read from* x *during the (honestly executed) deterministic RAM computation* $\mathsf{Eval}_{\mathsf{DS}}(\widetilde{x}, \mathsf{D})$*,* $\widetilde{\pi} = (\widetilde{\mathcal{I}}, \{\widetilde{\mathsf{D}}_1, \dots, \widetilde{\mathsf{D}}_n\}, \{\widetilde{\pi}_1, \dots, \widetilde{\pi}_n\})$*,* $\widetilde{\mathcal{I}} = \{\widetilde{i_1}, \dots, \widetilde{i_n}\}$*,* $\mathsf{h} = \mathsf{c}$*,* $\mathsf{pp} = \mathsf{pp}_{\mathsf{VC}}$*,* $\mathsf{map} = (\widetilde{i_1}, \dots, \widetilde{i_n})$*, and* $(\mathsf{D}'_1, \dots, \mathsf{D}'_n) = (\mathsf{D}_{\widetilde{i_1}}, \dots, \mathsf{D}_{\widetilde{i_n}})$.

Proof. Observe that the following conditions hold:

- map is ordered since it is computed from $\widetilde{\mathcal{I}}$ which in turn is an ordered set; otherwise, $\mathsf{Verify}(\mathsf{pp}, \mathsf{h}, \widetilde{x}, \widetilde{y}, \widetilde{\pi})$ would output 0, contradicting Equation (3).
- By definition, Lemma 6 assumes that $\nexists i_j \in \mathcal{I}_{\mathsf{D},\widetilde{x}}$ such that $i_j \notin \mathsf{map}$ (or equivalently $\forall i_j \in \mathcal{I}_{\mathsf{D},\widetilde{x}}$ we have $i_j \in \mathsf{map}$).
- By definition, Lemma 6 assumes that $\forall \widetilde{i_j} \in \mathsf{map}$ we have $\mathsf{D}'_j = \mathsf{D}_{\widetilde{i_j}}$. Moreover, in combination with the above condition, we can also conclude that $\forall \widetilde{i_j} \in \mathsf{map}$ we have if $\widetilde{i_j} \in \mathcal{I}_{\mathsf{D},\widetilde{x}}$ then $\mathsf{D}'_j = \mathsf{D}_{\widetilde{i_j}}$.

By leveraging the above conditions, Lemma 6 follows by using the perfect correctness (Definition 11) of the localized deterministic RAM algorithm $\mathsf{Local.Eval}_{\mathsf{DS}}$ of $\mathsf{Eval}_{\mathsf{DS}}$. □

To conclude the proof of Theorem 5, we need to observe the following:

1. By leveraging ℓ times Lemma 4, we conclude that $\exists j \in [n]$ such that $\mathsf{D}_{\widetilde{i_j}} \neq \widetilde{\mathsf{D}}_j$ with probability probability at most $\ell \cdot \epsilon_{\mathsf{VC}}$. By combining this observation with Equation (3), we obtain that $(\mathsf{D}_{\widetilde{i_1}}, \ldots, \mathsf{D}_{\widetilde{i_n}}) = (\widetilde{\mathsf{D}}_1, \ldots, \widetilde{\mathsf{D}}_n)$.
2. Conditioned to $(\mathsf{D}_{\widetilde{i_1}}, \ldots, \mathsf{D}_{\widetilde{i_n}}) = (\widetilde{\mathsf{D}}_1, \ldots, \widetilde{\mathsf{D}}_n)$, if $\exists i_j \in \mathcal{I}_{\mathsf{D},\widetilde{x}}$ such that $i_j \notin \widetilde{\mathcal{I}}$, then also map (computed by the verification algorithm Verify) is such that $\exists i_j \in \mathcal{I}_{\mathsf{D},\widetilde{x}}$ where $\widetilde{i_j} \notin \mathsf{map}$ (this because map is computed from $\widetilde{\mathcal{I}}$). In turn, by leveraging Lemma 5 we conclude that the localized deterministic RAM computation $\mathsf{Local.Eval}_{\mathsf{DS}}$ (of $\mathsf{Eval}_{\mathsf{DS}}$) will output $y' = \bot$. By definition, this can not happen since $y' = \bot \neq f(\widetilde{x})$ and, in turn, this would require $\mathsf{Verify}(\mathsf{pp}, \mathsf{h}, \widetilde{x}, \widetilde{y}, \widetilde{\pi})$ to output 0, contradicting Equation (3). Hence, it must be that $\nexists i_j \in \mathcal{I}_{\mathsf{D},\widetilde{x}}$ such that $i_j \notin \widetilde{\mathcal{I}}$.
3. On the other hand, conditioned to $(\mathsf{D}_{\widetilde{i_1}}, \ldots, \mathsf{D}_{\widetilde{i_n}}) = (\widetilde{\mathsf{D}}_1, \ldots, \widetilde{\mathsf{D}}_n)$, if $\nexists i_j \in \mathcal{I}_{\mathsf{D},\widetilde{x}}$ such that $i_j \notin \widetilde{\mathcal{I}}$, then also map (computed by the verification algorithm Verify) is such that $\nexists i_j \in \mathcal{I}_{\mathsf{D},\widetilde{x}}$ where $\widetilde{i_j} \notin \mathsf{map}$ (this is because map is computed from $\widetilde{\mathcal{I}}$). Similarly to the previous case, by leveraging Lemma 5 we conclude that the localized deterministic RAM algorithm $\mathsf{Local.Eval}_{\mathsf{DS}}$ (of $\mathsf{Eval}_{\mathsf{DS}}$) will output $\mathsf{Local.Eval}_{\mathsf{DS}}(\widetilde{x}, (\widetilde{\mathsf{D}}_1, \ldots, \widetilde{\mathsf{D}}_n), \mathsf{map}) = y' = f(\widetilde{x}) = \mathsf{Eval}_{\mathsf{DS}}(\widetilde{x}, \mathsf{D})$. Finally, $\mathsf{Verify}(\mathsf{pp}, \mathsf{h}, \widetilde{x}, \widetilde{y}, \widetilde{\pi}) = 1$ (as defined in Equation (3)) implies that $y = y'$. Hence, we conclude that $y = f(\widetilde{x})$ which contradicts Equation (3) (which says $y \neq f(\widetilde{x})$).

Theorem 5 follows from the combination of Items 1 to 3 above (which in turn they leverage Lemmas 4 to 6).

A.6 Proof of Corollary 5

By leveraging Corollary 3 we have that the size $|\mathsf{D}|$ of D (output by $\mathsf{GenData}$) is $|f(X)| \cdot \gamma_{\mathsf{DS}}$, where $\gamma_{\mathsf{DS}} = (d+1)^{\delta} \log^{o(1)}(p)$. In addition, Corollary 2 implies that $|\mathsf{aux}| = \ell \cdot z = |f(X)| \cdot \gamma_{\mathsf{DS}}$ since $\ell = \ell(d)$ corresponds to the number of blocks of D when each block (of D) is of size z (see Construction 1). Finally, by leveraging Theorem 6 we conclude that

$$\gamma = \gamma_{\mathsf{DS}} + \frac{|\mathsf{aux}|}{|f(X)|} = 2 \cdot \gamma_{\mathsf{DS}} = 2(d+1)^{\delta} \log^{o(1)}(p).$$

Regarding the size of digest, we can observe that $\mathsf{h} = \mathsf{c}$ and $|\mathsf{c}| = \lambda$ (see Corollary 2) where c is generated by Π_{VC}.

A.7 Proof of Theorem 6

We prove each efficiency property individually.

(γ)-expansion: This property is straigthforward. Since $|\mathsf{D}| = |f(X)| \cdot \gamma_{\mathsf{DS}} + |\mathsf{aux}|$, we obtain that

$$\gamma = \frac{|\mathsf{D}| + |\mathsf{aux}|}{|f(X)|} = \gamma_{\mathsf{DS}} + \frac{|\mathsf{aux}|}{|f(X)|},$$

where γ_{DS} is the expansion factor of the underlying Π_{DS} and aux is the auxiliary information output by Π_{VC}.

Efficient of evaluation and verification: As for evaluation, we can observe that Eval proceeds as follows:

1. It executes $\mathsf{Eval}_{\mathsf{DS}}(x, \mathsf{D})$ and computes the ordered set of indexes $\mathcal{I}_{x,\mathsf{D}}$. Since Π_{DS} is efficient (see Definition 6), we obtain that both $\mathsf{Eval}_{\mathsf{DS}}(x, \mathsf{D})$ and the computation of $\mathcal{I}_{x,\mathsf{D}}$ requires time $\mathsf{poly}(\log(d), \log(p))$. Observe that the running time for computing $\mathcal{I}_{x,\mathsf{D}}$ holds since $\mathsf{Eval}_{\mathsf{DS}}(x, \mathsf{D})$ runs in time $\mathsf{poly}(\log(d), \log(p))$ which, in turn, is an upper-bound on the number of indexes read. In addition, ordering $\mathcal{I}_{x,\mathsf{D}}$ has a logarithmic multiplicate overhead which is absorbed by the asymptotic notation $\mathsf{poly}(\log(d), \log(p))$.
2. It computes $|\mathcal{I}_{x,\mathsf{D}}| \in \mathsf{poly}(\log(d), \log(p))$ openings π_j by executing $|\mathcal{I}_{x,\mathsf{D}}|$ times the opening algorithm $\mathsf{Open}_{\mathsf{VC}}$. Since Π_{VC} is efficient (Definition 2), computing these $|\mathcal{I}_{x,\mathsf{D}}|$ openings requires time $\mathsf{poly}(\lambda, \log(d), \log(p))$.[27]

On the other hand, as for verification, Verify proceeds as follows:

1. It checks that $\mathcal{I}_{x,\mathsf{D}} \leq \ell$ and that $\mathcal{I}_{x,\mathsf{D}}$ is ordered. Since $\mathcal{I}_{x,\mathsf{D}}$ is upper-bounded by the (worst-case) running time of $\mathsf{Eval}_{\mathsf{DS}}$, we conclude that these steps require (worst-case) time $\mathsf{poly}(\log(d), \log(p))$.
2. It executes $|\mathcal{I}_{x,\mathsf{D}}|$ times the verification algorithm $\mathsf{Verify}_{\mathsf{VC}}$. As usual Π_{VC} is efficient. Thus, the overall (worst-case) running time of this step is $\mathsf{poly}(\lambda, \log(d), \log(p))$.
3. Finally, it executes the localized computation $\mathsf{Local}.\mathsf{Eval}_{\mathsf{DS}}(x, (\mathsf{D}_{i_1}, \ldots, , \mathsf{D}_{i_k}), \mathsf{map})$ where $|\mathsf{map}| = |\mathcal{I}_{x,\mathsf{D}}| \in \mathsf{poly}(\log(d), \log(p))$ since $\mathsf{Eval}_{\mathsf{DS}}$ runs in time $\mathsf{poly}(\log(d), \log(p))$. By combining Theorem 3 with the above observation, we have that the localized RAM algorithm $\mathsf{Local}.\mathsf{Eval}_{\mathsf{DS}}$ has (worst-case) running time $\mathsf{poly}(\log(d), \log(p))$.

To conclude, the (worst-case) running times of both Eval and Verify are $\mathsf{poly}(\lambda, \log(d), \log(p))$ in the RAM model of computation.

This implies that Construction 1 is doubly-efficient.

A.8 Proof of Theorem 8

Consider the following extractor $\mathsf{Ext}^{\mathsf{A}_3(\cdot)}$ (with oracle access to A_3 as defined in experiment $\mathbf{G}^{\mathsf{extract}}_{\mathsf{A},\Pi,\mathsf{Ext}}(\lambda, t, u)$ of Definition 21):

$\mathsf{Ext}^{\mathsf{A}_3(\cdot)}(1^\lambda, \mathsf{ek}, (\mathsf{h}_i, \mathsf{id}_i)_{i\in[u]})$**:** On input the security parameter 1^λ, the public encoding key $\mathsf{ek} = (\mathsf{pp}_{\mathsf{MHF}}, \mathsf{pp}_{\mathsf{VDS}})$, the public verification key $\mathsf{vk} = \mathsf{pp}_{\mathsf{VDS}}$, u pairs of digests and identifiers

[27] Recall that $\ell = \ell(d)$ is a polynomial function that depends on d. Thus, $\log(\ell) = O(\log(d))$.

$(\mathsf{h}_i, \mathsf{id}_i)_{i\in[u]}$ (each corresponding to a message m_i that Ext needs to extract), and oracle access to the adversary $\mathsf{A}_3(\cdot)$,[28] the extractor proceeds as follows:

1. Initialize $y_{i,j} = \bot$ for every $i \in [u]$ and $j \in [d]$.
2. For every $j \in [d]$ and for every $q \in [\mathsf{trials}]$ where $\mathsf{trials}(\lambda) \in \omega(\log(\lambda))$ (note that trials depends on the security parameter λ):
   (a) Send $\mathsf{chall}_{j,q} \leftarrow\!\!\$\, \mathcal{C}$ to A_3 and receive the answer $(\pi_{i,j,q})_{i\in[u]}$ where $\pi_{i,j,q} = (y_{i,j,q}, \pi'_{i,j,q})$ (recall that A_3 will reply with u proofs, one for each message).
   (b) If $\mathsf{Verify}(\mathsf{vk}, \mathsf{h}_i, \mathsf{chall}_{j,q}, \pi_{i,j,q}) = 1$ for every $i \in [u]$, set $y_{i,j} = y_{i,j,q}$ and $x_j = \mathsf{chall}_{j,q}$ for every $i \in [u]$.
3. If $\exists i \in [u]$ and $j \in [d]$ such that $y_{i,j}$, the extractor aborts and outputs $\bot$.
4. For every $i \in [u]$, use Lagrange interpolation over the points $(x_j)_{j\in[d]}$ and the evaluations $(y_{i,j})_{j\in[d]}$ to compute $f'_i(X) \in \mathbb{Z}_p[X]$.
5. Finally, return $(m'_1, \ldots, m'_u)$ where $m_i = f'_i(X) \oplus \mathsf{H}(\mathsf{Eval}(\mathsf{pp}_{\mathsf{MHF}}, \mathsf{id}_i), \mathsf{id}_i)$ for $i \in [u]$.

It is easy to see that the extractor runs in polynomial-time since $u \in \mathsf{poly}(\lambda)$, $d \in \mathbb{Z}_p[X]$, and $\mathsf{trials} = \omega(\log(\lambda))$.

We now prove the following lemmas with respect to $\mathsf{A}_3(\cdot)$ and the aforementioned extractor $\mathsf{Ext}^{\mathsf{A}_3(\cdot)}$.

Lemma 7. *For every PPT adversary* $\mathsf{A} = (\mathsf{A}_1, \mathsf{A}_2, \mathsf{A}_3)$ *we have*

$$\mathbb{P}\Big[\mathbf{G}^{\mathsf{replicate}}_{\mathsf{A},\Pi}(\lambda, t, n, u) = 1\Big] \geq \frac{1}{\mathsf{poly}(\lambda)} \implies$$
$$\mathbb{P}\Big[\mathsf{Ext}^{\mathsf{A}_3(\cdot)}(1^\lambda, \mathsf{ek}, \mathsf{vk}, (\mathsf{h}_i, \mathsf{id}_i)_{i\in[u]}) \neq \bot\Big] \geq 1 - \mathsf{negl}(\lambda).$$

Proof. Let $\nu(\lambda)$ be a non-negligible probability such that

$$\mathbb{P}\Big[\mathbf{G}^{\mathsf{replicate}}_{\mathsf{A},\Pi}(\lambda, t, n, u) = 1\Big] \geq \nu(\lambda).$$

Recall that the above implies that the probability that A_3 returns at least one proof (among the u) that does not verify is at most $1 - \nu(\lambda)$ which, in turn, is non-negligible (this is because $\nu(\lambda)$ is non-negligible). The probability that $\mathsf{Ext}^{\mathsf{A}_3(\cdot)}(1^\lambda, \mathsf{ek}, \mathsf{vk}, (\mathsf{h}_i, \mathsf{id}_i)_{i\in[u]}) = \bot$ is bounded as follows:

$$\mathbb{P}\Big[\mathsf{Ext}^{\mathsf{A}_3(\cdot)}(1^\lambda, \mathsf{ek}, \mathsf{vk}, (\mathsf{h}_i, \mathsf{id}_i)_{i\in[u]}) = \bot\Big] = \mathbb{P}[\exists i \in [u], \exists j \in [d], y_{i,j} = \bot] =$$
$$\mathbb{P}[\exists i \in [u], \exists j \in [d], \forall q \in [\mathsf{trials}], \mathsf{Verify}(\mathsf{vk}, \mathsf{h}_i, \mathsf{chall}_{j,q}, \pi_{i,j,q}) = 0].$$

It is easy to see that the last probability is at most $(1 - \nu(\lambda))^{\mathsf{trials}} = (1 - \nu(\lambda))^{\omega(\log(\lambda))}$ which is negligible. In turn, this implies that $\mathsf{Ext}^{\mathsf{A}_3(\cdot)}(1^\lambda, \mathsf{ek}, \mathsf{vk}, (\mathsf{h}_i, \mathsf{id}_i)_{i\in[u]}) \neq \bot$ with probability at least $1 - (1 - \nu(\lambda))^{\omega(\log(\lambda))} = 1 - \mathsf{negl}(\lambda)$. This concludes the proof. □

Lemma 8. $\mathbb{P}[\forall j_1 \in [d], \forall j_2 \in [d] \setminus \{j_1\}, x_{j_1} \neq x_{j_2}] \geq 1 - \mathsf{negl}(\lambda)$.

Proof. The lemma follows by simply observing that:

- for every $j \in [d]$, x_j corresponds to a randomly sampled challenge $\mathsf{chall}_{j,q} \leftarrow\!\!\$\, \mathcal{C}$ (for some $q \in [\mathsf{trials}]$), and

[28] Recall that the extractor can only submit challenges to the adversary A_3 as defined in Definition 21

- $\mathcal{C} = \mathbb{Z}_q$ and q is a (s_q)-bits prime were $s_q \in \omega(\log(\lambda))$ (see Item 1 of Construction 2), then $|\mathcal{C}| = |\mathbb{Z}_q| = 2^{\omega(\log(\lambda))}$.

Since Ext runs in polynomial-time (i.e., it samples at most $\mathsf{poly}(\lambda)$ challenges from $\mathcal{C}$) and $|\mathcal{C}|$ is of size super-polynomial in the security parameter λ, then the probability of sampling the same point twice (i.e., $\exists j_1 \in [d], \exists j_2 \in [d] \setminus \{j_1\}$ such that $x_{j_1} = x_{j_2}$) is negligible. This concludes the proof. □

Lemma 9. *If Π_{VDS} is $(\mathsf{negl}(\lambda))$-sound then $\forall i \in [u], \forall j \in [d], \forall q \in \mathsf{trials}$ we have*

$$\mathbb{P}[\mathsf{Verify}(\mathsf{vk}, \mathsf{h}_i, \mathsf{chall}_{j,q}, \pi_{i,j,q}) = 1 \wedge y_{i,j,q} \neq f_i(\mathsf{chall}_{j,q})] \leq \mathsf{negl}(\lambda), \tag{4}$$

where $f_i(X) = m_i \oplus \mathsf{H}(\mathsf{Eval}(\mathsf{pp}_{\mathsf{MHF}}, \mathsf{id}_i), \mathsf{id}_i)$ for every $i \in [u]$.

Proof. Observe that $\mathsf{Verify}(\mathsf{vk}, \mathsf{h}_i, \mathsf{chall}_{j,q}, \pi_{i,j,q}) = 1$ only if $\mathsf{Verify}_{\mathsf{VDS}}(\mathsf{pp}_{\mathsf{VDS}}, \mathsf{h}_i, \mathsf{chall}_{j,q}, \pi'_{i,j,q}) = 1$ (see Construction 2). Thus, if there exists $i \in [u], j \in [d], q \in \mathsf{trials}$ for which Equation (4) does not hold (i.e., the probability is non-negligible) then we would contradict the $(\mathsf{negl}(\lambda))$-soundness of Π_{VDS} (recall that $u, d \in \mathsf{poly}(\lambda)$ and $\mathsf{trials} \in \omega(\log(\lambda))$). The proof is standard so we omit it (the proof is similar to that of Lemma 14 of the proof of Theorem 9). □

The lemmas above imply the following:

- By leveraging Lemma 7, the extractor does not abort (i.e., it outputs $\perp$) with overwhelming probability. This also implies that for every $i \in [u]$, for every $j \in [u]$, we have $y_{i,j} \neq \perp$ and $\mathsf{Verify}(\mathsf{vk}, \mathsf{h}_i, \mathsf{chall}_{j,q}, \pi_{i,j,q}) = 1$ (for the corresponding $q \in \mathsf{trials}$) where $\mathsf{chall}_{j,q} = x_j$.
- Conditioned to the above, we have that $y_{i,j} = f_i(x_j)$ for every $i \in [u]$, for every $j \in [d]$ with overwhelming probability (this follows by leveraging Lemma 9).
- Lastly, by leveraging Lemma 8 we have that points $(x_1, \dots x_d)$ are all different with overwhelming probability (Lemma 8).

Hence, the above observations imply that the Lagrange interpolation (executed by Ext) computes the correct polynomial $f'_i(X) = f_i(X) = m_i \oplus \mathsf{H}(\mathsf{Eval}_{\mathsf{MHF}}(\mathsf{pp}_{\mathsf{MHF}}, \mathsf{id}_i), \mathsf{id}_i)$. Thus, Ext correctly extracts the messages $(m'_1, \dots, m'_u) = (m_1, \dots, m_u)$ with overwhelming probability. This concludes the proof.

A.9 Proof of Theorem 9

Consider the following hybrid experiments (defined in the random oracle model):

$\mathbf{H}^0(\lambda, t, n, u)$: This hybrid experiment is identical to the adaptive experiment $\mathbf{G}^{\mathsf{replicate}}_{\mathsf{A},\Pi}(\lambda, t, n, u)$ of PoRep.

$\mathbf{H}^1(\lambda, t, n, u)$: Identical to $\mathbf{H}^0(\lambda, t, n, u)$ except that the challenger simulates the majority of the random oracle queries using the PRF scheme Π_{PRF}. More in detail, the challenger computes $\mathsf{k} \leftarrow\$ \mathsf{KGen}_{\mathsf{PRF}}(1^\lambda)$ and samples the u challenge random identifier $(\mathsf{id}_1, \dots, \mathsf{id}_u) \leftarrow\$ \mathcal{I}^u$ in advance. Then, on input a query $(v', \mathsf{id}') \in \mathcal{Y}_{\mathsf{MHF}} \times \mathcal{X}_{\mathsf{MHF}}$ for the random oracle $\mathsf{H}(\cdot)$, the challenger proceeds as follows:

  - If $v' \neq v_i = \mathsf{Eval}_{\mathsf{MHF}}(\mathsf{pp}_{\mathsf{MHF}}, \mathsf{id}_i)$ or $\mathsf{id}' \neq \mathsf{id}_i$ for every $i \in [u]$, the challenger returns $\mathsf{F}_{\mathsf{PRF}}(\mathsf{k}, (v', \mathsf{id}'))$ (instead of a random value).
  - Otherwise, if there exists $i \in [u]$ such that $v' = v_i = \mathsf{Eval}_{\mathsf{MHF}}(\mathsf{pp}_{\mathsf{MHF}}, \mathsf{id}_i)$ and $\mathsf{id}' = \mathsf{id}_i$, the challenger returns a random value $r' \leftarrow\$ \{0, 1\}^{d \cdot s_p}$.

$\mathbf{H}^2(\lambda, t, n, u)$**:** Identical to $\mathbf{H}^1(\lambda, t, n, u)$ except that the challenger aborts if the adversary A_3 (i.e., the third adversary) submits the random oracle query $(v', \mathsf{id}') \in \mathcal{Y}_{\mathsf{MHF}} \times \mathcal{X}_{\mathsf{MHF}}$ such that $v' = v_i = \mathsf{Eval}(\mathsf{pp}_{\mathsf{MHF}}, \mathsf{id}_i)$ for some $i \in [u]$, i.e., the challenger aborts if $(\mathsf{Eval}(\mathsf{pp}_{\mathsf{MHF}}, \mathsf{id}_i), \mathsf{id}') \in \mathcal{Q}_{\mathsf{A}_3,\mathsf{H}}$ (for some id' and $i \in [u]$) where $\mathcal{Q}_{\mathsf{A}_3,\mathsf{H}}$ is the set of random oracle queries submitted by A_3 and id_i is the i-th challenge random identifier which is sampled at random from $\mathcal{I}$. Observe that if A_1 and A_2 submit $(\mathsf{Eval}(\mathsf{pp}_{\mathsf{MHF}}, \mathsf{id}_i), \mathsf{id}')$ (for some id' and $i \in [u]$) to the random oracle, the challenger does not abort.

$\mathbf{H}^3(\lambda, t, n, u)$**:** Identical to $\mathbf{H}^2(\lambda, t, n, u)$ except that the challenger aborts if the adversary A_1 (i.e., the first adversary) submits the random oracle query (v', id') such that $\mathsf{id}' = \mathsf{id}_i$ for some $i \in [u]$, i.e., the challenger aborts if $(v', \mathsf{id}_i) \in \mathcal{Q}_{\mathsf{A}_1,\mathsf{H}}$ (for some v' and $i \in [u]$) where $\mathcal{Q}_{\mathsf{A}_1,\mathsf{H}}$ is the set of random oracle queries submitted by A_1 and id_i is the i-th challenge random identifier which is sampled at random from $\mathcal{I}$ in advance.

$\mathbf{H}^4(\lambda, t, n, u)$**:** Identical to $\mathbf{H}^5(\lambda, t, n, u)$ except that the challenger changes its strategy for computing $f_1(X), \ldots, f_u(X)$ and answering to the random oracle query (v', id') such that $v' = v_i = \mathsf{Eval}_{\mathsf{MHF}}(\mathsf{pp}_{\mathsf{MHF}}, \mathsf{id}_i)$ and $\mathsf{id}' = \mathsf{id}_i$ (for some $i \in [u]$), where id_i is the i-th challenge random identifier which is sampled in advance. More formally, let $\mathbf{F}^u_{d-1,p}$ (where p is a $(s_p + 1)$-bits prime) be a distribution over univariate polynomials of degree $d - 1$ from $\mathbb{Z}_p[X]$ that samples u polynomials $f_1(X), \ldots, f_u(X) \in \mathbb{Z}_p[X]$ as follows:

1. Sample $(a_0, \ldots, a_{u \cdot d - 1}) \leftarrow\!\!\$\, \mathbf{U}_{u \cdot d \cdot s_p}$.
2. Return u univariate polynomials $f_1(X), \ldots, f_u(X)$ such that $f_j(X) = \sum_{i=0}^{d-1} a_{j \cdot d + i} \cdot X^i$ for every $j \in [u - 1] \cup \{0\}$ (i.e., each binary string a_i is interpreted as an element of $\mathbb{Z}_p$).

The challenger proceeds as follows:

- The challenger samples $(\mathsf{id}_1, \ldots, \mathsf{id}_u) \leftarrow\!\!\$\, \mathcal{I}^u$ and $(f_1(X), \ldots, f_u(X)) \leftarrow\!\!\$\, \mathbf{F}^u_{d-1,p}$.
- The challenger starts the experiment $\mathbf{H}^4(\lambda, t, n, u)$.
- When $\mathsf{A}_1(1^\lambda, \mathsf{ek}, \mathsf{pk}, \mathsf{vk})$ outputs $(m_1, \ldots, m_u, \beta)$, the challenger runs $v_i = \mathsf{Eval}_{\mathsf{MHF}}($ $\mathsf{pp}_{\mathsf{MHF}}, \mathsf{id}_i)$ and sets $\mathsf{H}(v_i, \mathsf{id}_i) = r_i = f_i(X) \oplus m_i$ for every $i \in [u]$.
- Then, the challenger continues the execution of $\mathbf{H}^4(\lambda, t, n)$ which is identical to $\mathbf{H}^3(\lambda, t, n, u)$ except that the challenger will use $f_1(X), \ldots, f_u(X)$ and $\mathsf{H}(v_1, \mathsf{id}_1), \ldots,$ $\mathsf{H}(v_u, \mathsf{id}_u)$ computes as described defined above.

$\mathbf{H}^5(\lambda, t, n, u)$**:** Identical to $\mathbf{H}^4(\lambda, t, n, u)$ except that the outcome of the experiment $\mathbf{H}^5(\lambda, t, n, u)$ is set to 0 if there exists $i \in [u]$ such that $y_i \neq f_i(\mathsf{chall})$ where $\pi_i = (y_i, \pi'_i)$ is the i-th proof output by the adversary A_3.

Lemma 10. *If* Π_{PRF} *is* $(\epsilon_{\mathsf{PRF}})$*-secure then*

$$\mathbf{H}^0(\lambda, t, n, u) \approx_{\epsilon_{\mathsf{PRF}}} \mathbf{H}^1(\lambda, t, n, u).$$

Proof. Assume there exists a PPT distinguisher $\mathsf{A} = (\mathsf{A}_1, \mathsf{A}_2, \mathsf{A}_3)$ that distinguishes between $\mathbf{H}^0(\lambda, t, n, u)$ and $\mathbf{H}^1(\lambda, t, n, u)$ with advantage greater than ϵ_{PRF}. Then, we build $\mathsf{A}_{\mathsf{PRF}}$ that breaks the $(\epsilon_{\mathsf{PRF}})$-security of Π_{PRF}. $\mathsf{A}_{\mathsf{PRF}}$ is defind as follows:

1. Compute $(\mathsf{ek}, \mathsf{pk}, \mathsf{vk}) \leftarrow\!\!\$\, \mathsf{Setup}(1^\lambda, 1^t)$ where $\mathsf{ek} = (\mathsf{pp}_{\mathsf{MHF}}, \mathsf{pp}_{\mathsf{VDS}})$, $\mathsf{pk} = \mathsf{vk} = \mathsf{pp}_{\mathsf{VDS}}$.
2. For every $i \in [u]$, sample $\mathsf{id}_i \leftarrow\!\!\$\, \mathcal{I}$, $r_i \leftarrow\!\!\$\, \{0, 1\}^{d \cdot s_p}$, and compute $v_i = \mathsf{Eval}_{\mathsf{MHF}}(\mathsf{pp}_{\mathsf{MHF}}, \mathsf{id}_i)$.

3. Execute $\mathsf{A}_1(1^\lambda, \mathsf{ek}, \mathsf{pk}, \mathsf{vk})$ and answer the incoming random oracle queries as follows:

   (a) On input the random oracle query $(v', \mathsf{id}') \in \mathcal{Y}_{\mathsf{MHF}} \times \mathcal{X}_{\mathsf{MHF}}$ such that $v_i \neq v'$ or $\mathsf{id}_i \neq \mathsf{id}'$ for every $i \in [u]$, $\mathsf{A}_{\mathsf{PRF}}$ forwards (v', id') to its oracle and returns the answer.

   (b) On the other hand, on input the random oracle query $(v', \mathsf{id}') \in \mathcal{Y}_{\mathsf{MHF}} \times \mathcal{X}_{\mathsf{MHF}}$ such that $v_i = v'$ and $\mathsf{id}_i = \mathsf{id}'$ for some $i \in [u]$, $\mathsf{A}_{\mathsf{PRF}}$ returns r_i.

4. Eventually, $\mathsf{A}_1(1^\lambda, \mathsf{ek}, \mathsf{pk}, \mathsf{vk})$ outputs $(m_1, \dots, m_u, \mathsf{state})$.

5. For every $i \in [u]$, compute h_i and $\mathsf{c}_i = (\mathsf{D}_i, \mathsf{aux}_i)$ where $f_i(X) = r_i \oplus m_i$ and $(\mathsf{D}_i, \mathsf{h}_i, \mathsf{aux}_i) = \mathsf{GenData}_{\mathsf{VDS}}(\mathsf{pp}_{\mathsf{VDS}}, f_i, p)$.

6. Execute $\mathsf{A}_2(1^\lambda, (\mathsf{id}_i, \mathsf{h}_i, \mathsf{c}_i)_{i\in[u]}, \mathsf{state})$ and answer the incoming random oracle queries as in Item 3.

7. Eventually, $\mathsf{A}_2(1^\lambda, (\mathsf{id}_i, \mathsf{h}_i, \mathsf{c}_i)_{i\in[u]}, \mathsf{state})$ outputs α.

8. Sample $\mathsf{chall} \leftarrow\!\!\$\, \mathcal{C}$.

9. Execute $\mathsf{A}_3(1^\lambda, \mathsf{ek}, \mathsf{pk}, \mathsf{vk}, (\mathsf{id}_i)_i \in [u], \mathsf{chall}, \alpha)$, and answer the incoming queries as in Item 3.

10. Finally, output whatever is returned by $\mathsf{A}_3(1^\lambda, \mathsf{ek}, \mathsf{pk}, \mathsf{vk}, (\mathsf{id}_i)_{i\in[u]}, \mathsf{chall}, \alpha)$.

It is easy to see that, in the random oracle model, $\mathsf{A}_{\mathsf{PRF}}$ correctly simulates the view of $\mathsf{A} = (\mathsf{A}_1, \mathsf{A}_2, \mathsf{A}_3)$. Hence, $\mathsf{A}_{\mathsf{PRF}}$ retains the same advantage ϵ_{PRF} of $\mathsf{A} = (\mathsf{A}_1, \mathsf{A}_2, \mathsf{A}_3)$. This concludes the proof. □

Lemma 11. *If Π_{MHF} is $(\epsilon_{\mathsf{MHF}}, \sigma_{\mathsf{MHF}}, n_{\mathsf{MHF}})$-secure (Definition 1) then for every valid PPT distinguisher $\mathsf{A} = (\mathsf{A}_1, \mathsf{A}_2, \mathsf{A}_3)$ we have that*

$$\mathbf{H}^1(\lambda, t, n, u) \approx_{u \cdot q_{\mathsf{H}} \cdot \epsilon_{\mathsf{MHF}}} \mathbf{H}^2(\lambda, t, n, u).$$

A distinguisher $\mathsf{A} = (\mathsf{A}_1, \mathsf{A}_2, \mathsf{A}_3)$ is called valid if $|\alpha| \leq n$ and A_3 runs in parallel time σ_{MHF} with $\mathsf{poly}(t)$ processors (as defined in Theorem 9).

Proof. Assume there exists a valid PPT distinguisher $\mathsf{A} = (\mathsf{A}_1, \mathsf{A}_2, \mathsf{A}_3)$ that distinguishes between $\mathbf{H}^1(\lambda, t, n, u)$ and $\mathbf{H}^2(\lambda, t, n, u)$ with advantage greater than $u \cdot q_{\mathsf{H}} \cdot \epsilon_{\mathsf{MHF}}$.

Let $\mathbf{E}$ be the event that there exists $i \in [u]$, $(v', \mathsf{id}') \in \mathcal{Q}_{\mathsf{A}_3,\mathsf{H}}$ such that $\mathsf{Eval}_{\mathsf{MHF}}(\mathsf{pp}_{\mathsf{MHF}}, \mathsf{id}_i) = v_i = v'$ where $\mathcal{Q}_{\mathsf{A}_3,\mathsf{H}}$ is the set of random oracle queries submitted by A_3, i.e., $\mathbf{E}$ corresponds to the event that the challenger aborts. Also, let $\mathbf{E}^*$ be the event that $\mathsf{A} = (\mathsf{A}_1, \mathsf{A}_2, \mathsf{A}_3)$ outputs $b = 1$. Then, the advantage of A can be rewritten as follows:

$$\big|\, \mathbb{P}[\mathbf{E}^* | \mathbf{E}] \cdot \mathbb{P}[\mathbf{E}] - \mathbb{P}[\mathbf{E}^* | \neg\mathbf{E}] \cdot \mathbb{P}[\neg\mathbf{E}] \big| > u \cdot q_{\mathsf{H}} \cdot \epsilon_{\mathsf{MHF}}.$$

We observe that $\mathbb{P}[\mathbf{E}^* | \neg\mathbf{E}] = 0$ when $\neg\mathbf{E}$ occurs (this is because, conditioned to $\neg\mathbf{E}$, the hybrids $\mathbf{H}^1(\lambda, t, n, u)$ and $\mathbf{H}^2_d(\lambda, t, n, u)$ are identical). Hence, it must be that

$$\mathbb{P}[\mathbf{E}^* | \mathbf{E}] \cdot \mathbb{P}[\mathbf{E}] \geq \mathbb{P}[\mathbf{E}] > u \cdot q_{\mathsf{H}} \cdot \epsilon_{\mathsf{MHF}}. \tag{5}$$

By leveraging the fact that $\mathbb{P}[\mathbf{E}] > u \cdot q_{\mathsf{H}} \cdot \epsilon_{\mathsf{MHF}}$, we build an adversary $\mathsf{A}_{\mathsf{MHF}} = (\mathsf{A}_{\mathsf{MHF},1}, \mathsf{A}_{\mathsf{MHF},2})$ that breaks the $(\epsilon_{\mathsf{MHF}}, \sigma_{\mathsf{MHF}}, n_{\mathsf{MHF}})$-security of Π_{MHF}. Recall that the validity of $\mathsf{A} = (\mathsf{A}_1, \mathsf{A}_2, \mathsf{A}_3)$ guarantees A_3 runs in parallel time σ_{MHF} with $\mathsf{poly}(t)$ processors (as required in Definition 1). Without loss of generality, we assume that both $\mathsf{A}_{\mathsf{MHF},1}$ and $\mathsf{A}_{\mathsf{MHF},2}$ have harcoded $\mathsf{pp}_{\mathsf{VDS}} \leftarrow\!\!\$\, \mathsf{Setup}_{\mathsf{VDS}}(1^\lambda)$, $\mathsf{k} \leftarrow\!\!\$\, \mathsf{KGen}_{\mathsf{PRF}}(1^\lambda)$, $r_i \leftarrow\!\!\$\, \{0,1\}^{d \cdot s_p}$ for every $i \in [u]$, $\mathsf{chall} \leftarrow\!\!\$\, \mathcal{C}$, $i^* \leftarrow\!\!\$\, [u]$, $\mathsf{id}_i \leftarrow\!\!\$\, \mathcal{I}$ for every $i \in [u] \setminus \{i^*\}$, and $j^* \leftarrow\!\!\$\, [q_{\mathsf{H}}]$.[29]

$\mathsf{A}_{\mathsf{MHF}} = (\mathsf{A}_{\mathsf{MHF},1}, \mathsf{A}_{\mathsf{MHF},2})$ is defined as follows:

[29] Observe that q_{H} is unknown but upper-bounded by $\mathsf{poly}(\lambda)$.

$\mathsf{A}_{\mathsf{MHF},1}(1^\lambda, 1^t, \mathsf{pp}_{\mathsf{MHF}}, \mathsf{id}^*)$**:** On input the security parameter 1^λ, the time parameter 1^t, the public parameters $\mathsf{pp}_{\mathsf{MHF}}$, and the input $\mathsf{id}^* \in \mathcal{X}_{\mathsf{MHF}}$, $\mathsf{A}_{\mathsf{MHF},1}$ proceeds as follows:

1. Set $\mathsf{id}_{i^*} = \mathsf{id}^*$.
2. For every $i \in [u]$, compute $v_i = \mathsf{Eval}_{\mathsf{MHF}}(\mathsf{pp}_{\mathsf{MHF}}, \mathsf{id}_i)$.
3. Execute $\mathsf{A}_1(1^\lambda, \mathsf{ek}, \mathsf{pk}, \mathsf{vk})$ where $\mathsf{ek} = (\mathsf{pp}_{\mathsf{MHF}}, \mathsf{pp}_{\mathsf{VDS}})$ and $\mathsf{pk} = \mathsf{vk} = \mathsf{pp}_{\mathsf{VDS}}$.
4. Answer the incoming random oracle queries (submitted by A_1) as follows:
   (a) On input the random oracle query $(v', \mathsf{id}') \in \mathcal{Y}_{\mathsf{MHF}} \times \mathcal{X}_{\mathsf{MHF}}$ such that $v_i \neq v'$ or $\mathsf{id}_i \neq \mathsf{id}'$ for every $i \in [u]$, $\mathsf{A}_{\mathsf{MHF},1}$ returns $r' = \mathsf{F}_{\mathsf{PRF}}(\mathsf{k}, (v', \mathsf{id}'))$.
   (b) On the other hand, on input the random oracle query $(v', \mathsf{id}') \in \mathcal{Y}_{\mathsf{MHF}} \times \mathcal{X}_{\mathsf{MHF}}$ such that $v_i = v'$ and $\mathsf{id}_i = \mathsf{id}'$ for some $i \in [u]$, $\mathsf{A}_{\mathsf{MHF},1}$ returns r_i.
5. Eventually, $\mathsf{A}_1(1^\lambda, \mathsf{ek}, \mathsf{pk}, \mathsf{vk})$ outputs $(m_1, \ldots, m_u, \mathsf{state})$.
6. For every $i \in [u]$, compute h_i and $\mathsf{c}_i = (\mathsf{D}_i, \mathsf{aux}_i)$ where $f_i(X) = r_i \oplus m_i$ and $(\mathsf{D}_i, \mathsf{h}_i, \mathsf{aux}_i) = \mathsf{GenData}_{\mathsf{VDS}}(\mathsf{pp}_{\mathsf{VDS}}, f_i, p)$.
7. Execute $\mathsf{A}_2(1^\lambda, (\mathsf{id}_i, \mathsf{h}_i, \mathsf{c}_i)_{i\in[u]}, \mathsf{state})$ and answer the incoming random oracle queries (submitted by A_2) as defined in Item 4.
8. Finally, return α output by $\mathsf{A}_2(1^\lambda, (\mathsf{id}_i, \mathsf{h}_i, \mathsf{c}_i)_{i\in[u]}, \mathsf{state})$.

$\mathsf{A}_{\mathsf{MHF},2}(1^\lambda, 1^t, \mathsf{pp}_{\mathsf{MHF}}, \mathsf{id}^*, \alpha)$**:** On input the security parameter 1^λ, the time parameter 1^t, the public parameters $\mathsf{pp}_{\mathsf{MHF}}$, the input $\mathsf{id}^* \in \mathcal{X}_{\mathsf{MHF}}$, and the pre-computed string α, $\mathsf{A}_{\mathsf{MHF},2}$ proceeds as follows:

1. Set $\mathsf{id}_{i^*} = \mathsf{id}^*$.
2. Execute $\mathsf{A}_3(1^\lambda, \mathsf{ek}, \mathsf{pk}, \mathsf{vk}, (\mathsf{id}_i)_{i\in[u]}, \mathsf{chall}, \alpha)$.
3. Answer the incoming random oracle queries (submitted by A_3) as follows:
   (a) On input the j-th random oracle query $(v'_j, \mathsf{id}'_j) \in \mathcal{Y}_{\mathsf{MHF}} \times \mathcal{X}_{\mathsf{MHF}}$ such that $j \neq j^*$, $\mathsf{A}_{\mathsf{MHF},2}$ returns $r' = \mathsf{F}_{\mathsf{PRF}}(\mathsf{k}, (v'_j, \mathsf{id}'_j))$.
   (b) On the other hand, on input the j-th random oracle query $(v'_j, \mathsf{id}'_j) \in \mathcal{Y}_{\mathsf{MHF}} \times \mathcal{X}_{\mathsf{MHF}}$ such that $j = j^*$, $\mathsf{A}_{\mathsf{MHF},2}$ stops and outputs v'_j.

First, observe that $\mathsf{A}_{\mathsf{MHF}}$ is valid with respect to the MHF experiment (see Definition 1). Indeed, $\mathsf{A}_{\mathsf{MHF},1}$ satisfies the following conditions:

1. $\mathsf{A}_{\mathsf{MHF},1}$ outputs α which, in turn, is output by A_2. Since A_2 is valid according to Lemma 11, we conclude that $|\alpha| \leq n$.

2. $\mathsf{A}_{\mathsf{MHF},2}$ has the same running time of A_3 which, in turn, runs in parallel time σ_{MHF} with $\mathsf{poly}(t)$.

Assume that $\mathbf{E}$ holds (i.e., there exists $i \in [u]$ and $(v', \mathsf{id}') \in \mathcal{Q}_{\mathsf{A}_3,\mathsf{H}}$ such that $\mathsf{Eval}_{\mathsf{MHF}}(\mathsf{pp}_{\mathsf{MHF}}, \mathsf{id}_i) = v_i = v'$). Conditioned to $\mathbf{E}$, suppose that A_3 queries (v_i, id') (for some id') to the random oracle during the j-th query (v'_j, id'_j) (i.e., $v'_j = v_i$). Conditioned to $\mathbf{E}$ and $j^* = j$, it is easy to see that $\mathsf{A}_{\mathsf{MHF},2}$ correctly simulates A_3's view until the j^*-th oracle query. Moreover, conditioned to $\mathbf{E}$ (i.e., there exists $i \in [u]$ and $(v', \mathsf{id}') \in \mathcal{Q}_{\mathsf{A}_3,\mathsf{H}}$ such that $\mathsf{Eval}_{\mathsf{MHF}}(\mathsf{pp}_{\mathsf{MHF}}, \mathsf{id}_i) = v_i = v'$), we have that

- $i^* = i$ (i.e., the input $\mathsf{id}_{i^*} = \mathsf{id}^*$ corresponds to the i-th id_i such that $(v_i, \mathsf{id}') \in \mathcal{Q}_{\mathsf{A}_3,\mathsf{H}}$ where $v_i = \mathsf{Eval}_{\mathsf{MHF}}(\mathsf{pp}_{\mathsf{MHF}}, \mathsf{id}_i)$) happens with probability $\frac{1}{u}$.

- Conditioned to $i^* = i$, $j^* = j$ (i.e., the case for which $\mathsf{A}_{\mathsf{MHF},2}$ wins against the MHF experiment) happens with probability $\frac{1}{\mathcal{Q}_{\mathsf{A}_3,\mathsf{H}}} \leq \frac{1}{q_h}$.

By combining Equation (5) and the above observations, we conclude that $\mathsf{A}_{\mathsf{MHF}} = (\mathsf{A}_{\mathsf{MHF},1}, \mathsf{A}_{\mathsf{MHF},2})$ is valid (with respect to the MHF exerpeiment) and has an advantage of at least $\mathbb{P}[\mathbf{E}] \cdot \frac{1}{q_{\mathsf{H}} \cdot u} = \epsilon_{\mathsf{MHF}}$. This concludes the proof. □

Lemma 12. $\mathbf{H}^2(\lambda, t, n, u) \approx_{\frac{u}{|\mathcal{X}_{\mathsf{MHF}}|}} \mathbf{H}^3(\lambda, t, n, u)$.

Proof. The only difference between these two hybrid experiments is that the challenger aborts when A_1 submits a random oracle query (v', id') such that $\mathsf{id}_i = \mathsf{id}'$ for some $i \in [u]$, where id_i is the i-th challenge random identifier sampled by the challenger. Since A_1 does not know the value of $\mathsf{id}_1, \dots, \mathsf{id}_u$ (which are only revealed to A_2) we have that A_1 submits a random oracle query (v', id') such that $\mathsf{id}' = \mathsf{id}_i$ (for some $i \in [u]$) with probability at most $\frac{u}{|\mathcal{X}_{\mathsf{MHF}}|}$. This concludes the proof. □

Lemma 13. $\mathbf{H}^3(\lambda, t, n, u) \equiv \mathbf{H}^4(\lambda, t, n, u)$.

Proof. It is easy to see that these two hybrids are identical. This is because A_1 does not submit a random oracle query (v', id') such that $\mathsf{id}' = \mathsf{id}_i$ (for some $i \in [u]$) where id_i is the challange random identifier sampled by the challenger (see definition of $\mathbf{H}^3(\lambda, t, n, u)$). Hence, the challenger of $\mathbf{H}^4(\lambda, t, n, u)$, which samples $(f_1(X), \dots, f_u(X)) \leftarrow\$ \mathbf{F}^u_{d-1,p}$ and programs the random oracle as $\mathsf{H}(v_i, \mathsf{id}_i) = \mathsf{H}(\mathsf{Eval}(\mathsf{pp}_{\mathsf{MHF}}, \mathsf{id}_i), \mathsf{id}_i) = r_i = f_i(X) \oplus m_i$ only after it receives $m_1, \dots, m_u$ from A_1 (as defined in $\mathbf{H}^4(\lambda, t, n, u)$), is equivalent to the challenger of $\mathbf{H}^3(\lambda, t, n, u)$. Moreover, since $r_i = f_i(X) \oplus m_i$ (for every $i \in [u]$), we have that the output of the encoding algorithm is correctly distributed. This is because $f_i(X) = \mathsf{H}(\mathsf{Eval}(\mathsf{pp}_{\mathsf{MHF}}, \mathsf{id}_i), \mathsf{id}_i) \oplus m_i = r_i \oplus m_i = f_i(X) \oplus m_i \oplus m_i = f_i(X)$. This concludes the proof. □

Lemma 14. *If* Π_{VDS} *is* $(\epsilon_{\mathsf{VDS}})$*-sound then*

$$\mathbf{H}^4(\lambda, t, n, u) \approx_{u \cdot \epsilon_{\mathsf{VDS}}} \mathbf{H}^5(\lambda, t, n, u).$$

Proof. Assume there exists a valid PPT distinguisher $\mathsf{A} = (\mathsf{A}_1, \mathsf{A}_2, \mathsf{A}_3)$ that distinguishes between $\mathbf{H}^4(\lambda, t, n, u)$ and $\mathbf{H}^5(\lambda, t, n, u)$ with advantage greater than $u \cdot \epsilon_{\mathsf{VDS}}$.

Let $\mathbf{E}$ be the event that there exists $i \in [u]$ such that $\mathsf{Verify}(\mathsf{vk}, \mathsf{h}_i, \mathsf{chall}, \pi_i) = 1 \wedge y_i \neq f_i(\mathsf{chall})$ where $\pi_i = (y_i, \pi'_i)$ is the proof output by A_3. Also, let $\mathbf{E}^*$ be the event that $\mathsf{A} = (\mathsf{A}_1, \mathsf{A}_2, \mathsf{A}_3)$ outputs $b = 1$. Then, the advantage of A can be rewritten as follows:

$$\left| \mathbb{P}[\mathbf{E}^* | \mathbf{E}] \cdot \mathbb{P}[\mathbf{E}] - \mathbb{P}[\mathbf{E}^* | \neg\mathbf{E}] \cdot \mathbb{P}[\neg\mathbf{E}] \right| > u \cdot \epsilon_{\mathsf{VDS}}.$$

We observe that $\mathbb{P}[\mathbf{E}^* | \neg\mathbf{E}] = 0$ when $\neg\mathbf{E}$ occurs (this is because, conditioned to $\neg\mathbf{E}$, the hybrids $\mathbf{H}^4(\lambda, t, n, u)$ and $\mathbf{H}^5(\lambda, t, n, u)$ are identical). Hence, it must be that

$$\mathbb{P}[\mathbf{E}^* | \mathbf{E}] \cdot \mathbb{P}[\mathbf{E}] \geq \mathbb{P}[\mathbf{E}] > u \cdot \epsilon_{\mathsf{VDS}}. \tag{6}$$

By leveraging the fact that $\mathbb{P}[\mathbf{E}] > u \cdot \epsilon_{\mathsf{VDS}}$, we build an adversary $\mathsf{A}_{\mathsf{VDS}}$ that breaks the $(\epsilon_{\mathsf{VDS}})$-soundness of Π_{VDS}. $\mathsf{A}_{\mathsf{VDS}}$ is defined as follows:

1. Sample $(f_1(X), \dots, f_u(X)) \leftarrow\$ \mathbf{F}^u_{d-1,p}$ and $i^* \leftarrow\$ [u]$.

2. Send $f_{i^*}(X)$ to the challenger (i.e., $\mathsf{A}_{\mathsf{VDS}}$ will play the VDS's experiment with respect to $f_{i^*}(X)$).

3. Receive $\mathsf{pp}_{\mathsf{VDS}}$, D_{i^*}, h_{i^*}, aux_{i^*} from the challenger (computed by the challenger using $f_{i^*}(X)$).

4. For every $i \in [u]$, sample $\mathsf{id}_i \leftarrow\!\$\, \mathcal{I}$ and compute $v_i = \mathsf{Eval}_{\mathsf{MHF}}(\mathsf{pp}_{\mathsf{MHF}}, \mathsf{id}_i)$.

5. Compute $\mathsf{pp}_{\mathsf{MHF}} \leftarrow\!\$\, \mathsf{Setup}_{\mathsf{MHF}}(1^\lambda, 1^t)$ and $\mathsf{k} \leftarrow\!\$\, \mathsf{KGen}_{\mathsf{PRF}}(1^\lambda)$.

6. Set $\mathsf{ek} = (\mathsf{pp}_{\mathsf{MHF}}, \mathsf{pp}_{\mathsf{VDS}})$, $\mathsf{pk} = \mathsf{vk} = \mathsf{pp}_{\mathsf{VDS}}$, and $\mathsf{c}_{i^*} = (\mathsf{D}_{i^*}, \mathsf{aux}_{i^*})$.

7. For every $i \in [u] \setminus \{i^*\}$, compute h_i and $\mathsf{c}_i = (\mathsf{D}_i, \mathsf{aux}_i)$ where $(\mathsf{D}_i, \mathsf{h}_i, \mathsf{aux}_i) = \mathsf{GenData}_{\mathsf{VDS}}($ $\mathsf{pp}_{\mathsf{VDS}}, f_i, p)$.

8. Execute $\mathsf{A}_1(1^\lambda, \mathsf{ek}, \mathsf{pk}, \mathsf{vk})$ and answer the incoming random oracle queries as follows:

   (a) On input the random oracle query $(v', \mathsf{id}') \in \mathcal{Y}_{\mathsf{MHF}} \times \mathcal{X}_{\mathsf{MHF}}$ such that $\mathsf{id}' \neq \mathsf{id}_i$ for every $i \in [u]$, $\mathsf{A}_{\mathsf{VDS}}$ returns $r' = \mathsf{F}_{\mathsf{PRF}}(\mathsf{k}, (v', \mathsf{id}'))$.

   (b) On input the random oracle query $(v', \mathsf{id}') \in \mathcal{Y}_{\mathsf{MHF}} \times \mathcal{X}_{\mathsf{MHF}}$ such that $\mathsf{id}' = \mathsf{id}_i$ for some $i \in [u]$, $\mathsf{A}_{\mathsf{VDS}}$ aborts.

9. Eventually, $\mathsf{A}_1(1^\lambda, \mathsf{ek}, \mathsf{pk}, \mathsf{vk})$ outputs $(m_1, \dots, m_u, \mathsf{state})$.

10. For every $i \in [u]$, set $\mathsf{H}(v_i, \mathsf{id}_i) = r_i = f_i(X) \oplus m_i$.

11. Execute $\mathsf{A}_2(1^\lambda, (\mathsf{id}_i, \mathsf{h}_i, \mathsf{c}_i)_{i\in[u]}, \mathsf{state})$ and answer the incoming random oracle queries as follows:

   (a) On input the random oracle query $(v', \mathsf{id}') \in \mathcal{Y}_{\mathsf{MHF}} \times \mathcal{X}_{\mathsf{MHF}}$ such that $v' \neq v_i$ or $\mathsf{id}' \neq \mathsf{id}_i$ for every $i \in [u]$, $\mathsf{A}_{\mathsf{VDS}}$ returns $r' = \mathsf{F}_{\mathsf{PRF}}(\mathsf{k}, (v', \mathsf{id}'))$.

   (b) On input the random oracle query $(v', \mathsf{id}') \in \mathcal{Y}_{\mathsf{MHF}} \times \mathcal{X}_{\mathsf{MHF}}$ such that $v' = v_i$ and $\mathsf{id}_i = \mathsf{id}'$ for some $i \in [u]$, $\mathsf{A}_{\mathsf{VDS}}$ returns r_i.

12. Eventually, $\mathsf{A}_2(1^\lambda, (\mathsf{id}_i, \mathsf{h}_i, \mathsf{c}_i)_{i\in[u]}, \mathsf{state})$ outputs α.

13. Sample $\mathsf{chall} \leftarrow\!\$\, \mathcal{C}$.

14. Execute $\mathsf{A}_3(1^\lambda, \mathsf{ek}, \mathsf{pk}, \mathsf{vk}, (\mathsf{id}_i)_{i\in[u]}, \mathsf{chall}, \alpha)$ and answer the incoming random oracle queries as follows:

   (a) On input the random oracle query $(v', \mathsf{id}') \in \mathcal{Y}_{\mathsf{MHF}} \times \mathcal{X}_{\mathsf{MHF}}$ such that $v' \neq v_i$ for every $i \in [u]$, $\mathsf{A}_{\mathsf{VDS}}$ returns $r' = \mathsf{F}_{\mathsf{PRF}}(\mathsf{k}, (v', \mathsf{id}'))$.

   (b) On input the random oracle query $(v', \mathsf{id}') \in \mathcal{Y}_{\mathsf{MHF}} \times \mathcal{X}_{\mathsf{MHF}}$ such that $v' = v_i$ for some $i \in [u]$, $\mathsf{A}_{\mathsf{VDS}}$ aborts.

15. Finally, output $(\mathsf{chall}, y_{i^*}, \pi'_{i^*})$ where $\pi_{i^*} = (y_{i^*}, \pi'_{i^*})$ is the i^*-th proof output of $\mathsf{A}_3(1^\lambda,$ $\mathsf{ek}, \mathsf{pk}, \mathsf{vk}, (\mathsf{id}_i)_{i\in[u]}, \mathsf{chall}, \alpha)$.

It is easy to see that $\mathsf{A}_{\mathsf{VDS}}$ correctly simulates the views of both A_1, A_2, and A_3. Assume that $\mathbf{E}$ happens and let $i \in [u]$ be an index such that $y_i \neq f_i(\mathsf{chall})$. Conditioned to $\mathbf{E}$ and $i^* = i$ (which happends with probability $\frac{1}{u}$), we have that $y_{i^*} \neq f_{i^*}(\mathsf{chall})$ and $\mathsf{Verify}(\mathsf{vk}, \mathsf{h}_{i^*}, \mathsf{chall}, \pi_{i^*}) = 1$ which, in turn, implies that $\mathsf{Verify}_{\mathsf{VDS}}(\mathsf{pp}_{\mathsf{VDS}}, \mathsf{h}_{i^*}, \mathsf{chall}, \pi'_{i^*}) = 1$. By combining Equation (6) with the observations above, we conclude that $\mathsf{A}_{\mathsf{VDS}}$'s advantage is at least $\mathbb{P}[\mathbf{E}] \cdot \frac{1}{u} = \epsilon_{\mathsf{VDS}}$. This concludes the proof. □

Lemma 15. $\mathbb{P}\big[\mathbf{H}^5_d(\lambda, t, n, u) = 1\big] \leq \frac{d-1}{|\mathbb{Z}_q|} + \frac{1}{2^c}$.

Proof. Assume there exists a valid PPT adversary $\mathsf{A} = (\mathsf{A}_1, \mathsf{A}_2, \mathsf{A}_3)$ such that

$$\mathbb{P}\big[\mathbf{H}^4_d(\lambda, t, n, u) = 1\big] > \frac{d-1}{|\mathbb{Z}_q|} + \frac{1}{2^c}. \tag{7}$$

Then, we build an adversary $\mathsf{A}' = (\mathsf{A}'_1, \mathsf{A}'_2)$ that contradicts Theorem 2. Without loss of generality, we assume that both A'_1 and A'_2 have harcoded $(\mathsf{ek}, \mathsf{pk}, \mathsf{vk}) \leftarrow\!\!\$\ \mathsf{Setup}(1^\lambda, 1^t)$, $\mathsf{k} \leftarrow\!\!\$\ \mathsf{KGen}_{\mathsf{PRF}}(1^\lambda)$, $\mathsf{id}_i \leftarrow\!\!\$\ \mathcal{I}$ for $i \in [u]$, where $\mathsf{ek} = (\mathsf{pp}_{\mathsf{MHF}}, \mathsf{pp}_{\mathsf{VDS}})$ and $\mathsf{pk} = \mathsf{vk} = \mathsf{pp}_{\mathsf{VDS}}$. $\mathsf{A}' = (\mathsf{A}'_1, \mathsf{A}'_2)$ is defined as follows:

$\mathsf{A}'_1(1^\lambda, f_1, \dots, f_u)$**:** On input the security parameter 1^λ and u univariate polynomials $f_1(X), \dots, f_u(X) \in \mathbb{Z}_p[X]$ of degree $d-1$, A'_1 proceeds as follows:

1. For every $i \in [u]$, compute $v_i = \mathsf{Eval}_{\mathsf{MHF}}(\mathsf{pp}_{\mathsf{MHF}}, \mathsf{id}_i)$.
2. Execute $\mathsf{A}_1(1^\lambda, \mathsf{ek}, \mathsf{pk}, \mathsf{vk})$ and answer to the incoming random oracle queries as follows:
   (a) On input the random oracle query $(v', \mathsf{id}') \in \mathcal{Y}_{\mathsf{MHF}} \times \mathcal{X}_{\mathsf{MHF}}$ such that $\mathsf{id}' \neq \mathsf{id}_i$ for every $i \in [u]$, $\mathsf{A}_{\mathsf{VDS}}$ returns $r' = \mathsf{F}_{\mathsf{PRF}}(\mathsf{k}, (v', \mathsf{id}'))$.
   (b) On input the random oracle query $(v', \mathsf{id}') \in \mathcal{Y}_{\mathsf{MHF}} \times \mathcal{X}_{\mathsf{MHF}}$ such that $\mathsf{id}' = \mathsf{id}_i$ for some $i \in [u]$, $\mathsf{A}_{\mathsf{VDS}}$ aborts.[30]
3. Eventually, $\mathsf{A}_1(1^\lambda, \mathsf{ek}, \mathsf{pk}, \mathsf{vk})$ outputs $(m_1, \dots, m_u, \mathsf{state})$.
4. For every $i \in [u]$, set $\mathsf{H}(v_i, \mathsf{id}_i) = r_i = f_i(X) \oplus m_i$ and compute $(\mathsf{h}_i, \mathsf{c}_i) = \mathsf{Encode}(\mathsf{ek}, m_i, \mathsf{id}_i)$ using the fact that $\mathsf{H}(v_i, \mathsf{id}_i) = r_i = f_i(X) \oplus m_i$.
5. Execute $\mathsf{A}_2(1^\lambda, (\mathsf{id}_i, \mathsf{h}_i, \mathsf{c}_i)_{i \in [u]}, \mathsf{state})$ and answer the incoming oracle queries as follows:
   (a) On input the random oracle query $(v', \mathsf{id}') \in \mathcal{Y}_{\mathsf{MHF}} \times \mathcal{X}_{\mathsf{MHF}}$ such that $v' \neq v_i$ or $\mathsf{id}' \neq \mathsf{id}_i$ for every $i \in [u]$, $\mathsf{A}_{\mathsf{VDS}}$ returns $r' = \mathsf{F}_{\mathsf{PRF}}(\mathsf{k}, (v', \mathsf{id}'))$.
   (b) On input the random oracle query $(v', \mathsf{id}') \in \mathcal{Y}_{\mathsf{MHF}} \times \mathcal{X}_{\mathsf{MHF}}$ such that $v' = v_i$ and $\mathsf{id}_i = \mathsf{id}'$ for some $i \in [u]$, $\mathsf{A}_{\mathsf{VDS}}$ returns r_i.
6. Eventually, return α which is the output of $\mathsf{A}_2(1^\lambda, (\mathsf{id}_i, \mathsf{h}_i, \mathsf{c}_i)_{i \in [u]}, \mathsf{state})$.

$\mathsf{A}'_2(1^\lambda, x, \alpha)$**:** On input the security parameter 1^λ, a point $x \in \mathbb{Z}_q$, and a string α, A'_2 proceeds as follows:

1. Execute $\mathsf{A}_3(1^\lambda, \mathsf{ek}, \mathsf{pk}, \mathsf{vk}, (\mathsf{id}_i)_{i \in [u]}, x, \alpha)$ and answer the incoming random oracle queries as follows:
   (a) On input the random oracle query $(v', \mathsf{id}') \in \mathcal{Y}_{\mathsf{MHF}} \times \mathcal{X}_{\mathsf{MHF}}$ such that $v' \neq v_i$ for every $i \in [u]$, $\mathsf{A}_{\mathsf{VDS}}$ returns $r' = \mathsf{F}_{\mathsf{PRF}}(\mathsf{k}, (v', \mathsf{id}'))$.
   (b) On input the random oracle query $(v', \mathsf{id}') \in \mathcal{Y}_{\mathsf{MHF}} \times \mathcal{X}_{\mathsf{MHF}}$ such that $v' = v_i$ for some $i \in [u]$, $\mathsf{A}_{\mathsf{VDS}}$ aborts.[31]
2. Finally, output $(y_1, \dots, y_u)$ where $\pi_i = (y_i, \pi'_i)$ is the i-th proof output by $\mathsf{A}_3(1^\lambda, \mathsf{ek}, \mathsf{pk}, \mathsf{vk}, (\mathsf{id}_i)_{i \in [u]}, x, \alpha)$.

[30] Note that this first aborting condition is fundamental for concluding the proof correctly. This is because, at this point, A'_1 does not know $m_1, \dots, m_u$ (which are chosen by A_1). Hence, A'_1 would not be able to program the random oracle output of $\mathsf{H}(v_i, \mathsf{id}_i) = f_i(X) \oplus m_i$ for $i \in [u]$.

[31] Note that this second aborting condition is fundamental for concluding the proof correctly. This is because A'_2 can simulate A_3's view without knowing $r_1, \dots, r_u$, $f_1(X), \dots, f_u(X)$, and $m_1, \dots, m_u$ which are too large to be encoded into α.

First, observe that, conditioned to $\mathbf{H}^5(\lambda, t, n, u) = 1$, we have that $y_i = f_i(x)$ for every $i \in [u]$. Second, since $\mathsf{A} = (\mathsf{A}_1, \mathsf{A}_2, \mathsf{A}_3)$ is valid, we know that $|\alpha| \le n$ and

$$n = \min\{n_{\mathsf{MHF}}, d(u \cdot s_p - s_q) - c\}.$$

Hence, by combining Equation (7) and the above arguments, we conclude that $\mathsf{A}' = (\mathsf{A}'_1, \mathsf{A}'_2)$ is valid and outputs $(f_1(x), \dots f_u(x))$ with probability greater than $\frac{d-1}{|\mathbb{Z}_q|} + \frac{1}{2^c}$. This contradicts Theorem 2 and concludes the proof. □

Theorem 9 follows by combining Lemmas 10 to 15. This concludes the proof.

A.10 Proof of Corollary 6

The corollary follows by plugging Corollaries 1, 4 and 5 into Theorems 8 and 9 and Definition 18, and observing that

- under the collision resistance assumption (or the RO), there exists an $(\mathsf{negl}(\lambda))$-secure PRF scheme,
- $u, d \in \mathsf{poly}(\lambda)$ thus $u \cdot q_{\mathsf{H}}\epsilon_{\mathsf{MHF}}$, $u \cdot \epsilon_{\mathsf{VDS}}$, $\frac{u}{|\mathcal{X}_{\mathsf{MHF}}|} = \frac{u}{2^\lambda}$ (of Theorem 9) are negligible,
- by setting s_p, s_q, and c as defined in Corollary 4 (see also the corresponding proof) then $\frac{d-1}{|\mathbb{Z}_q|} + \frac{1}{2^c} \le O(\frac{1}{2^\lambda})$ and $d(u \cdot s_p - s_q) - c = d \cdot u \cdot \lambda^{1+\delta_1} - O(d \cdot \lambda)$ (of Theorem 9),
- by leveraging Corollary 1 we can set n_{MHF} to be larger than $d(u \cdot s_p - s_q) - c$. That, in turn, implies $n = \min\{n_{\mathsf{MHF}}, d(u \cdot s_p - s_q) - c\}$ (of Theorem 9) equal to $d(u \cdot s_p - s_q) - c$,
- the (γ)-expansion of Construction 2 is exactly the (γ)-expansion of the underlying VDS scheme (see Theorem 7) and Construction 2 leverages polynomials of degree $d - 1$ from $\mathbb{Z}_p[X]$ where p is of $s_p = \lambda^{1+\delta_1}$ bits (according to our choice of parameters); thus $\gamma = 2 \cdot d^{\delta_2} \cdot \log^{o(1)}(p) = 2 \cdot d^{\delta_2} \cdot \lambda^{o(1)(1+\delta_1)}$ for any choice of positive constant $\delta_2 > 0$.

Lastly, as for the (η)-gap, it follows by observing that $n = \min\{n_{\mathsf{MHF}}, d(u \cdot s_p - s_q) - c\} = d(u \cdot s_p - s_q) - c$ (as defined above), $d(u \cdot s_p - s_q) - c = d \cdot u \cdot \lambda^{1+\delta_1} - (d+1) \cdot \lambda$ (for $s_p = \lambda^{1+\delta_1}$, $s_q = \lambda$, and $c = \lambda$ as defined in Corollary 4. See also the corresponding proof), and

$$\begin{aligned} d \cdot u \cdot \lambda^{1+\delta_1} - (d+1) \cdot \lambda = (1 - \eta)(u \cdot d \cdot \lambda^{1+\delta_1}) \implies \\ \eta = 1 - \frac{d \cdot u \cdot \lambda^{1+\delta_1} - (d+1) \cdot \lambda}{u \cdot d \cdot \lambda^{1+\delta_1}} \implies \eta = \frac{(d+1) \cdot \lambda}{u \cdot d \cdot \lambda^{1+\delta_1}} \in O\left(\frac{1}{u \cdot \lambda^{\delta_1}}\right), \end{aligned}$$

where we used the fact that $s_p = \lambda^{1+\delta_1}$, $s_q = \lambda$, and $c = \lambda$ in Corollary 4 (see also the corresponding proof).