UDC 681.3.06

Proof of Data Possession Protocol with Hash-based Deterministic Challenges and Privately Verifiable Payments

Maksym Strielnikov[1], Liudmyla Kovalchuk[1]

[1] *National Technical University of Ukraine 'Kyiv Polytechnic Institute', Ukraine*

Abstract

The research presents a novel protocol for remote verification of data outsourced to third-party storage. The protocol aims to verify the possession of data in potentially untrusted storage without downloading. We identified challenges in existing proof-of-possession protocols (PDP) by conducting a comprehensive literature review. We stated the optimal threshold for the minimal communication cost needed in existing PDP protocols to ensure the validity of the target percentage of data blocks while maintaining high confidence. Building on these findings, we propose our own PDP requires a fixed amount of communication and offers practically deterministic validity guarantees based on the security of cryptographic hash functions. We show a scenario for monetization in both cloud-native and blockchain environments to incentivize storage providers. A rigorous security analysis demonstrates resilience against forgery attacks aimed to falsify integrity checks or compromise verifiability assumptions. Our protocol significantly reduces communication overhead compared to existing solutions while eliminating the cheating probability to negligible levels.

Keywords: *proof of data possession, data storage outsourcing, verifiable storage, remote integrity check, cryptographic hash.*

Introduction

Proof of Data Possession (PDP) protocols, a specialized class of cryptographic schemes, have been developed since the early 2000s to enable verifiable integrity of remotely stored data without requiring full data retrieval. The foundational work by Golle et al. (Golle, Jarecki, and Mironov 2002) introduced the use of RSA-based signatures for remote integrity checks [1], establishing the cryptographic basis for ensuring data integrity in outsourced storage. This approach was formalized into a comprehensive PDP framework by Ateniese et al. (Ateniese et al. 2007), who proposed a protocol relying on the RSA-based digital signatures with knowledge-of-exponent assumption, enabling probabilistic verification of data possession [3]. RSA-based signature was replaced by a specially designed symmetric authenticated encryption scheme at (Ateniese et al. 2008) to achieve computational improvements [6]. Concurrently, Juels and Kaliski at (Juels and Jr. 2007) introduced a PDP variant using HMAC for data blocks authentication [5]. Xu and Chang at (Xu, Chang, and Zhou 2012) integrated coding techniques with zero-knowledge proofs (ZKPs) based on the knowledge-of-exponent assumption, allowing verification even in the presence of partial data corruption [8]. The incorporation of error-correcting codes was advanced by Han et al. (Han et al. 2013), who utilized Maximum Rank Distance (MRD) codes to enable partial restoration of corrupted data blocks in PDP protocols [9]. More recently, Kaaniche and Laurent (Kaaniche, Moustaine, and Laurent 2014) proposed a PDP protocol leveraging ZKPs and bilinear pairings on elliptic curve groups, offering enhanced security and flexibility for modern cloud storage environments [11].

The integration of Merkle trees into PDP protocols was introduced by Niaz and Wu (Niaz and Saake 2015), leveraging their efficiency in verifying large datasets in cloud storage through hierarchical hashing, reducing communication overhead [13].

Walker provided an exhaustive overview of the multiple PDP protocols in classical client-

server setting and pointed to their drawbacks in (Walker, Hewage, and Jayal 2022) [16]. Additionally, Yang et al. (Yang et al. 2024) developed PDP scheme to verify cached data in cloud and edge computing nodes, introducing a rewarding mechanism for verified nodes with applications in web security [19].

The application of Merkle tree was extended in decentralized systems with the use of Merkle Directed Acyclic Graphs (DAGs)in the InterPlanetary File System (IPFS) (Benet 2014) [10], which forms the core storage layer of Filecoin (Protocol Labs 2017) [14]. Filecoin enhances data retrievability and replication using succinct non-interactive arguments of knowledge (SNARKs), ensuring verifiable storage in peer-to-peer networks. Verifiability mechanisms in IPFS were further explored by Azizi et al. (Azizi, Azizi, and Elboukhari 2022), focusing on manual rechecking of verifiability and security guarantees stated in IPFS [15]. Nalina et al. (Nalina et al. 2024) investigated extensions of IPFS for blockchain-integrated storage, addressing scalability and trust challenges [18]. Hall-Andersen and Simkin in (Hall-Andersen, Simkin, and Wagner 2025) examined data integrity and availability in blockchain-based storage, highlighting challenges in decentralized trust models [20]. Dumas et al. (Dumas et al. 2023) proposed a PDP scheme using homomorphic encryption and bilinear pairings to verify evaluations of secret polynomials, offering robust security for distributed storage [17].

Let us examine the structure of the most representative PDP protocols. We propose to have a close look (Ateniese et al. 2008), [6] which was the basis for subsequent works. First, introduce the protocol parties. They are *OWN* for data owner and storage server *SRV*. Before outsourcing data *D*, *OWN* precomputes a certain number of short possession verification tokens v_i, each token covering some set of data blocks. The actual data is then handed over to *SRV*. Subsequently, when *OWN* wants to obtain a proof of data possession, it challenges *SRV* with a set of randomly appeared block indices. In turn, *SRV* must compute a short integrity check over the specified blocks (corresponding to the indices) and return it to *OWN*. The proposed scheme is based entirely on symmetric key cryptography including authenticated encryption AE_K on symmetric key *K*, and two pseudorandom functions (PRFs) f_W and f_Z on symmetric keys *W* and *Z* respectively. The protocol consists of the setup and verification phase. During the setup phase, the owner *OWN* generates in advance number of possible random challenges and the corresponding answers. These answers are called tokens. To produce the i-th token, the owner generates a set of *r* indices as follows:

1. Generate a permutation key $k_i = f_W(i)$ and a challenge nonce $c_i = f_Z(i)$
2. Compute the set of indices $\{ I_j \in [1, \ldots, d] \mid 1 \le j \le r \}$, where $I_j = g_k(j)$ and $g_k(j)$ is the permutation function based on AES (or other symmetric encryption) with secret key k_i
3. Compute the token as $v_i = H(c_i, D[I_1], \ldots, D[I_r])$, using hash function *H*

Basically, each token v_i is the answer we expect to receive from the storage server whenever we challenge it on the randomly selected data blocks$\{D[I_1], \ldots, D[I_r]\}$. The challenge nonce c_i is needed to prevent potential precomputations performed by the storage server. Notice that each token is the output of a cryptographic hash function so its size is small. Once all tokens are computed, the owner outsources the entire set to the server, along with the file *D*, by encrypting each token with an authenticated encryption function AE_K. The setup phase algorithm is shown in Figure 1.

Algorithm 1: Setup

```
Choose parameters c, l, k, L and functions f, g
for i ← 1 to r do
    begin Round i
    Generate k_i = f_W(i) and c_i = f_Z(i)
    Compute
    v_i = H(c_i, D[g_{k_i}(1)], ..., D[g_{k_i}(r)])
    Compute v_i = AE_K(i, v_i)
    end
end
Send to SRV: (D, {[i, v_i] for 1 ≤ i ≤ t})
```

Figure 1: Setup phase

To verify the i-th proof of possession, *OWN* generates the i-th token key k_i as in step 1 of the setup algorithm on Figure 1. Note that *OWN* only needs to store the master keys *W*, *Z*, and *K*, and the current token index *i*. He also recomputes c_i. Then, *OWN* sends to *SRV* both k_i and c_i as showed on the step 2 of algorithm (Figure 1). Having received the message from *OWN*, *SRV* computes:

$$z = H(c_i, D[g_k(1)], \ldots, D[g_k(r)]) \qquad (1)$$

Then *SRV* retrieves v_i and returns *(z, v_i)* to *OWN* who, in turn, computes $v \neq AE_k^{-1}(V_i)$ and checks whether $v \neq (i, z)$. If the check succeeds, *OWN* assumes that *SRV* is storing all of *D* with a certain probability. The verification algorithm shown in Figure 2.

Algorithm 2: Verification phase

1 $\mathcal{OWN}$ computes $k_i = f_W(i)$ and $c_i = f_Z(i)$
2 $\mathcal{OWN}$ sends $\{k_i, c_i\}$ to $\mathcal{SRV}$
3 $\mathcal{SRV}$ computes $z = H(c_i, D[g_{k_i}(1)], ..., D[g_{k_i}(r)])$
4 $\mathcal{SRV}$ sends $\{z, v_i'\}$to $\mathcal{OWN}$
5 $\mathcal{OWN}$ extracts v from v_i'
6 if decryption fails or $v_i \neq (i, z)$ then REJECT.

Figure 2: verification algorithm

The authors don't state any requirements for the implementation of the algorithm AE_K. According to definition of authenticated encryption, it implies that information is symmetrically encrypted aside with some padding data *PAD*.

OWN required to remember *PAD* he used to perform AE_K. On each verification attempt, he can decrypt v_i by AE_K^{-1} yielding *PAD'* then checking $PAD = PAD`$. Decryption is considered failed if the condition doesn't hold. The AE_K works as a symmetric replacement for a digital signature.

A similar protocol was stated in (Juels and Jr. 2007) [5], which also relies on a number of randomly chosen blocks to authenticate, but it employs $HMAC_K$ with secret key *K* instead of AE_K. The other similar construction based exclusively on pseudo-random functions stated in (Shacham and Waters 2008) [7].

We can observe the same drawbacks in PDP protocols according to mentioned works:

1. Protocols run challenges on certain subset of data blocks which implies cheating probability by avoiding checks on corrupted blocks shown above
2. The number of data blocks to check depends on the size of the entire data and requires balancing between target confidence level and amount of communication to ensure it on verification step
3. Some protocols may involve complex setups that require data owners to keep their private parameters in secret, especially those based on symmetric encryption or SNARKs

1. Estimation of cheating probability in PDP

If the PDP protocol requires querying of *k* blocks of data $\{D_i \subset D: i = \overline{1.k}, k < n\}$, then there exists a non-negligible cheating probability P_{cheat} for the data storage provider to evade checking of potentially corrupted blocks. C_k^n is the number of ways to choose *k* blocks from all blocks *n*. If *m* is the number of missing blocks, then the combination of ways to choose the correct blocks and avoid the corrupted ones is C_k^{n-m}. Then P_{cheat} is the probability of avoiding integrity checking for corrupted blocks is defined according to Ateniese et al. 2008 [6] as:

$$P_{cheat} = \frac{C_k^{n-m}}{C_k^n} \qquad (2)$$

Expand C_k^{n-m} and C_k^n according to definition of factorial and we got the same intermediate result for P_{cheat} shown in Ateniese et al. 2007 [3]:

$$\begin{aligned} P_{cheat} &= \prod_{i=0}^{k} \frac{(n-i)-m}{n-i} \\ &= \prod_{i=0}^{k} \left(1 - \frac{m}{n-i}\right) \\ &\approx \left(1 - \frac{m}{n-i}\right)^k \end{aligned} \qquad (3)$$

We can approximate P_{cheat} by assuming the entire file is large and $n \gg k$. Observe that $\frac{m}{n} = \varepsilon \in (0;1)$ it is the fraction of corrupted blocks is a percentage. Assume that the optimal data owner wants to cover as many blocks as possible with his remote check to ensure a low number of unverified blocks $\varepsilon \ll 1$. Then the probability of detecting cheating *P* for small ε

approximated further using Bernoulli inequality for exponent:

$$P \approx 1 - p_{cheat} \approx 1 - (1 - \varepsilon)^k \approx 1 - e^{-\varepsilon k} \quad (4)$$

In practice, reasonable values for ε are around $10^{-2}..10^{-4}$ to allow at most 1%..0.001% unchecked blocks. We can see that PDP protocols we surveyed are probabilistic, since a non-negligible probability P_{cheat} of check evasion exists for blocks.

2. The optimal number of data blocks to verify

Let's find the optimal value of data blocks $k \rightarrow \min$ to query by maintaining the cheating detection probability $P \rightarrow 1$ for the targeted fraction ε of the data blocks. The condition of maximization P implies minimization of the cheating probability by keeping $P \geq 1 - \sigma$, where $\sigma \ll 1$ is the maximum allowed cheating probability threshold or verification error with the following bound:

$$1 - e^{-\varepsilon k} \geq 1 - \sigma \Rightarrow e^{-\varepsilon k} \leq \sigma \quad (5)$$

then the optimal k is expressed as a dependence on σ and ε:

$$-\varepsilon k \leq ln\sigma \Rightarrow k \geq \frac{-\ln \sigma}{\varepsilon} \quad (6)$$

Reasonable values for σ are around $10^{-2}..10^{-4}$ to tolerate targeted cheating probability $\sigma \leq 1\%$ on $\varepsilon \leq 1\%$ of all data blocks for the single challenge round of PDP. Such a small percentage of potentially allowed corrupted blocks can be restored with the application of error correction codes as proposed in Han et al. 2013 [9]. We got the estimate of the optimal k to ensure the probability of cheating bounded by at most σ with high confidence $P \rightarrow 1$. Then optimal communication complexity O for probabilistic PDP is also proportional to optimal k, but restricts k to be integer rounded up:

$$O \sim \lceil k \rceil = \left\lceil \frac{-\ln \sigma}{\varepsilon} \right\rceil \quad (7)$$

Note that optimal solution for k requires to keep both ε and σ as small as possible values within similar range of $10^{-2}..10^{-4}$ or smaller. Then we can express optimal communication complexity O as a dependency on the number of data blocks $O \propto n$ using approximation $\sigma \approx \varepsilon$:

$$O \sim \lceil k \rceil = \left\lceil \frac{-\ln \sigma}{\varepsilon} \right\rceil \approx \lceil \varepsilon^{-1} \ln \varepsilon^{-1} \rceil = \left\lceil \frac{n}{m} \ln \frac{n}{m} \right\rceil \quad (8)$$

We can see that the optimal number of data blocks k to query from the remote storage depends on the target ratio of available data blocks ε we want to ensure as $O \sim \lceil \varepsilon^{-1} \ln \varepsilon^{-1} \rceil$.

3. Our approach

In this section we present our own PDP with challenges based on hashes. This approach not limited to authentication of subset of data blocks, but can be applied on the whole data volume *D*. The protocol parties are data owner Alice (*A)* who outsources the data to the storage provider Bob (*B)*. The PDP protocol consists of two phases. The first phase involves setting the private and public parameters. The second phase is a sequence of integrity checks *f* performed on the outsourced data *D* within an agreed number of rounds *r* and compared with precomputed challenge answers v_i for $i \in [1, r]$. Define challenge as:

$$v_i = f(D, t_i) = hash(D \mid t_i) \quad (9)$$

where t_i in a one-time nonce. The result of the challenge per round v_i is computationally bound to *D* and one-time parameter t_i because of computational binding property of cryptographic hash function. *B* should find a preimage of the cryptographic hash function to forge the proof v_i without *D*. *A* must provide one-time parameters t_i for the function *f* to prevent *B* from cheating related to precomputing of v_i or brute forcing them. To get a simple PDP with a precomputed

number of rounds *r*, we can simulate the following game:

1. Setup phase:
    a. *A* agree on a number of challenges (rounds) *r* with B
    b. A generates a set of random nonces for per-round challenge $\{t_i\} = \{t_i \leftarrow 1^{\lambda}: i = \overline{1.r}\}$, with bit security of λ
    c. *A* computes answers $\{v_i\}$ to each challenge before any information exchange with B as: $\{v_i = hash(D \mid t_i),\ i = \overline{1.r}\}$
    d. *A* signs *D* with *Sign(D,sk)=(Sig,pk)* using a private key *sk* yielding a pair of signature *Sig* and public key *pk*
    e. *A* publishes *(Sig,pk)* and transfers data *D* to *B* by finishing setup

2. Setup phase A performs *r* challenge rounds for *B* on data *D*. On each i-th round while $i \leq r$:
    a. *A* sends t_i to *B* and asks him to compute $v_i' = hash(D \mid t_i)$
    b. *B* presents v_i' to *A*
    c. *A* checks $v_i \neq v_i'$ by comparing with precomputed answer for the index *i*
    d. If $v_i \neq v_i'$, then *A* consider *B* is cheating and aborts the protocol

3. If all *r* challenges completed successfully, then *A* requests back copy *D'* of the data *D* from *B*
4. On retrieval, *A* verifies the signature by *Verify(D',pk)* to ensure the authenticity of the data implying D=D'
5. If verification is evaluated to *True*, both sides conclude the end of the protocol

The protocol we got achieves private verifiability through the independent computation of a hash function by parties over data *D* with an additional one-time parameter t_i. Private verifiability achieved by comparing results of each party got. Unlike protocols discussed in the literature overview, our protocol requires all blocks of data *D* to be involved in an integrity check, which prevents evasion of checks by any of blocks. The computational complexity to generate a single challenge t_i and response v_i is *O(1)* assuming random oracle access model for the hash function. We can observe that *A* required to keep all *r* pairs of challenges and corresponding responses (v_i,t_i) during the execution of protocol. Each of v_i or t_i has a fixed size *O(1)*, implying that the complexity of the space depends on the number of rounds *O(r)*. The amount of communication needed to send t_i and receive v_i of fixed size within the single round is *O(1)* and does not depends on the size of data *D*. The downsides of the proposed protocol:

1. The protocol has a fixed number of rounds
2. The honest *A* required to store *r* nonces t_i within static number of rounds

4. PDP for storage service with verifiable payments

In this section, we apply the PDP scheme we discovered in the previous section to develop a protocol for verifiable storage with payments. We ensure *B* store data for *r* rounds by hash-based challenges for each round. Corresponded setup algorithm provided on Figure 3.

We will incentivize *B* to follow the protocol to the end by rewarding him for successful challenges. But we need to ensure *A* committed to pay full price for all successful challenges. In blockchain native environments, we can guarantee the ability of *A* to pay the price *Y* by requiring her to transfer funds to smart contract.

Algorithm 3: Setup

Input : D, r
Output: $\bar{t}$, $\bar{v}$, Sig, pk

```
for i ← 1 to r do
    t_i ← 1^λ // where 1^λ is random bit string of length λ
    v_i ← hash(D | t_i)
end
t̄ ← (t_1, t_2, ..., t_r) // t̄ is a vector of random tags
v̄ ← (v_1, v_2, ..., v_r) // v̄ is a vector of hashed tags
sk ← 1^λ
(Sig, pk) ← Sign(D, sk)
return t̄, v̄, Sig, pk
```

Figure 3: Setup (Algorithm 3)

These funds will be locked and unlocked in equal portions $y=Y/r$ gradually for each successful challenge for *B*. On successful challenge, a fraction of the funds y will be transferred to *B* automatically, achieving full payment *Y* for the storage service after *r* rounds. In case of failing challenge, *A* withdraws funds from the smart-contract and terminates its execution.

The financial commitment of *A* can be ensured without blockchain. In this case, *A* and *B* should agree on the usage of third-party payment provider that accepts the full payment from *A* and pays by fractions of the price *y* to *B* after confirmation of each successful challenge by *A*.

The blockchain native approach is more reliable, since the usage of the smart contract, which is used to account payments and track the history of challenges, is publicly verifiable and not profit-biased towards either of sides. The trust to the third-party payment provider or bank may require additional verification for each party.

The complete algorithm with verifiable payments stated in Figure 4.

Algorithm 4: VerifiableStorage

Input : D, r, Y
Output: $Success \in \{True, False\}$
// $\mathcal{A}$ performs the setup of parameters
$\bar{t}, \bar{v}, \mathrm{Sig}, pk \leftarrow \mathrm{Setup}(D, r)$
// $\mathcal{A}$ locks the funds and publishes the address of the smart contract or payment gateway credentials to access the funds
$addr \leftarrow LockFunds(Y)$
// Communication: $\mathcal{A}$ sends (D, r) to $\mathcal{B}$
// Set reward counter for storage provider $\mathcal{B}$ to 0
$\mathcal{B}_{reward} \leftarrow 0$
for $i \leftarrow 1$ ***to*** r **do**
  // Communication: $\mathcal{A}$ sends t_i to $\mathcal{B}$
  // $\mathcal{B}$ computes hash tag v_i
  $v_i' \leftarrow \mathrm{hash}(D \mid t_i)$
  // Communication: $\mathcal{B}$ sends v_i to $\mathcal{A}$
  // $\mathcal{A}$ checks the integrity poof
  if $v_i \neq v_i'$ **then**
    // $\mathcal{A}$ terminate the process if integrity proof is false and withdraw her unused funds
    $Success \leftarrow False$
    return $False$
  end
  // Communication: $\mathcal{A}$ unlock portion of funds for successful i-th challenge for $\mathcal{B}$ as the reward
  $y \leftarrow UnlockFunds(addr)$
  $\mathcal{B}_{reward} := \mathcal{B}_{reward} + y$
end
$Success \leftarrow True$
return $Success$

Figure 4: Verifiable storage 2-party protocol

We assume payment as verifiable if *A* ensures data availability by storage provider *B* first before each payment.

5. Adversary models

5.1 Hash guessing attack

Let's look at the following adversary model. Assume that *B* can convince *A* on i-th round by guessing a hash preimage for v_i without actual knowledge of data *D* with certain probability *P*. Let *n* be the bit length of hash value v_i. Then the probability of guessing a random bit string v_i is:

$$P = \frac{1}{2^n} \tag{10}$$

In practice, the minimal yet secure bit length of cryptographic hash functions such as SHA-2 or SHA-3 is 224 bits, which gives us the probability of a successful guess in the worst case for a single i-th round $1/2^{224}$, which is negligibly small.

5.2 Hash collision search by lookup table

Put T is a set of precomputed nonce values $t_i \in T$ by *A* which she share on i-th round and *r* is the number of rounds. It follows that $|T|=r$, where *r* is relatively small. Suppose *A* and *B* agreed on the number of rounds *r*, and *B* got *D* to store. Malicious *B* can improve guessing attack by generating a lookup table beforehand from many randomly sampled nonces b_j, where $N \gg r$ is big and $1 \leq j \leq N$. Then $T = \left\{\left(b_j, u_j = hash(D \mid b_j)\right), j = \overline{1.N}\right\}$. On this point, *B* can discard *D* entirely. Each time *B* receives nonce t_i from *A*, *B* checks the presence of t_i in the lookup table *T* and return the corresponding u_j as the challenge response if such u_j is present in lookup table *T*. Let's calculate the probability that $\exists t_j \in T, \exists b_j \in B: t_i = b_j \Leftrightarrow P(T \cap B)$. *B* observes a sequence of *r* trials with one outcome from the set {True,False} for each.

Since both t_i and b_j are uniformly distributed bit strings with length n, then the probability of single match guessing is 2^{-n}. The probability $\exists b_j \in B$: $b_j = t_i$:

$$P(t_i \in T) = \frac{N}{2^n} \tag{11}$$

then the probability of an opposite event is:

$$P(t_i \notin B) = 1 - P(t_j \in B) = 1 - \frac{N}{2^n} \tag{12}$$

Considering there are r trials took place:

$$\prod_{i \le r}^{P} (t_i \notin B) = P(A \cap B = \emptyset) = \left(1 - \frac{N}{2^n}\right)^r \approx e^{\frac{-rN}{2^n}} \tag{13}$$

Assuming that $N \ll 2^n$:

$$P(A \cap B = \emptyset) = e^{\frac{-rN}{2^n}} \approx e^0 \approx 1 \Rightarrow P(A \cap B \neq \emptyset) = 0 \tag{14}$$

Observe that $P(A \cap B \neq \emptyset)$ is negligibly small. In practice, it requires bit size of t_i be equal or greater then 2^{80} to prevent precomputation attacks.

5.3 Hash length extension attack

The vulnerability affects hash functions with the Merkle-Damgård structure that are computed over data concatenated with a string suffix known to the adversary. The hash length extension attack exploits the weakness of Merkle-Damgård construction, following from the fact that a single block of hash input is processed at a time while keeping the hash function's internal state exposed in memory. It implies predictability of number of iterations I within hash function with ability to track and modify hash function state. In our case, length extension attack allows malicious B to compute $v_i = hash(D \mid t_i)$ without knowing D if the underlying hash function is derived from Merkle-Damgård's construction. Malicious Bob should know only the bit length |D| of the bit string D and the block length H of the Merkle-Damgård hash function employed. Note that $|D|$ should be multiple of H and H is the power of two. If the data length |D| is not divided by H, then $|D|$ is being padded by MD-compliant padding according to the given cryptographic hash function standard. The simplified example of hash padding algorithm provided Figure 5. Then the padded data is being split into blocks of length H and processed through a number of iterations $I=|D|/H$ which is also equal number of H-length blocks fit into |D|.

Algorithm 5: HashMDPadding

```
Input  : D, H
Output: D_PAD
// Check length of the data D is multiple of block length
   H of utilized hash function
1 if H ∤ |D| then
   // If H don't divide |D| then complete D by
      concatenating of MD-compliant padding PAD_MD
2  D_PAD := D|PAD_MD return D_PAD
3 end
4 return D
```

Figure 5: MD padding

We can view Merkle-Damgård construction as a finite automaton with memory, where memory is the current state of the hash function $State_i$. The state is updated at the end of each iteration of Merkle-Damgård's structure with function *UpdateHashState*. The next state $State_{i+1}$ is calculated from the current $State_i$ and block D_i of input data D for $i \in [1, I]$. The last state $State_I$ calculated within is the final result of hash function [2]. The number of iterations depends on the block length of the hash function used. Malicious B can determine the number of blocks I by knowing $|D|$ and H. If $|D|$ is not a multiple of H, then the adversary fills additional bits with the MD-compliant padding specified by the hash function's standard to extend the message length $|D|$ to length |D'|, such as H divide $|D'|$. At this point, B has everything in place to perform the attack. The algorithm for padding the data before hashing is shown below.

Then computation of *hash(D)* with Merkle-Damgård construction performed block by block of input data *D*. The example of padding function provided in Figure 6.

Algorithm 6: MerkleDamgardHash

```
Input  : D, H
Output: hash(D)
// Pad D to ensure its length |D| is divisible by hash
   block length H
D := HashMDPadding(D)
// Calculate number of blocks
I ← |D| / H
// Iterate over blocks of data and update the hash state
for i ← 1 to I − 1 do
    State_{i+1} = UpdateHashState(D_i, State_i)
end
// The last computed state of hash considered as the result
   value of the hash function
hash(D) ← State_I
return hash(D)
```

Figure 6: Merkle-Damgard hash algorithm

Let us apply the definition of the hash function to perform the hash length extension attack. We modified of *MerkleDamgardHash* to start calculation not from initial state but for the given state. The pseudo code of modified hash function shown on Figure 7 which exploits the vulnerability described in Figure 6.

As the algorithm in Figure 7 shows, Bob can fabricate the proof v_i to deceive Alice into paying for the storage even if the data *D* has been lost.

Algorithm 7: HashLengthExtensionAttack

```
Input  : l, hash(D), H, a_i
Output: h_i
// Forge random string D' with length l equal to the length
   of original data
D' ← {0,1}^l
// Ensure fake string is padded and it's divisible by length
   of hash block H
D' := HashMDPadding(D')
// Pad a_i with MD-compliant padding to ensure it's multiple
   of hash block length H
a_i := HashMDPadding(a_i)
D' := D'|a_i
// Calculate the number of iterations over suffix a_i needed
   to finish computation of hash(D|a_i)
I ← |a_i|
// Put hash(D) as a start state
State_1 = hash(D)
// Finish computation of hash(D|a_i) without original D but
   starting from state of hash(D) and iterate over
   remaining suffix a_i
for i ← 1 to I do
    State_{i+1} = UpdateHashState(D', State_i)
end
// Finish computation of v_i = hash(D|a_i) without original D
   but starting from state of hash(D) and iterating
   according to Merkle-Damgård over remaining suffix a_i
v_i ← State_I
return v_i
```

Figure 7: Hash length extension attack

The scenario of the attack on the protocol which exploits hash length extension vulnerability shown on Figure 8.

Algorithm 8: VulnerabilityScenario

```
Input  : hash, k
// Communication:  B receives data D from A
// Precompute parameters for length extension attack
// Compute and remember hash(D)
h ← hash(D) // Compute and remember length of data D
l ← |D|
// Run challenges
for i ← 1 to r do
    // Malicious B decided to perform hash length extension
       attack on round k
    if i = k then
        // B discards data D
        // Forge calculation of t_i = hash(D|a_i) without D
        v_i ← HashLengthExtensionAttack(l, hash(D), H, a_i)
            // Communication:  B sends forged challenge
            response v_i to A
        // B succeeded in cheating
    end
end
```

Figure 8: Scenario of vulnerability exploitation

The most straightforward method to counter this attack is to utilize XOR instead of concatenation when producing proofs, leveraging the formula $v_i = hash(D \oplus t_i)$ for hash functions based on the Merkle-Damgård framework, such as SHA-2. This approach mitigates risks linked to length extension attacks. If the use of SHA-2 hashes is required, truncated versions such as SHA-384 and SHA-224 with truncated output state to withstand hash extension attacks [12]. Alternatively, one can employ hash functions based on the sponge functions, such as SHA-3 and Keccak. Such hash functions use permutations and sponge function rate adjustment to obfuscate the internal state on absorption phase and on each update of hash state [4]. This prevents the internal state from being explicitly exposed in memory.

Unlike Merkle-Damgård constructions, sponge-based schemes can be adjusted to produce a variable output length which prevents hash state length manipulations further. Consecutively, an attacker cannot reuse internal state to extend the hash and compute $v_i = hash(D \mid t_i)$ without knowing the full original input. We can observe that the security of the proposed protocol directly depends on the security assumptions of the hash function used to produce a proof v_i. In most of the analyzed

works, the security of the underlying hash functions is overlooked, while the usage of SHA-256 is widely encouraged.

Conclusions

We have presented a verifiable data retrieval protocol with practically deterministic guarantees of retrievability backed by reliability of cryptographic hash function. The PDP protocol applicable in both cloud and blockchain native scenarios. Our approach leverages hash-based mechanisms to improve communication efficiency and security. By analyzing adversarial models, we identified potential attack vectors and introduced mitigation to maintain high security. The protocol's advantages include the ability to construct fixed-size retriviability proofs, which is particularly effective when the amount of information stored is large. This feature provides efficiency and scalability. However, the protocol's main drawback is the absence of public verifiability and a fixed number of rounds which are in common with existing protocols. These limitations will be addressed in the following works.

References

[1] Golle, Philippe, Stanislaw Jarecki, and Ilya Mironov. 2002. "Cryptographic Primitives Enforcing Communication and Storage Complexity." In *Financial Cryptography, 6th International Conference, FC 2002, Southampton, Bermuda, March 11-14, 2002, Revised Papers*. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer. URL: https://doi.org/10.1007/3-540-36504-4_9.

[2] Coron, Jean-Sébastien, Yevgeniy Dodis, Cécile Malinaud, and Prashant Puniya. 2005. "Merkle-Damgård Revisited: How to Construct a Hash Function." In *Advances in Cryptology – Crypto 2005*, edited by Victor Shoup, 3621:430–48. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer. URL: https://doi.org/10.1007/11535218_26.

[3] Ateniese, Giuseppe, Randal Burns, Reza Curtmola, Joseph Herring, Lea Kissner, Zachary Peterson, and Dawn Song. 2007. "Provable Data Possession at Untrusted Stores." In *Proceedings of the 14th Acm Conference on Computer and Communications Security*. New York, NY, USA: ACM. URL: https://archiv.infsec.ethz.ch/education/fs08/secsem/ateniese-ccs07.pdf

[4] Bertoni, Guido, Joan Daemen, Michaël Peeters, and Gilles Van Assche. 2007. "Sponge Functions." STMicroelectronics; NXP Semiconductors. URL: https://keccak.team/files/SpongeFunctions.pdf

[5] Juels, Ari, and Burton S. Kaliski Jr. 2007. "PORs: Proofs of Retrievability for Large Files." In *Proceedings of the 14th Acm Conference on Computer and Communications Security (CCS 2007)*. New York, NY, USA: ACM. URL: https://doi.org/10.1145/1315245.1315317

[6] Ateniese, Giuseppe, Roberto Di Pietro, Luigi V. Mancini, and Gene Tsudik. 2008. "Scalable and Efficient Provable Data Possession." *SecureComm*. URL: https://www.researchgate.net/publication/220333081_Scalable_and_Efficient_Provable_Data_Possession

[7] Shacham, Hovav, and Brent Waters. 2008. "Compact Proofs of Retrievability." In *Advances in Cryptology – Asiacrypt 2008*, edited by Josef Pieprzyk, 5350:90–107. Lecture Notes in Computer Science. Springer. URL: https://doi.org/10.1007/978-3-540-89255-7_7

[8] Xu, Jia, Ee-Chien Chang, and Jianying Zhou. 2012. "Towards Efficient Provable Data Possession in Cloud Storage." URL: https://www.i2r.a-star.edu.sg/

[9] Han, Shuai, Shengli Liu, Kefei Chen, and Dawu Gu. 2013. "Proofs of Data Possession and Retrievability Based on MRD Codes." URL: https://eprint.iacr.org/2013/789.pdf

[10] Benet, Juan. 2014. "IPFS - Content Addressed, Versioned, P2p File System." URL: https://doi.org/10.48550/arXiv.1407.356

[11] Kaaniche, Nesrine, Ethmane El Moustaine, and Maryline Laurent. 2014. "A Novel Zero-Knowledge Scheme for Proof of Data Possession in Cloud Storage Applications." URL: https://www-public.telecom-

sudparis.eu/~lauren_m/articles/2014-ZeroKnowPDPScheme.pdf

[12] National Institute of Standards and Technology. 2015. "Secure Hash Standard (Shs)." FIPS PUB 180-4 FIPS PUB 180-4. Gaithersburg, MD: Information Technology Laboratory, National Institute of Standards; Technology. URL: https://doi.org/10.6028/NIST.FIPS.180-4

[13] Niaz, Muhammad Saqib, and Gunter Saake. 2015. "Merkle Hash Tree Based Techniques for Data Integrity of Outsourced Data." In *Proceedings of the 10th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2015)*. Magdeburg, Germany: CEUR-WS.org. URL: https://ceur-ws.org/Vol-1366/paper13.pdf

[14] Potocol Labs. 2017. "Filecoin: A Decentralized Storage Network." URL: https://filecoin.io/filecoin.pdf

[15] Azizi, Yassine, Mostafa Azizi, and Mohamed Elboukhari. 2022. "Log Data Integrity Solution Based on Blockchain Technology and Ipfs." URL: https://doi.org/10.3991/ijim.v16i15.31713

[16] Walker, Ieuan, Chaminda Hewage, and Ambikesh Jayal. 2022. "Provable Data Possession (PDP) and Proofs of Retrievability (POR) of Current Big*International Journal of Interactive Mobile Technologies (iJIM)* 16 (15): 4–15. User Data." *SN Computer Science* 3 (83). URL: https://doi.org/10.1007/s42979-021-00968-z

[17] Dumas, Jean-Guillaume, Aude Maignan, Clément Pernet, and Daniel S. Roche. 2023. "VESPo: Verified Evaluation of Secret Polynomials (with Application to Dynamic Proofs of Retrievability)." *Proceedings on Privacy Enhancing Technologies* 2023 (3): 354–74. URL: https://doi.org/10.56553/popets-2023-0085

[18] Nalina, V., S. Navaneeth, Rahul Anil Nayak, and Nidhi Prakash. 2024. "Decentralized File Storage Platform Using Ipfs and Blockchain." In *Proceedings of the 2024 International Conference on Emerging Technologies in Computer Science for Interdisciplinary Applications (Icetcs)*. Bengaluru, India: IEEE. URL: https://doi.org/10.1109/ICETCS61022.2024.10543705

[19] Yang, Fan, Yi Sun, Qi Gao, and Xingyuan Chen. 2024. "EDI-C: Reputation-Model-Based Collaborative Audit Scheme for Edge Data Integrity." *Electronics* 13 (1): 75. URL: https://doi.org/10.3390/electronics13010075

[20] Hall-Andersen, Mathias, Mark Simkin, and Benedikt Wagner. 2025. "Foundations of Data Availability Sampling." *IACR Communications in Cryptology* 1 (4). URL: https://doi.org/10.62056/a09qudhdj