StealthGuard: Proofs of Retrievability with Hidden Watchdogs

Monir Azraoui, Kaoutar Elkhiyaoui, Refik Molva, and Melek Önen*

EURECOM, Sophia Antipolis, France
{azraoui, elkhiyao, molva, onen}@eurecom.fr

Abstract. This paper presents **StealthGuard**, an efficient and provably secure proof of retrievabillity (POR) scheme. **StealthGuard** makes use of a privacy-preserving word search (WS) algorithm to search, as part of a POR query, for randomly-valued blocks called watchdogs that are inserted in the file before outsourcing. Thanks to the privacy-preserving features of the WS, neither the cloud provider nor a third party intruder can guess which watchdog is queried in each POR query. Similarly, the responses to POR queries are also obfuscated. Hence to answer correctly to every new set of POR queries, the cloud provider has to retain the file in its entirety. **StealthGuard** stands out from the earlier sentinel-based POR scheme proposed by Juels and Kaliski (JK), due to the use of WS and the support for an unlimited number of queries by **StealthGuard**. The paper also presents a formal security analysis of the protocol.

Keywords: Cloud storage, Proofs of Retrievability, Privacy-preserving word search

1 Introduction

Nowadays outsourcing, that is, delegating one's computing to external parties, is a well established trend in cloud computing. Along with unprecedented advantages such as lower cost of ownership, adaptivity, and increased capacity, outsourcing also raises new security and privacy concerns in that critical data processing and storage operations are performed remotely by potentially untrusted parties. In this paper we focus on data retrievability, a security requirement akin to outsourced data storage services like Dropbox [1] and Amazon Simple Storage Service [2]. Data retrievability provides the customer of a storage service with the assurance that a data segment is actually present in the remote storage. Data retrievability is a new form of integrity requirement in that the customer of the storage or the data owner does not need to keep or get a copy of the data segment in order to get the assurance of retrievability thereof. A cryptographic building block called Proof of Retrievability (POR) was first developed by Juels and Kaliski [13] (JK) to meet this requirement. In the definition of [13], a successful execution of the POR scheme assures a verifier that it can retrieve F in its entirety. Classical

* Authors are listed in alphabetical order.
[1] Dropbox - https://www.dropbox.com/
[2] Amazon Simple Storage Service - http://aws.amazon.com/fr/s3/

integrity techniques such as transferring F with some integrity check value are not practical since they incur very high communication or computational costs that are linear with the size of F. POR schemes aim at much lower cost both in terms of communications and processing by avoiding transmission or handling of F in its entirety. To that effect, POR schemes require the prover to perform some operations on some randomly selected parts of F and the verifier is able to check the result returned by the prover with the knowledge of very brief reference about the data like a secret key. Most POR schemes thus are probabilistic and their performance is measured in the trade-off between the bandwidth and processing overhead and the rate of retrievability assurance.

In this paper we develop **StealthGuard**, a new POR scheme that achieves good retrievability assurance with acceptable costs. The main idea behind the new scheme is a combination of a privacy-preserving word search (WS) algorithm suited to large datastores with the insertion in data segments of randomly generated short bit sequences called **watchdogs**. In **StealthGuard**, the user inserts these watchdogs in randomly chosen locations of the file F and stores the resulting file in the cloud. In order to check the retrievability of F the user issues lookup queries for selected values of watchdogs using the WS scheme. The user decrypts the WS replies from the cloud server in order to get the proof of retrievability for each segment targeted by the WS queries. Each positive result is the proof of presence for the corresponding data segment. Thanks to the features of the WS, neither the cloud server nor a third party intruder can guess which watchdogs are targeted by each WS query or response.

Even though there is an analogy between the watchdogs used in **StealthGuard** and the sentinels akin to the JK scheme [13], there is a major difference between the two schemes due to the use of WS by **StealthGuard**: the number of POR queries that can be issued in **StealthGuard** without requiring any update of the watchdogs is unbounded whereas in the JK scheme a given set of sentinels can be used for a finite number of POR queries only. **StealthGuard** only requires the transfer of some additional data that is a small percentage of F in size and a good POR rate can be achieved by only processing a fraction of F. In addition to the description of our proposal, we give a new security model that enhances existing security definitions of POR schemes [13, 17]. We state a generic definition of the soundness property that applies to any POR scheme.

Contributions. To summarize, this paper offers two main contributions:

− We present **StealthGuard**, a new POR scheme based on the insertion of watchdogs that requires a light file preprocessing and on a privacy-preserving WS that allows a user to issue an unbounded number of POR queries. Besides, the user is stateless since it only needs to keep a secret key to be able to run the POR protocol.

− We propose a new security model which improves existing security definitions [13, 17]. We also provide a formal proof of our proposal under this new security model.

The rest of the paper is organized as follows. Section 2 defines the entities and the algorithms involved in a POR scheme. Section 3 describes the adversary models that are considered in this paper. Section 4 provides an overview of **StealthGuard** and Section 5 gives details of the protocol. Section 6 analyses its security properties. Section 7 evaluates its security and its efficiency. We review the state of the art in Section 8.

2 Background

Before presenting the formal definition of PORs and the related security definitions, we introduce the entities that we will refer to in the remainder of this paper.

2.1 Entities

A POR scheme comprises the following entities:

- Client $\mathcal{C}$: It possesses a set of files $\mathcal{F}$ that it outsources to the cloud server $\mathcal{S}$. Without loss of generality, we assume that each file $F \in \mathcal{F}$ is composed of n splits $\{S_1, S_2, ..., S_n\}$ of equal size L bits. In practice, if the size of F is not a multiple of L, then padding bits will be added to F. We also suppose that each split S_i comprises m blocks of l bits $\{b_{i,1}, b_{i,2}, ..., b_{i,m}\}$, i.e., $L = m \cdot l$.
- Cloud Server $\mathcal{S}$ (a potentially malicious prover): For each file $F \in \mathcal{F}$, the cloud server $\mathcal{S}$ stores an "enlarged" verifiable version $\hat{F}$ of that file, that enables it to prove to a verifier $\mathcal{V}$ that the client $\mathcal{C}$ can still retrieve its original file F.
- Verifier $\mathcal{V}$: It is an entity which via an interactive protocol can check whether the cloud server $\mathcal{S}$ (i.e., the prover) is still storing some file $F \in \mathcal{F}$ or not. The verifier can be either the client itself or any other *authorized* entity, such as an auditor.

2.2 POR

A POR scheme consists of five polynomial-time algorithms (cf. [13, 17]):

- $\mathsf{KeyGen}(1^\tau) \rightarrow K$: This probabilistic key generation algorithm is executed by client $\mathcal{C}$. It takes as input a security parameter τ, and outputs a *secret key* K for $\mathcal{C}$.
- $\mathsf{Encode}(K, F) \rightarrow (\mathsf{fid}, \hat{F})$: It takes the key K and the file $F = \{S_1, S_2, ..., S_n\}$ as inputs, and returns the file $\hat{F} = \{\hat{S}_1, \hat{S}_2, ..., \hat{S}_n\}$ and F's *unique* identifier fid. Cloud server $\mathcal{S}$ is required to store $\hat{F}$ together with fid. $\hat{F}$ is obtained by first applying to F an *error-correcting code* (ECC) which allows client $\mathcal{C}$ to recover the file from minor corruptions that may go undetected by the POR scheme, and further by adding some *verifiable redundancy* that enables client $\mathcal{C}$ to check whether cloud server $\mathcal{S}$ still stores a *retrievable* version of F or not.
  Note that the Encode algorithm is invertible. Namely, there exists an algorithm Decode that allows the client $\mathcal{C}$ to recover its original file F from the file $\hat{F}$.
- $\mathsf{Challenge}(K, \mathsf{fid}) \rightarrow \mathsf{chal}$: The verifier $\mathcal{V}$ calls this *probabilistic* algorithm to generate a challenge chal for an execution of the POR protocol for some file F. This algorithm takes as inputs the secret key K and the file identifier fid, and returns the challenge chal that will be sent to cloud server $\mathcal{S}$.
- $\mathsf{ProofGen}(\mathsf{fid}, \mathsf{chal}) \rightarrow \mathcal{P}$: On receiving the challenge chal and the file identifier fid, cloud server $\mathcal{S}$ executes ProofGen to generate a proof of retrievability $\mathcal{P}$ for the file $\hat{F}$ whose identifier is fid. The proof $\mathcal{P}$ is then transmitted to verifier $\mathcal{V}$.
- $\mathsf{ProofVerif}(K, \mathsf{fid}, \mathsf{chal}, \mathcal{P}) \rightarrow b \in \{0, 1\}$: Verifier $\mathcal{V}$ runs this algorithm to check the validity of the proofs of retrievability sent by cloud server $\mathcal{S}$. On input of the key K, the file identifier fid, the challenge chal, and the proof $\mathcal{P}$, the ProofVerif algorithm outputs bit $b = 1$ if the proof $\mathcal{P}$ is a valid proof, and $b = 0$ otherwise.

3 Adversary models

A POR scheme should ensure that if cloud server $\mathcal{S}$ is storing the outsourced files, then the ProofVerif algorithm should always output 1, meaning that ProofVerif does not yield any false negatives. This corresponds to the *completeness* property of the POR scheme. PORs should also guarantee that if $\mathcal{S}$ provides a number (to be determined) of valid proofs of retrievability for some file F, then verifier $\mathcal{V}$ can deduce that server $\mathcal{S}$ is storing a retrievable version of F. This matches the *soundness* property of POR. These two properties are formally defined in the following sections.

3.1 Completeness

If cloud server $\mathcal{S}$ and verifier $\mathcal{V}$ are both honest, then on input of a challenge chal and some file identifier fid sent by verifier $\mathcal{V}$, the ProofGen algorithm generates a proof of retrievability $\mathcal{P}$ that will be accepted by verifier $\mathcal{V}$ with probability 1.

Definition 1 (Completeness). *A POR scheme is* complete *if for any honest pair of cloud server $\mathcal{S}$ and verifier $\mathcal{V}$, and for any challenge* $\mathsf{chal} \leftarrow \mathsf{Challenge}(K, \mathsf{fid})$*:*

$$\mathsf{Pr}(\mathsf{ProofVerif}(K, \mathsf{fid}, \mathsf{chal}, \mathcal{P}) \rightarrow 1 \mid \mathcal{P} \leftarrow \mathsf{ProofGen}(\mathsf{fid}, \mathsf{chal})) = 1$$

3.2 Soundness

A proof of retrievability is deemed sound, if for any malicious cloud server $\mathcal{S}$, the only way to convince verifier $\mathcal{V}$ that it is storing a file F is by actually keeping a retrievable version of that file. This implies that any cloud server $\mathcal{S}$ that generates (a polynomial number of) valid proofs of retrievability for some file F, must possess a version of that file that can be used later by client $\mathcal{C}$ to recover F. To reflect the intuition behind this definition of soundness, Juels and Kaliski [13] suggested the use of a file extractor algorithm $\mathcal{E}$ that is able to retrieve the file F by interacting with cloud server $\mathcal{S}$ using the *sound* POR protocol. Along these lines, we present a new and a more generic soundness definition that refines the formalization of Shacham and Waters [17] which in turn builds upon the work of Juels and Kaliski [13]. Although the definition of Shacham and Waters [17] captures the soundness of POR schemes that empower the verifier with unlimited (i.e. exponential) number of "possible" POR challenges [3, 17, 23], it does not define properly the soundness of POR schemes with limited number of "possible" POR challenges such as in [13, 19] and in **StealthGuard**[3]. We recall that the formalization in [17] considers a POR to be sound, if a file can be recovered whenever the cloud server generates a valid POR response for that file with a *non-negligible* probability. While this definition is accurate in the case where the verifier is endowed with unlimited number of POR challenges, it cannot be employed to evaluate the soundness of the mechanisms introduced in [13, 19] or the solution we will present in this paper. For example, if we take the POR scheme in [19] and if we consider a scenario where the

[3] Note that having a bounded number of POR challenges does not negate the fact that the verifier can perform unlimited number of POR queries with these same challenges, cf. [19].

cloud server corrupts randomly half of the outsourced files, then the cloud server will be able to correctly answer half (which is non-negligible) of the POR challenges that the verifier issues, yet the files are irretrievable. This implies that this POR mechanism is not secure in the model of Shacham and Waters [17], still it is arguably sound.

The discrepancy between the soundness definition in [17] and the work of [13, 19] springs from the fact that in practice to check whether a file is correctly stored at the cloud server, the verifier issues a polynomial number of POR queries to which the server has to respond correctly; otherwise, the verifier detects a corruption attack (the corruption attack could either be malicious or accidental) and flags the server as malicious. This is actually what the PORs of [13, 19] and **StealthGuard** aim to capture. In order to remedy this shortcoming, we propose augmenting the definition of Shacham and Waters [17] (as will be shown in Algorithm 2) with an additional parameter γ that quantifies the number of POR queries that verifier should issue to either be sure that a file is retrievable or to detect a corruption attack on that file.

Now in accordance with [17], we first formalize *soundness* using a game that describes the capabilities of an adversary $\mathcal{A}$ (i.e., malicious cloud server) which can deviate arbitrarily from the POR protocol, and then we define the extractor algorithm $\mathcal{E}$.

To formally capture the capabilities of adversary $\mathcal{A}$, we assume that it has access to the following oracles:

- $\mathcal{O}_{\mathsf{Encode}}$: This oracle takes as inputs a file F and the client's key K, and returns a file identifier fid and a verifiable version $\hat{F}$ of F that will be outsourced to $\mathcal{A}$. Note that adversary $\mathcal{A}$ can corrupt the outsourced file $\hat{F}$ either by modifying or deleting $\hat{F}$'s blocks.
- $\mathcal{O}_{\mathsf{Challenge}}$: On input of a file identifier fid and client's key K, the oracle $\mathcal{O}_{\mathsf{Challenge}}$ returns a POR challenge chal to adversary $\mathcal{A}$.
- $\mathcal{O}_{\mathsf{Verify}}$: When queried with client's key K, a file identifier fid, a challenge chal and a proof of retrievability $\mathcal{P}$, the oracle $\mathcal{O}_{\mathsf{Verify}}$ returns bit b such that: $b = 1$ if $\mathcal{P}$ is a valid proof of retrievability, and $b = 0$ otherwise.

Adversary $\mathcal{A}$ accesses the aforementioned oracles in two phases: a learning phase and a challenge phase. In the learning phase, adversary $\mathcal{A}$ can call oracles $\mathcal{O}_{\mathsf{Encode}}$, $\mathcal{O}_{\mathsf{Challenge}}$, and $\mathcal{O}_{\mathsf{Verify}}$ for a polynomial number of times in any interleaved order as depicted in Algorithm 1. Then, at the end of the learning phase, the adversary $\mathcal{A}$ specifies a file identifier fid^* that was already output by oracle $\mathcal{O}_{\mathsf{Encode}}$.

We note that the goal of adversary $\mathcal{A}$ in the challenge phase (cf. Algorithm 2) is to generate γ valid proofs of retrievability $\mathcal{P}_{\backslash}{}^*$ for file F^* associated with file identifier fid^*. To this end, adversary $\mathcal{A}$ first calls the oracle $\mathcal{O}_{\mathsf{Challenge}}$ that supplies $\mathcal{A}$ with γ challenges chal_i^*, then it responds to these challenges by outputting γ proofs $\mathcal{P}_i^*$. Now, on input of client's key K, file identifier fid^*challenges chal_i^* and proofs $\mathcal{P}_i^*$, oracle $\mathcal{O}_{\mathsf{Verify}}$ outputs γ bits b_i^*. Adversary $\mathcal{A}$ is said to be successful if $b^* = \bigwedge_{i=1}^{\gamma} b_i^* = 1$. That is, if $\mathcal{A}$ is able to generate γ proofs of retrievability $\mathcal{P}^*$ for file F^* that are accepted by oracle $\mathcal{O}_{\mathsf{Verify}}$.

Given the game described above and in line with [13, 17], we formalize the soundness of POR schemes through the definition of an extractor algorithm $\mathcal{E}$ that uses adversary $\mathcal{A}$ to recover/retrieve the file F^* by processing as follows:

– $\mathcal{E}$ takes as inputs the client's key K and the file identifier fid^*;
– $\mathcal{E}$ is allowed to initiate a polynomial number of POR executions with adversary $\mathcal{A}$ for the file F^*;
– $\mathcal{E}$ is also allowed to rewind adversary $\mathcal{A}$. This suggests in particular that extractor $\mathcal{E}$ can execute the challenge phase of the soundness game a polynomial number of times, while the state of adversary $\mathcal{A}$ remains unchanged.

Intuitively, a POR scheme is sound, if for any adversary $\mathcal{A}$ that wins the soundness game with a non-negligible probability δ, there exists an extractor algorithm $\mathcal{E}$ that succeeds in retrieving the challenge file F^* with an overwhelming probability. A probability is overwhelming if it is equal to $1 - \varepsilon$, where ε is negligible.

Algorithm 1: Learning phase of the soundness game

// $\mathcal{A}$ executes the following in any interleaved
// order for a polynomial number of times
$(\mathsf{fid}, \hat{F}) \leftarrow \mathcal{O}_{\mathsf{Encode}}(F, K)$;
$\mathsf{chal} \leftarrow \mathcal{O}_{\mathsf{Challenge}}(K, \mathsf{fid})$;
$\mathcal{P} \leftarrow \mathcal{A}$;
$b \leftarrow \mathcal{O}_{\mathsf{Verify}}(K, \mathsf{fid}, \mathsf{chal}, \mathcal{P})$;
// $\mathcal{A}$ outputs a file identifier fid^*
$\mathsf{fid}^* \leftarrow \mathcal{A}$;

Algorithm 2: Challenge phase of the soundness game

for $i = 1$ **to** γ **do**
  $\mathsf{chal}_i^* \leftarrow \mathcal{O}_{\mathsf{Challenge}}(K, \mathsf{fid}^*)$;
  $\mathcal{P}_i^* \leftarrow \mathcal{A}$;
  $b_i^* \leftarrow$
  $\mathcal{O}_{\mathsf{Verify}}(K, \mathsf{fid}_i^*, \mathsf{chal}_i^*, \mathcal{P}_i^*)$;
end
$b^* = \bigwedge_{i=1}^{\gamma} b_i^*$

Definition 2 (Soundness). *A POR scheme is said to be* (δ, γ)-sound*, if for every adversary $\mathcal{A}$ that provides γ valid proofs of retrievability in a row (i.e., succeeds in the soundness game described above) with a non-negligible probability δ, there exists an extractor algorithm $\mathcal{E}$ such that:*

$$Pr(\mathcal{E}(K, \mathsf{fid}^*) \rightarrow F^* \mid \mathcal{E}(K, \mathsf{fid}^*) \overset{\mathsf{interact}}{\longleftrightarrow} \mathcal{A}) \geq 1 - \varepsilon$$

Where ε is a negligible function in the security parameter τ.

The definition above could be interpreted as follows: if verifier $\mathcal{V}$ issues a sufficient number of queries ($\geq \gamma$) to which cloud server $\mathcal{S}$ responds correctly, then $\mathcal{V}$ can ascertain that $\mathcal{S}$ is still storing a retrievable version of file F^* with high probability. It should be noted that while γ characterizes the number of *valid* proofs of retrievability that $\mathcal{E}$ has to receive (successfully or in a row) to assert that file F^* is still retrievable, δ quantifies the number of operations that the extractor $\mathcal{E}$ has to execute and the amount of data that it has to download to first declare F^* as retrievable and then to extract it. Actually, the computation and the communication complexity of extractor $\mathcal{E}$ will be of order $O(\frac{\gamma}{\delta})$.

4 Overview

4.1 Idea

In **StealthGuard**, client $\mathcal{C}$ first injects some pseudo-randomly generated *watchdogs* into random positions in the encrypted data. Once data is outsourced, $\mathcal{C}$ launches lookup

queries to check whether the watchdogs are stored as expected by the cloud. By relying on a privacy-preserving word search (WS), we ensure that neither the cloud server $\mathcal{S}$ nor eavesdropping intruders can discover which watchdog was targeted by search queries. As a result, $\mathcal{C}$ can launch an unbounded number of POR queries (even for the same watchdog) without the need of updating the data with new watchdogs in the future. The responses are also obfuscated thanks to the underlying WS scheme. This ensures that the only case in which $\mathcal{S}$ returns a valid set of responses for the POR scheme is when it stores the entire file and executes the WS algorithm correctly (soundness property).

Besides, as in [13], in order to protect the data from small corruptions, **StealthGuard** applies an ECC that enables the recovery of the corrupted data. Substantial damage to the data is detected via the watchdog search.

4.2 StealthGuard phases

A client $\mathcal{C}$ uploads to the cloud server $\mathcal{S}$ a file F which consists of n splits $\{S_1, ..., S_n\}$. Thereafter a verifier $\mathcal{V}$ checks the retrievability of F using **StealthGuard**.

The protocol is divided into three phases:

- *Setup*: During this phase, client $\mathcal{C}$ performs some transformations over the file and inserts a certain number of watchdogs in each split. The resulting file is sent to cloud server $\mathcal{S}$.
- *WDSearch*: This phase consists in searching for some watchdog w in a privacy-preserving manner. Hence, verifier $\mathcal{V}$ prepares and sends a lookup query for w; the cloud $\mathcal{S}$ in turn processes the relevant split to generate a correct response to the search and returns the output to $\mathcal{V}$.
- *Verification*: Verifier $\mathcal{V}$ checks the validity of the received response and makes the decision about the existence of the watchdog in the outsourced file.
  We note that if $\mathcal{V}$ receives at least γ (γ is a threshold determined in Section 6.2) correct responses from the cloud, then it can for sure decide that F is retrievable. On the other hand, if $\mathcal{V}$ receives one response that is not valid, then it is convinced either the file is corrupted or even lost.

5 StealthGuard

This section details the phases of the protocol. Table 1 sums up the notation used in the description. We also designed a dynamic version of **StealthGuard** that allows efficient POR even when data is updated. Due to space limitations, we only present in Section 5.4 an overview of dynamic **StealthGuard**.

5.1 Setup

This phase prepares a verifiable version $\hat{F}$ of file $F = \{S_1, S_2, ..., S_n\}$. Client $\mathcal{C}$ first runs the KeyGen algorithm to generate the master secret key K. It derives $n + 3$ additional keys, used for further operations in the protocol: $K_{enc} = H_{enc}(K)$, $K_{wdog} = H_{wdog}(K)$, $K_{permF} = H_{permF}(K)$ and for $i \in [\![1, n]\!]$, $K_{permS,i} = H_{permS}(K, i)$

Index	Description
n	number of splits S_i in F
m	number of blocks in a split S_i
D	number of blocks in an encoded split $\tilde{S}_i$
v	number of watchdogs in one split
C	number of blocks in a split $\hat{S}_i$ with watchdogs
i	index of a split $\in [\![1, n]\!]$
k	index of a block in $\hat{S}_i \in [\![1, C]\!]$
j	index of a watchdog $\in [\![1, v]\!]$
l	size of a block
p	index of a block in $\tilde{F} \in [\![1, n \cdot D]\!]$
q	number of cloud's matrices
κ	index of a cloud's matrix $\in [\![1, q]\!]$
(s, t)	size of cloud's matrices
(x, y)	coordinates in a cloud's matrix $\in [\![1, s]\!] \times [\![1, t]\!]$

Table 1: Notation used in the description of StealthGuard

with H_{enc}, H_{wdog}, H_{permF} and H_{permS} being four cryptographic hash functions. K is the single information stored at the client.

Once all keying material is generated, $\mathcal{C}$ runs the Encode algorithm which first generates a pseudo-random and unique file identifier fid for file F, and then processes F as depicted in Figure 1.

1. **Error correcting**: The error-correcting code (ECC) assures the protection of the file against small corruptions. This step applies to each split S_i an ECC that operates over l-bit symbols. It uses an efficient $[m+d-1, m, d]$-ECC, such as Reed-Solomon codes [16], that has the ability to correct up to $\frac{d}{2}$ errors[4]. Each split is expanded with $d-1$ blocks of redundancy. Thus, the new splits are made of $D = m+d-1$ blocks.
2. **File block permutation**: **StealthGuard** applies a pseudo-random permutation to permute all the blocks in the file. This operation conceals the dependencies between the original data blocks and the corresponding redundancy blocks within a split. Without this permutation, the corresponding redundancy blocks are just appended to this split. An attacker could for instance delete all the redundancy blocks and a single data block from this split and thus render the file irretrievable. Such an attack would not easily be detected since the malicious server could still be able to respond with valid proofs to a given POR query targeting other splits in the file. The permutation prevents this attack since data blocks and redundancy blocks are mixed up among all splits. Let $\Pi_F : \{0,1\}^\tau \times [\![1, n \cdot D]\!] \rightarrow [\![1, n \cdot D]\!]$ be a pseudo-random permutation: for each $p \in [\![1, n \cdot D]\!]$, the block at current position p will be at position $\Pi_F(K_{permF}, p)$ in the permuted file that we denote $\tilde{F}$. $\tilde{F}$ is then divided into n splits $\{\tilde{S}_1, \tilde{S}_2, ..., \tilde{S}_n\}$ of equal size D.
3. **Encryption**: **StealthGuard** uses a semantically secure encryption E that operates over l-bit blocks[5] to encrypt the data. An encryption scheme like AES in counter mode [10] can be used. The encryption E is applied to each block of $\tilde{F}$ using K_{enc}.

[4] d is even

[5] Practically, l will be 128 or 256 bits.

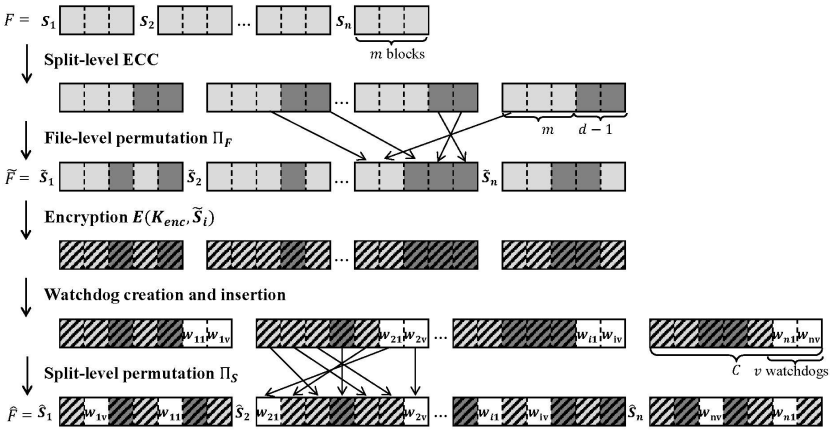

Fig. 1: Setup phase in StealthGuard

4. **Watchdog creation**: For each encrypted split, v l-bit watchdogs are generated using a pseudo-random function $\Phi : \{0,1\}^\tau \times [\![1,n]\!] \times [\![1,v]\!] \times \{0,1\}^* \rightarrow \{0,1\}^l$. Hence, for $j \in [\![1,v]\!]$, $w_{i,j} = \Phi(K_{wdog}, i, j, \mathsf{fid})$. The use of fid guarantees that two different files belonging to the same client have different watchdogs. Since the watchdogs are pseudo-randomly generated and the blocks in the split are encrypted, a malicious cloud cannot distinguish watchdogs from data blocks.
5. **Watchdog insertion**: The v watchdogs are appended to each split. Let $C = D + v$ be the size of the new splits. A split-level pseudo-random permutation $\Pi_S : \{0,1\}^\tau \times [\![1,C]\!] \rightarrow [\![1,C]\!]$ is then applied to the blocks within the same split in order to randomize the location of the watchdogs: for $i \in [\![1,n]\!]$, the block at current position k will be at position $\Pi_S(K_{permS,i}, k)$ in the permuted split. Note that in practice, the permutation is only applied to the last v blocks: for $k \in [\![D,C]\!]$, this step swaps block at current position k for block at position $\Pi_S(K_{permS,i}, k)$. We denote $\hat{S}_i$, $i \in [\![1,n]\!]$, the permuted split and $\hat{b}_{i,k}$, $k \in [\![1,C]\!]$ its blocks.

These operations yield file $\hat{F}$. The client uploads the splits $\{\hat{S}_i\}_{i=1}^n$ and fid to the cloud.

5.2 WDSearch

Verifier $\mathcal{V}$ wants to check the retrievability of F. Hence, it issues lookup queries for randomly selected watchdog, one watchdog for one split in one query. Cloud server $\mathcal{S}$ processes these queries without knowing what the values of the watchdogs are and where they are located in the splits. We propose WDSearch, a privacy-preserving WS solution derived from PRISM in [6]. Our proposal is a simpler version of PRISM and improves its performance in the particular context of **StealthGuard**. Note that this proposed building block is only an example and any existing privacy-preserving WS mechanism assuring the confidentiality of both the query and the result can be used in **StealthGuard**. PRISM and thus WDSearch are based on Private Information Retrieval (PIR). To process a query, $\mathcal{S}$ constructs q (s,t)-binary matrices such that $s \cdot t = C$. Each element in the matrices is filled with the witness (a very short information) of the corresponding block in the split. Based on the PIR query sent by the verifier, the server

retrieves in the matrices the witnesses corresponding to the requested watchdogs. We insist on the fact that WDSearch is not a PIR solution: the server does not retrieve the watchdog itself but only the witness.

WDSearch consists of two steps:

- **WDQuery**: Verifier $\mathcal{V}$ executes the Challenge algorithm to generate a challenge chal that is transmitted to cloud server $\mathcal{S}$. Challenge takes as input master key K and file identifier fid and it is executed in three phases. In the first phase, Challenge randomly selects a split index i and a watchdog index j ($i \in [\![1, n]\!]$ and $j \in [\![1, v]\!]$), and computes the position $\mathsf{pos_j}$ of the watchdog $w_{i,j}$ in the split $\hat{S}_i$ by applying the permutation performed during the watchdog insertion step: $\mathsf{pos_j} = \Pi_S(K_{permS,i}, D+j)$. Then, Challenge maps the position $\mathsf{pos_j}$ to a unique position (x_j, y_j) in an (s, t)-matrix:

  $$x_j = \lceil \frac{\mathsf{pos_j}}{t} \rceil \qquad\qquad y_j = \mathsf{pos_j} - \lceil \frac{\mathsf{pos_j}}{t} \rceil \times t + t$$

  In the second phase, given (x_j, y_j) and using any efficient PIR algorithm, Challenge computes a PIR query, denoted WitnessQuery, to retrieve the witness (and not the watchdog) at position (x_j, y_j) in the matrix. In the last phase, Challenge generates a random number r (this nonce will be used by the cloud when filling the binary matrices to guarantee freshness), and outputs the challenge $\mathsf{chal} = (\mathsf{WitnessQuery}, r, i)$. Eventually, verifier $\mathcal{V}$ sends the challenge chal and file identifier fid to cloud server $\mathcal{S}$.
- **WDResponse**: Upon receiving the challenge $\mathsf{chal} = (\mathsf{WitnessQuery}, r, i)$ and file identifier fid, cloud server $\mathcal{S}$ runs ProofGen to process the query. The cloud creates q binary matrices of size (s, t). For each block $\hat{b}_{i,k}$ in $\hat{S}_i$, the cloud computes $h_{i,k} = H(\hat{b}_{i,k}, r)$, where $k \in [\![1, C]\!]$. Here, H denotes a cryptographic hash function. The use of r forces the cloud to store the actual data block. Otherwise it could drop the block, only store the hash and respond to the query using that hash. Let $h_{i,k}|_q$ be the first q bits of $h_{i,k}$. For $\kappa \in [\![1, q]\!]$, let $\mathcal{M}_\kappa$ be one of the matrices created by the cloud. It fills the κ^{th} matrix with the κ^{th} bit of $h_{i,k}|_q$ as Algorithm 3 shows. It should be noted that according to the assignment process described in Algorithm 3, the witness at position (x_j, y_j) in $\mathcal{M}_\kappa$ is associated with watchdog $w_{i,j}$: it is the κ^{th} bit of $H(w_{i,j}, r)$.
  Once all the q binary matrices are filled, the cloud processes WitnessQuery by executing a PIR operation that retrieves one bit from each matrix $\mathcal{M}_\kappa$, $\kappa \in [\![1, q]\!]$. We denote $\mathsf{WitnessResponse}_\kappa$ the result of the PIR on matrix $\mathcal{M}_\kappa$. The ProofGen algorithm outputs $\mathcal{P}$, i.e. the proof of retrievability that consists in the set $\mathcal{P} = \{\mathsf{WitnessResponse}_1, ..., \mathsf{WitnessResponse}_q\}$. Cloud server $\mathcal{S}$ sends the proof $\mathcal{P}$ to verifier $\mathcal{V}$.

5.3 Verification

Verifier $\mathcal{V}$ runs ProofVerif to analyze the received proof $\mathcal{P}$. This algorithm takes as input master key K, proof $\mathcal{P}$, split index i, watchdog index j, and file identifier fid. ProofVerif outputs a bit equal to 1 if the proof is valid or 0 otherwise.

Algorithm 3: Filling the cloud matrices

```
// For a given (s, t)-matrix M_κ, a given split Ŝ_i and a given random number r
// k is the index of a block in split Ŝ_i
k = 1;
for x = 1 to s do
    for y = 1 to t do
        M_κ[x, y] ← κ^th bit of H(b̂_{i,k}, r);
        k = k + 1;
    end
end
```

$\mathcal{V}$ processes the q $\mathsf{WitnessResponse}_\kappa$ in order to retrieve the q bits ϵ_κ at position (x_j, y_j) in the matrix $\mathcal{M}_\kappa$, for $\kappa \in [\![1, q]\!]$. Let h denote $\epsilon_1\epsilon_2...\epsilon_q$.

We recall that verifier $\mathcal{V}$ queried watchdog $w_{i,j}$ for split $\hat{S}_i$ and that by having access to the master key K, $\mathcal{V}$ can recompute the value of $w_{i,j} = \Phi(K_{wdog}, i, j, \mathsf{fid})$ and its position in the split $\hat{S}_i$, $\mathsf{pos_j} = \Pi_S(K_{permS,i}, D + j)$. Thereafter, $\mathcal{V}$ computes the hash of the watchdog $h_{i,\mathsf{pos_j}} = H(w_{i,j}, r)$, with the same r chosen during the challenge and considers the q first bits of $h_{i,\mathsf{pos_j}}$. Based on the value of $h = \epsilon_1\epsilon_2...\epsilon_q$ and $h_{i,\mathsf{pos_j}}$, $\mathcal{V}$ checks whether $h = h_{i,\mathsf{pos_j}}|_q$. If it is the case, then $\mathcal{V}$ judges the proof valid and returns 1, otherwise it interprets the invalid proof as the occurrence of an attack and outputs 0.

As mentioned in section 4.2, in order to acknowledge the retrievability of F, verifier $\mathcal{V}$ needs to initiate at least γ POR queries[6] from randomly selected splits in order to either ascertain that F is retrievable or detect a corruption attack: if $\mathcal{V}$ receives γ valid POR responses, then it can conclude that cloud server $\mathcal{S}$ stores a retrievable version of F, otherwise, it concludes that $\mathcal{S}$ has corrupted part of the file.

5.4 Dynamic StealthGuard

The previously described protocol does not consider update operations that the client can perform over its data. Similarly to the work in [2, 8, 9, 11, 15, 18, 19, 21, 22, 24], we propose a scheme that handles these updates. Due to space limitations we present only an idea of how dynamic **StealthGuard** operates. Any update in the data impacts the security of our protocol. For example, if the client modifies the same block several times then the cloud can discover that this particular block is not a watchdog. Therefore, dynamic **StealthGuard** updates the watchdogs in a split each time an update occurs on that split. Besides, the verifier must be ensured that the file stored at the server is actually the latest version. Dynamic **StealthGuard** offers a versioning solution to assure that the cloud always correctly applies the required update operations and that it always stores the latest version of the file. Our proposal uses Counting Bloom Filters [12] and Message Authentication Codes (MAC) [5]. Each time a split is updated, some information regarding the split number and the version number is added into the counting Bloom filter which is authenticated using a MAC that can only be computed by the client and the verifier. Additionally, to guarantee the freshness of the response at each

[6] The value of γ will be determined in Section 6.2.

update query, a new MAC key is generated. This protocol does not imply any additional cost at the verifier except of storing an additional MAC symmetric key.

Another challenging issue is that updating a data block requires to update the corresponding redundancy blocks, resulting in the disclosure to the cloud server of the dependencies between the data blocks and the redundancy blocks. Therefore, the file permutation in the *Setup* phase becomes ineffective. Some techniques are available to conceal these dependencies such as batch updates [19] or oblivious RAM [8]. However, these approaches are expensive in terms of computation and communication costs. Hence, we choose to trade off between POR security and update efficiency by omitting the file permutation.

6 Security Analysis

In this section, we state the security theorems of **StealthGuard**.

6.1 Completeness

Theorem 1. ***StealthGuard*** *is complete.*

Proof. Without loss of generality, we assume that the honest verifier $\mathcal{V}$ runs a POR for a file F. To this end, verifier $\mathcal{V}$ sends a challenge $\mathsf{chal} = (\mathsf{WitnessQuery}, r, i)$ for watchdog $w_{i,j}$, and the file identifier fid of F. Upon receiving challenge chal and file identifier fid, the cloud server generates a proof of retrievability $\mathcal{P}$ for F.

According to **StealthGuard**, the verification of POR consists of first retrieving the first q bits of a hash h_{i,pos_j}, then verifying whether $h_{i,\mathsf{pos}_j}|q$ corresponds to the first q-bits of the hash $H(w_{i,j}, r)$. Since the cloud server $\mathcal{S}$ is honest, then this entails that it stores $w_{i,j}$, and therewith, can always compute $h_{i,\mathsf{pos}_j} = H(w_{i,j}, r)$.

Consequently, $\mathsf{ProofVerif}(K, \mathsf{fid}, \mathsf{chal}, \mathcal{P}) = 1$.

6.2 Soundness

As in Section 5, we assume that each split S_i in a file F is composed of m blocks, and that the Encode algorithm employs a $[D, m, d]$-ECC that corrects up to $\frac{d}{2}$ errors per split (i.e., $D = m + d - 1$). We also assume that at the end of its execution, the Encode algorithm outputs the encoded file $\hat{F}$ which consists of a set of splits $\hat{S}_i$ each comprising $C = (D + v)$ blocks (we recall that v is the number of watchdogs per split).

In the following, we state the main security theorem for **StealthGuard**.

Theorem 2. *Let τ be the security parameter of* **StealthGuard** *and let ρ denote $\frac{d}{2D}$.*

StealthGuard is (δ, γ)-sound in the random oracle model, for any $\delta > \delta_{\mathsf{neg}}$ and $\gamma \geq \gamma_{\mathsf{neg}}$, where

$$\delta_{\mathsf{neg}} = \frac{1}{2^{\tau}}$$

$$\gamma_{\mathsf{neg}} = \lceil \frac{\mathsf{ln}(2)\tau}{\rho_{\mathsf{neg}}} \rceil$$

$$(1 - \frac{\rho}{\rho_{\mathsf{neg}}})^2 \rho_{\mathsf{neg}} = \frac{3\mathsf{ln}(2)\tau}{D} \text{ and } \rho_{\mathsf{neg}} \leq \rho$$

Actually if $\gamma \geq \gamma_{\mathsf{neg}}$, then there exists an extractor $\mathcal{E}$ that recovers a file F with a probability $1-\frac{n}{2^\tau}$, such that n is the number of splits in F, by interacting with an adversary $\mathcal{A}$ against StealthGuard who succeeds in the soundness game with a probability $\delta > \frac{1}{2^\tau}$.

Due to space limitations, a proof sketch of this theorem is provided in our long report [4]. We note that the results derived above can be interpreted as follows: if verifier $\mathcal{V}$ issues $\gamma \geq \gamma_{\mathsf{neg}}$ POR queries for some file F to which the cloud server $\mathcal{S}$ responds correctly, then $\mathcal{V}$ can declare F as retrievable with probability $1 - \frac{n}{2^\tau}$. Also, we recall that a POR execution for a file F in **StealthGuard** consists of fetching (obliviously) a witness of a watchdog from the encoding $\hat{F}$ of that file. Consequently, to ensure a security level of $\frac{1}{2^\tau}$, the client $\mathcal{C}$ must insert at least γ_{neg} watchdogs in F. That is, if file F comprises n splits, then $nv \geq \gamma_{\mathsf{neg}}$ (v is the number of watchdogs per split).

7 Discussion

StealthGuard requires the client to generate $v > \frac{\gamma_{\mathsf{neg}}}{n}$ watchdogs per split where n is the number of splits and γ_{neg} is the threshold of the number of queries that verifier $\mathcal{V}$ should issue to check the retrievability of the outsourced data. As shown in Theorem 2, this threshold does not depend on the size of data (in bytes). Instead, γ_{neg} is defined solely by the security parameter τ, the number $D = m + d - 1$ of data blocks and redundancy block per split and the rate $\rho = \frac{d}{2D}$ of errors that the underlying ECC can correct. Namely, γ_{neg} is inversely proportional to both D and ρ. This means that by increasing the number of blocks D per split or the *correctable* error rate ρ, the number of queries that the client should issue decreases. However, having a large ρ would increase the size of data that client $\mathcal{C}$ has to outsource to cloud server $\mathcal{S}$, which can be inconvenient for the client. Besides, increasing D leads to an increase of the number of blocks $C = s \cdot t$ per split $\hat{S}_i$ which has a direct impact on the communication cost and the computation load *per query* at both the verifier $\mathcal{V}$ and the cloud server $\mathcal{S}$. It follows that when defining the parameters of **StealthGuard**, one should consider the tradeoff between the affordable storage cost and the computation and communication complexity per POR query.

To enhance the computation performances of **StealthGuard**, we suggest to use the **Trapdoor Group Private Information Retrieval** which was proposed in [20] to implement the PIR instance in WDSearch. This PIR enables the verifier in **StealthGuard** to fetch a row from an (s, t) matrix (representing a split) without revealing to the cloud which row the verifier is querying. One important feature of this PIR is that it only involves random number generations, additions and multiplications in $\mathbb{Z}_p$ (where p is a prime of size $|p| = 200$ bits) which are not computationally intensive and could be performed by a lightweight verifier. In addition, we emphasize that PIR in **StealthGuard** is not employed to retrieve a watchdog, but rather to retrieve a q-bit hash of the watchdog (typically $q = 80$), and that it is not performed on the entire file, but it is instead executed over a split. Finally, we indicate that when employing **Trapdoor Group Private Information Retrieval**, the communication cost of **StealthGuard** is minimal when $s \simeq \sqrt{Cq}$ and $t \simeq \sqrt{\frac{C}{q}}$. This results in a computation and a communication complexity (per query) at the verifier of $O(\sqrt{Cq})$ and a computation and communication complexity at the server of $O(C)$ and $O(\sqrt{Cq})$ respectively.

Example. A file F of 4GB is divided into $n = 32768$ splits $F = \{S_1, S_2, ..., S_n\}$, and each split S_i is composed of 4096 blocks of size 256 bits. **StealthGuard** inserts 8 watchdogs per split and applies an ECC that corrects up to 228 corrupted blocks (i.e., $\rho = 5\%$). We obtain thus $\hat{F} = \{\hat{S}_1, \hat{S}_2, ..., \hat{S}_n\}$, where $\hat{S}_i$ is composed of 4560 blocks of size 256 bits. This results in a redundancy of $\simeq 11.3\%$, where 11.1% redundancy is due to the use of ECC, and 0.20% redundancy is caused by the use of watchdogs.

If $(s, t) = (570, 8)$, $q = 80$ and **StealthGuard** implements the Trapdoor Group PIR [20] where $|p| = 200$ bits, then the verifier's query will be of size $\simeq 13.9$ KB, whereas the cloud server's response will be of size $\simeq 15.6$KB. In addition, if the cloud server still stores the file $\hat{F}$, then the verifier will declare the file as retrievable with probability $1 - \frac{n}{2^{60}} \simeq 1 - \frac{1}{2^{45}}$ by executing the POR protocol 1719 times. That is, by downloading 26.2MB which corresponds to 0.64% of the size of the original file F.

8 Related Work

The approach that is the closest to **StealthGuard** is the sentinel-based POR introduced by Juels and Kaliski [13]. As in **StealthGuard**, before outsourcing the file to the server, the client applies an ECC and inserts in the encrypted file special blocks, *sentinels*, that are indistinguishable from encrypted blocks. However, during the challenge, the verifier asks the prover for randomly-chosen sentinels, disclosing their positions and values to the prover. Thus, this scheme suggests a limited number of POR queries. Therefore, the client may need to download the file in order to insert new sentinels and upload it again to the cloud. [13] mentions, without giving any further details, a PIR-based POR scheme that would allow an unlimited number of challenges by keeping the positions of sentinels private, at the price of high computational cost equivalent in practice to downloading the entire file. In comparison, **StealthGuard** uses a PIR within the WS technique to retrieve a witness of the watchdog (a certain number of bits instead of the entire watchdog), and does not limit the number of POR verifications.

Ateniese et al. [1] define the concept of Provable Data Possession (PDP), which is weaker than POR in that it assures that the server possesses parts of the file but does not guarantee its full recovery. PDP uses RSA-based homomorphic tags as check-values for each file block. To verify possession, the verifier asks the server for tags for randomly chosen blocks. The server generates a proof based on the selected blocks and their respective tags. This scheme provides public verifiability meaning that any third party can verify the retrievability of a client's file. However, this proposal suffers from an initial expensive tag generation leading to high computational cost at the client. The same authors later propose in [3] a *robust auditing* protocol by incorporating erasure codes in their initial PDP scheme [1] to recover from small data corruption. To prevent an adversary from distinguishing redundancy blocks from original blocks, the latter are further permuted and encrypted. Another permutation and encryption are performed on the redundancy blocks only which are then concatenated to the file. This solution suffers from the fact that a malicious cloud can selectively delete redundant blocks and still generate valid proofs. Even though these proofs are valid, they do not guarantee that the file is retrievable.

Shacham and Waters in [17] introduce the concept of Compact POR. The client applies

Scheme	**Parameter**	**Setup cost**	**Storage overhead**	**Server cost**	**Verifier cost**	**Communication cost**
Robust PDP [3]	block size: 2 KB tag size: 128 B	4.4×10^6 exp 2.2×10^6 mul	tags: 267 MB	764 PRP 764 PRF 765 exp 1528 mul	challenge: 1 exp verif: 766 exp 764 PRP	challenge: 168 B response: 148 B
JK POR [13]	block size: 128 bits number of sentinels: 2×106	2×10^6 PRF	sentinels: 30.6 MB	$\perp$	challenge: 1719 PRP verif: $\perp$	challenge: 6 KB response:26.9 MB
Compact POR [17]	block size: 80 bits number of blocks in one split: 160 tag size: 80 bits	1 enc 5.4×10^6 PRF 1.1×10^9 mul	tags: 51 MB	7245 mul	challenge: 1 enc, 1 MAC verif: 45 PRF, 160 + 205 mul	challenge: 1.9 KB response: 1.6 KB
Efficient POR [23]	block size: 160 bits number of blocks in one split: 160	2.2×10^8 mul 1.4×10^6 PRF	tags: 26 MB	160 exp $2.6 * 10^5$ mul	challenge:$\perp$ verif: 2 exp, 1639 PRF, 1639 mul	challenge: 36 KB response: 60 B
StealthGuard	block size: 256 bits number of blocks in one split: 4096	2.6×10^5 PRF 2.6×10^5 PRP	watchdogs: 8 MB	6.2×10^8 mul	challenge: 2.0×10^6 mul verif: 1.4×10^5 mul	challenge: 23.3 MB response: 26.2 MB

Table 2: Comparison of relevant related work with **StealthGuard**.

an erasure code and for each file block, it generates *authenticators* (similar to tags in [1]), with BLS signatures [7], for public verifiability, or with Message Authentication Codes (MAC) [5], for private verifiability. The generation of these values are computationally expensive. Moreover, the number of authenticators stored at the server is linear to the number of data blocks, leading to an important storage overhead. Xu and Chang [23] propose to enhance the scheme in [17] using the technique of polynomial commitment [14] which leads to light communication costs. These two schemes employ erasure codes in conjunction with authentication tags, which induces high costs at the time of retrieving the file. Indeed, erasure coding does not inform the verifier about the position of the corrupted blocks. Thus, the verifier has to check each tag individually to determine whether it is correct or not. When a tag is detected as invalid, meaning that the corresponding block is corrupted, the verifier applies the decoding to recover the original data block.

A recent work of Stefanov et al. [19], Iris, proposes a POR protocol over authenticated file systems subject to frequent changes. Each block of a file is authenticated using a MAC to provide file-block integrity which makes the tag generation very expensive.

Compared to all these schemes, **StealthGuard** performs computationally lightweight operations at the client, since the generation of watchdogs is less expensive than the generation of tags like in [1, 17]. In addition, the storage overhead induced by the storage of watchdogs is less important than in the previous work. At the cost of more bits transmitted during the POR challenge-response, **StealthGuard** ensures a better probability of detecting adversarial corruption.

Table 2 depicts the performance results of **StealthGuard** and compares it with previous work. We analyze our proposal compared to other schemes [3, 13, 17, 23] with respect to a file of size 4 GB. The comparison is made on the basis of the POR assur-

ance of $1 - \frac{1}{2^{45}}$ computed in Section 7. We assume that all the compared schemes have three initial operations in the *Setup* phase: the application of an ECC, the encryption and the file-level permutation of data and redundancy blocks. Since these three initial operations have comparable costs for all the schemes, we omit them in the table. Computation costs are represented with `exp` for exponentiation, `mul` for multiplication, `PRF` for pseudo-random function or `PRP` for pseudo-random permutation. For **StealthGuard**, we compute the different costs according to the values provided in Section 7. For the other schemes, all initial parameters derive from the respective papers. In [17] since the information on the number of blocks in a split is missing, we choose the same one as in [23]

Setup. In our scheme, the client computes $32768 \times 8 \approx 2.6 \times 10^5$ PRF and 2.6×10^5 PRP for the generation and the insertion of watchdogs. One of the advantages of **StealthGuard** is having a more lightweight setup phase when the client preprocesses large files. Indeed, the setup phase in most of previous work [3, 17, 19, 23] requires the client to compute an authentication tag for each block of data in the file which is computationally demanding in the case of large files.

Storage Overhead. The insertion of watchdogs in **StealthGuard** induces a smaller storage overhead compared to other schemes that employ authentication tags.

Proof Generation and Verification. For **StealthGuard**, we consider the PIR operations as multiplications of elements in $\mathbb{Z}_p$ where $|p| = 200$ bits. To get the server and verifier computational costs of existing work, based on the parameters and the bounds given in their respective papers, we compute the number of requested blocks in one challenge to obtain a probability of $1 - \frac{1}{2^{45}}$ to declare the file as irretrievable: 764 blocks in [3], 1719 sentinels in [13], 45 blocks in [17] and 1639 blocks in [23]. **StealthGuard** induces high cost compared to existing work but is still acceptable.

Communication. Even if its communication cost is relatively low compared to **StealthGuard**, JK POR [13] suffers from the limited number of challenges, that causes the client to download the whole file to regenerate new sentinels. Although we realize that **StealthGuard**'s communication cost is much higher than [3, 17, 23], such schemes would induce additional cost at the file retrieval step, as mentioned earlier.

To summarize, **StealthGuard** trades off between light computation at the client, small storage overhead at the cloud and significant but still acceptable communication cost. Nevertheless, we believe that **StealthGuard**'s advantages pay off when processing large files. The difference between the costs induced by existing schemes and those induced by **StealthGuard** may become negligible if the size of the outsourced file increases.

9 Conclusion

StealthGuard is a new POR scheme which combines the use of randomly generated watchdogs with a lightweight privacy-preserving word search mechanism to achieve high retrievability assurance. As a result, a verifier can generate an unbounded number of queries without decreasing the security of the protocol and thus without the need for updating the watchdogs. **StealthGuard** has been proved to be complete and sound.

As future work, we plan to implement **StealthGuard** in order to not only evaluate its efficiency in a real-world cloud computing environment but also to define optimal values for system parameters.

10 Acknowledgment

This work was partially funded by the Cloud Accountability project - A4Cloud (grant EC 317550).

References

[1] Giuseppe Ateniese, Randal C. Burns, Reza Curtmola, Joseph Herring, Lea Kissner, Zachary N. J. Peterson, and Dawn Song. Provable data possession at untrusted stores. In Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson, editors, *ACM Conference on Computer and Communications Security*, pages 598–609. ACM, 2007. ISBN 978-1-59593-703-2. URL `http://dblp.uni-trier.de/db/conf/ccs/ccs2007.html`.

[2] Giuseppe Ateniese, Roberto Di Pietro, Luigi V. Mancini, and Gene Tsudik. Scalable and efficient provable data possession. In *Proceedings of the 4th international conference on Security and privacy in communication networks*, SecureComm '08, pages 9:1–9:10, New York, NY, USA, 2008. ACM.

[3] Giuseppe Ateniese, Randal C. Burns, Reza Curtmola, Joseph Herring, Osama Khan, Lea Kissner, Zachary N. J. Peterson, and Dawn Song. Remote data checking using provable data possession. *ACM Trans. Inf. Syst. Secur.*, 14(1):12, 2011.

[4] Monir Azraoui, Kaoutar Elkhiyaoui, Refik Molva, and Melek Önen. Stealthguard: Proofs of retrievability with hidden watchdogs. Technical report, EURECOM, June 2014. URL `http://www.eurecom.fr/publication/4338`.

[5] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying Hash Functions for Message Authentication. In *Proceedings of the 16th Annual International Cryptology conference on Advances in Cryptology, CRYPTO'96*, pages 1–15. LNCS, August 1996.

[6] Erik-Oliver Blass, Roberto di Pietro, Refik Molva, and Melek Önen. PRISM - Privacy-Preserving Search in MapReduce. In *Proceedings of the 12th Privacy Enhancing Technologies Symposium (PETS 2012)*. LNCS, July 2012.

[7] Dan Boneh, Ben Lynn, and Hovav Shacham. Short Signatures from the Weil Pairing. *J. Cryptology*, 17(4):297–319, September 2004.

[8] David Cash, Alptekin Küpçü, and Daniel Wichs. Dynamic proofs of retrievability via oblivious ram. In *EUROCRYPT*, pages 279–295, 2013.

[9] Bo Chen and Reza Curtmola. Robust dynamic provable data possession. In *ICDCS Workshops*, pages 515–525, 2012.

[10] Morris Dworkin. *Recommendation for Block Cipher Modes of Operation: Methods and Techniques*. National Institute of Standards and Technology. Special Publication 800-38A, 2001.

[11] Chris Erway, Alptekin Küpçü, Charalampos Papamanthou, and Roberto Tamassia. Dynamic provable data possession. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 213–222, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-894-0. doi: 10.1145/1653662.1653688. URL `http://doi.acm.org/10.1145/1653662.1653688`.

[12] Li Fan, Pei Cao, Jurassa Almeida, and Andrei Z. Broder. Summary Cache: a Scalable Wide-Area Web Cache Sharing Protocol. *IEEE/ACM Trans. Netw.*, 8(3):281–293, June 2000. ISSN 1063-6692.

[13] Ari Juels and Burton S. Kaliski Jr. Pors: proofs of retrievability for large files. In Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson, editors, *ACM Conference on Computer and Communications Security*, pages 584–597. ACM, 2007. ISBN 978-1-59593-703-2. URL http://dblp.uni-trier.de/db/conf/ccs/ccs2007.html.

[14] Aniket Kate, Gregory Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. *Advances in Cryptology-ASIACRYPT 2010*, pages 177–194, 2010.

[15] Zhen Mo, Yian Zhou, and Shigang Chen. A dynamic proof of retrievability (por) scheme with o(logn) complexity. In *ICC*, pages 912–916, 2012.

[16] Irving S. Reed and Gustave Solomon. Polynomial Codes Over Certain Finite Fields. *Journal of the Society of Industrial and Applied Mathematics*, 8(2):300–304, 06/1960 1960.

[17] Shacham, Hovav and Waters, Brent. Compact proofs of retrievability. In *Proceedings of the 14th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology*, ASIACRYPT '08, pages 90–107, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-89254-0. doi: 10.1007/978-3-540-89255-7_7. URL http://dx.doi.org/10.1007/978-3-540-89255-7_7.

[18] Elaine Shi, Emil Stefanov, and Charalampos Papamanthou. Practical dynamic proofs of retrievability. In *ACM Conference on Computer and Communications Security*, pages 325–336, 2013.

[19] Emil Stefanov, Marten van Dijk, Ari Juels, and Alina Oprea. Iris: a scalable cloud file system with efficient integrity checks. In *ACSAC*, pages 229–238, 2012.

[20] Jonathan Trostle and Andy Parrish. Efficient Computationally Private Information Retrieval from Anonymity or Trapdoor Groups. In *Proceedings of Conference on Information Security*, pages 114–128, Boca Raton, USA, 2010.

[21] Qian Wang, Cong Wang, Jin Li, Kui Ren, and Wenjing Lou. Enabling public verifiability and data dynamics for storage security in cloud computing. In *Proceedings of the 14th European conference on Research in computer security*, ESORICS'09, pages 355–370, Berlin, Heidelberg, 2009. Springer-Verlag.

[22] Qian Wang, Cong Wang, Kui Ren, Wenjing Lou, and Jin Li. Enabling public auditability and data dynamics for storage security in cloud computing. *IEEE Trans. Parallel Distrib. Syst.*, 22(5):847–859, 2011.

[23] Jia Xu and Ee-Chien Chang. Towards efficient proofs of retrievability. In *ASIACCS*, pages 79–80, 2012.

[24] Qingji Zheng and Shouhuai Xu. Fair and dynamic proofs of retrievability. In *CODASPY*, pages 237–248, 2011.