SPORT: Sharing Proofs of Retrievability across Tenants

Frederik Armknecht
University of Mannheim, Germany
armknecht@uni-mannheim.de

Jens-Matthias Bohli
NEC Laboratories Europe, Germany
hjens.bohli@neclab.eu

David Froelicher
NEC Laboratories Europe, Germany
david.froelicher@neclab.eu

Ghassan O. Karame
NEC Laboratories Europe, Germany
ghassan@karame.org

***Abstract*—Proofs of Retrievability (POR) are cryptographic proofs which provide assurance to a single tenant (who creates tags using his secret material) that his files can be retrieved in their entirety. However, POR schemes completely ignore storage-efficiency concepts, such as multi-tenancy and data deduplication, which are being widely utilized by existing cloud storage providers. Namely, in deduplicated storage systems, existing POR schemes would incur an additional overhead for storing tenants' tags which grows linearly with the number of users deduplicating the same file. This overhead clearly reduces the (economic) incentives of cloud providers to integrate existing POR/PDP solutions in their offerings.**

In this paper, we propose a novel storage-efficient POR, dubbed SPORT, which transparently supports multi-tenancy and data deduplication. More specifically, SPORT enables tenants to securely share the same POR tags in order to verify the integrity of their deduplicated files. By doing so, SPORT considerably reduces the storage overhead borne by cloud providers when storing the tags of different tenants deduplicating the same content. We show that SPORT resists against malicious tenants/cloud providers (and against collusion among a subset of the tenants and the cloud). Finally, we implement a prototype based on SPORT, and evaluate its performance in a realistic cloud setting. Our evaluation results show that our proposal incurs tolerable computational overhead on the tenants and the cloud provider.

1. Introduction

The cloud is gaining several adopters among SMEs and large businesses that are mainly interested in minimizing the costs of both deployment and infrastructure management and maintenance.

Cost effectiveness is realized in the cloud through the integration of multi-tenancy and storage efficiency solutions with tailored distributed algorithms that ensure unprecedented levels of scalability and elasticity. The combination of multi-tenancy solutions with storage efficiency techniques (e.g., data deduplication) promises drastic cost reductions. For instance, recent studies show that cross-user data deduplication can save storage costs by more than 50% in standard file systems [27], [28], and by up to 90-95% for back-up applications [27]. Moreover, nearly three quarters of these savings could also be obtained by means of straightforward whole-file deduplication [28].

The advent of cloud storage, however, introduces serious concerns with respect to the confidentiality, integrity, and availability of the outsourced data [22]. For instance, Google recently admitted permanent loss of customers' data in their storage systems due to a malfunction of local utility grid located near one of Google's data centers [4]. The literature features a number of solutions that enable users to verify the integrity and availability of their outsourced data [14], [16], [26], [32]. Solutions include Proofs of Retrievability (POR) [26], [32] which provide end-clients with the assurance that their data is retrievable, and Proofs of Data Possession (PDP) [8] which enable a client to verify that its stored data has not undergone any modifications. Although these solutions can be indeed effective in detecting data loss, they completely ignore storage-efficiency requirements, such as multi-tenancy and data deduplication, which are being widely utilized by existing cloud storage providers.

Namely, existing POR/PDP solutions assume a single trusted tenant (i.e., honest verifier) who pre-processes the files to create tags using secret material before outsourcing them to the cloud, and later regularly performs verifications on the pre-processed files and tags in order to react as early as possible in case of data loss.

A straightforward adaptation for the multi-tenant case would be that each tenant constructs and stores his own tags in the cloud. However, this approach threatens to cancel out the benefits of data deduplication over popular objects—which might reduce the (economic) incentives of cloud providers to integrate existing POR/PDP solutions in their offerings.

In Figure 1, we estimate the additional storage overhead incurred by such an instantiation using the existing privately-verifiable POR scheme of [32], the publicly-verifiable RSA-based POR/PDP schemes of [8], [32], and finally the publicly-verifiable BLS-based POR scheme of [32]. Here, we assume a block size of 8 KB. Notice that the smaller

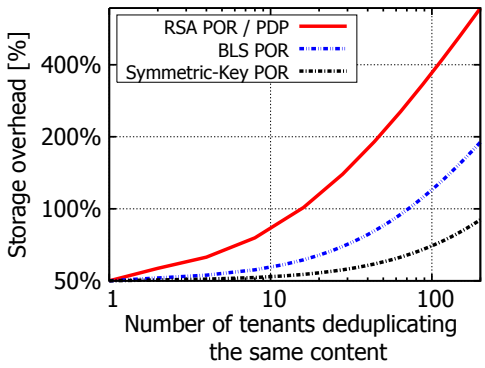


Figure 1. Storage overhead due to the storage of individual tenant tags in (private and public) POR schemes of [32] and the public PDP scheme of [8]. Here, we assume 8 KB block sizes.

the block size is, the larger is the storage overhead due to the storage of the tenant tags (since the tags' sizes are fixed per block and are independent of the block size). Our results show that, even with block sizes as large as 8 KB, the metadata storage per tenant incurred by existing POR/PDP schemes are considerable. Clearly, such storage overhead reduces the profitability of existing storage-efficiency solutions and, in turn, reduces the incentives of providers to integrate POR/PDP schemes in their offerings. For example, in the 2048-bit RSA-based schemes of [8], [32], each POR tag requires around 1.5% additional storage overhead per tenant per block; popular blocks, e.g., shared amongst 50 tenants, would then require 75% additional storage overhead.

Consequently, one way to minimize this additional storage overhead would require to use the same set of tags for each tenant. However, in practice, given that files are typically deduplicated across tenants, and different tenants do not tend to trust each other, tenants will be reluctant on sharing the secret material used to construct tags in POR/PDP. Notice that the leakage of the secret material invalidates the security of POR/PDP; for instance, a cloud provider which has access to the secret material can always construct correct responses in existing POR schemes—even if the oustourced data is deleted.

In this paper, we address this problem, and propose a novel solution (SPORT)—which goes one step beyond existing POR and transparently supports multi-tenancy and data deduplication. More specifically, SPORT enables different tenants to share the same POR tags in order to verify the integrity of their deduplicated files. By doing so, SPORT considerably reduces the storage overhead borne by cloud providers when storing the tags of different tenants deduplicating the same content. We show that SPORT resists against malicious tenants/cloud providers (and against collusion among a subset of the tenants and the cloud).

We argue that SPORT is technically and economically viable. Indeed, by reconciling functional (i.e., storage efficiency and multi-tenancy) and security requirements (i.e., data retrievability in the cloud), SPORT provides considerable incentives for cloud providers to issue security SLAs for their users accounting for data loss, in exchange, e.g., for additional service fees. While the main barriers of wider adoption of POR lie in the lack of compliance with functional requirements in the cloud, SPORT bridges these gaps and enables cloud providers to offer their customer extended POR services, without incurring additional bandwidth and storage overhead.

Our contributions can be summarized as follows:

Formal Framework. We propose the first formal framework and a security model for multi-tenant POR, dubbed MTPOR. Our framework extends the POR model outlined in [32] and addresses security risks that have not been covered so far in existing models.

Concrete Instantiation. We describe a concrete MTPOR scheme, dubbed SPORT, which builds upon publicly verifiable BLS POR scheme [32] and that is secure in the MTPOR model. SPORT deploys a novel mechanism which enables different tenants to (in-place) update the POR tags created by others corresponding to the same deduplicated file—thus saving considerable storage. We show that this process resists against a malicious cloud provider, and malicious tenants (as well as against any collusion between tenants/providers).

Prototype Implementation. We implement and evaluate a prototype based on SPORT, and we show that our proposal incurs comparable overhead on the users and cloud providers, when compared to existing publicly-verifiable POR/PDP schemes.

The remainder of this paper is organized as follows. In Section 2, we introduce a novel framework for secure multi-tenant POR, MTPOR. In Section 3, we propose SPORT, an efficient instantiation of MTPOR, and analyze its security. In Section 4, we describe a prototype implementation and evaluation of SPORT and compare its performance to existing publicly-verifiable POR/PDP schemes. In Section 5, we overview related work in the area and we conclude the paper in Section 6.

2. Multi-Tenant Proofs of Retrievability

In this section, we introduce a formal model for multi-tenant POR (MTPOR). Since MTPOR extends POR, we first quickly recall the POR model from [32].

2.1. Single-Tenant Proofs of Retrievability

Proofs of Retrievability (POR) are interactive protocols that cryptographically *prove* the retrievability of outsourced data. More precisely, POR consider a model comprising of a *single* user (or tenant), and a service provider that stores a file pertaining to the user. POR basically consist of a challenge-response protocol in which the service provider proves to the tenant that its file is still intact and retrievable. Note that POR only provide a guarantee that a fraction p of

the file can be retrieved. For that reason, POR are typically performed on a file which has been erasure-coded in such a way that the recovery of any fraction p of the stored data ensures the recovery of the file [7].

A POR scheme consists of four procedures [32], setup, store, verify, and prove. The latter two algorithms define a protocol for proving file retrievability. We refer to this protocol as the POR *protocol* (in contrast to a POR scheme that comprises all four procedures). A POR scheme consists of four procedures [32], setup, store, verify, and prove [32].

setup. This randomized algorithm generates the involved keys and distributes them to the parties. In case public keys are involved in the process, these are distributed amongst all parties.

store. This randomized algorithm takes as input the keys of the user and a file $\widetilde{M}$. The file is processed and store outputs M which will be stored on the server. store also generates a file tag τ, which contains additional information about M shared by the user and the service provider and is used in the subsequent protocol.

verify. The randomized verification algorithm takes the secret and public key, and the file tag τ outputted by store during protocol execution. It outputs at the end of the protocol run TRUE if the verification succeeds, meaning that the file is being stored on the server, and FALSE otherwise.

prove. The prover algorithm takes as input the public key and the file tag τ and M that is output by store.

2.2. Multi-Tenant POR (MTPOR)

In the sequel, we formally define the model for multi-tenant POR, dubbed MTPOR. Similar to the standard POR model, we distinguish between tenants and the service provider S. The main difference to the aforementioned POR model lies in the direct integration of the notion of multiple tenants (or users) that *can* upload the *same* file to be stored at S. In general, these tenants are independent of each other and do not necessarily trust each other. That is, we do not assume the existence of shared secrets or direct communication between the different tenants.

Conforming with the operation of existing cloud providers, we assume that S stores duplicate data (either at the block level or the file level) uploaded by different users only once—thus tremendously saving storage costs. Recent studies show that cross-user data deduplication can save storage costs by more than 50% in standard file systems [27], [28], and by up to 90-95% for back-up applications [27]. Nonetheless, any user who uploaded a file should be able to execute PORs later on to verify that the file has been stored. Therefore, we assume that the provider stores for each uploaded file M the following:

1) The file M itself which is stored only *once*.
2) Additional metadata $\mathsf{Data}(M)$ which contains e.g., a list of the users who uploaded M and information for conducting the POR on M.

If a provider stores several files, e.g., $M_1, \ldots, M_n$, we assume that S maintains separate metadata $\mathsf{Data}(M_i)$ for each file that he stores. Clearly, metadata corresponding to different files may contain duplicate information about the same user to account for the case where the user has uploaded different files. For simplicity, if we refer to a metadata $\mathsf{Data}(M)$ where the file M has not been uploaded yet, we assume that the metadata is empty, e.g., $\mathsf{Data}(M) = \perp$. Moreover, for ease of presentation, we simply write Data instead of $\mathsf{Data}(M)$, whenever the considered file is clear from the context.

In the sequel, we assume that all communication is authenticated and encrypted (e.g., using the TLS protocol). Similar to the POR model outlined in Section 2.1, we consider three basic procedures: Setup, Store, and POR where POR is an interactive protocol that extends the notions of verify and prove discussed earlier.

The Setup Protocol. The Setup protocol is used to establish the keys that are required by the users to instrument the POR; notice that these keys may be different from the keys used to authenticate and encrypt the communication. The input of this protocol is the security parameter λ and the outputs are a secret and a public key. In case such keys are not required in some settings, the respective values are simply $\perp$. Formally, it holds for a party running Setup that:

$$(\mathrm{sk}, \mathrm{pk}) \leftarrow \mathsf{Setup}(\lambda) \tag{1}$$

Whenever it is necessary to refer to the keys of a specific user U, we denote the corresponding secret and public key by $\mathrm{sk}_{\mathtt{U}}$ and $\mathrm{pk}_{\mathtt{U}}$, respectively.

The Store Protocol. This randomized file-storing protocol takes as input the keys of the user, a file $\widetilde{M}$ from the user U that has to be stored by the service provider and the current metadata of the provider. During the protocol execution, it may happen that the file is processed prior to being stored—we denote this file by M. The output of Store is a verification key $\mathrm{vk}_{\mathtt{U}}$ that allows the user to run the proof of retrievability. The verification key may be equal to (or depend on) the secret or public key that the user generated in the Setup procedure. On the provider's side, the (processed) file M is stored and the metadata $\mathsf{Data}(M)$ is updated.[1] Formally, it holds that:

$$\begin{aligned} \mathsf{Store}\colon \quad & [\mathtt{U} : \mathrm{sk}_{\mathtt{U}}, \mathrm{pk}_{\mathtt{U}}, \widetilde{M};\ \mathtt{S} : \mathsf{Data}(M)] \\ & \longrightarrow [\mathtt{U} : \mathrm{vk}_{\mathtt{U}};\ \mathtt{S} : M, \mathsf{Data}(M)]. \end{aligned}$$

Observe that unlike the single-tenant POR model outlined in Sec. 2.1, we do not utilize the notion of a file tag τ that needs to be known to both parties. Instead, we split this information into two parts: *(i)* the part that needs to be known to the user (or verifier)—captured by the notion of a verification key $\mathrm{vk}_{\mathtt{U}}$—and *(ii)* the part that needs to be stored by the provider, which is part of the metadata $\mathsf{Data}(M)$.

Observe that the Store protocol essentially requires a user to *join* the set of users who all uploaded the same file M. In practice, one may also prefer to have a protocol that allows users to *leave* this set. However, as we aim to

1. Recall that we model the state $\mathsf{Data}(M)$ being empty unless M is uploaded.

achieve that the model for a multi-tenant POR falls back to the classical single-tenant POR when only one user is present, we do not integrate such a protocol into the model.

The POR Protocol. The aim of the POR protocol is to allow a user (or a verifier) to check if the file is still entirely stored at the provider. To this end, the user uses his personal keys and the verification key to verify the response sent by the provider; the latter uses the public key of the user, the uploaded file, and the metadata to issue the response to the user. At the end, the user outputs a Boolean value $\mathrm{dec} \in \{\mathrm{TRUE}, \mathrm{FALSE}\}$. Formally, we have:

$$\begin{aligned}\mathsf{POR} \quad : \quad & [\mathtt{U} : \mathrm{sk}_{\mathtt{U}}, \mathrm{pk}_{\mathtt{U}}, \mathrm{vk}_{\mathtt{U}};\ \mathtt{S} : \mathrm{pk}_{\mathtt{U}}, M, \mathsf{Data}(M)] \\ & \longrightarrow [\mathtt{U} : \mathrm{vk}_{\mathtt{U}}, \mathrm{dec}].\end{aligned}$$

The protocol run is accepted by the user if and only if $\mathrm{dec} = \mathrm{TRUE}$. Unlike the Store protocol, we point out that metadata maintained by the provider is not modified in this protocol. However, it may be the case that the user has to update his verification key $\mathrm{vk}_{\mathtt{U}}$—which explains the reason why this is part of the local output.

Adopting common terminology, we say that an MTPOR is *publicly-verifiable* if the POR-protocol does not require the secret key $\mathrm{sk}_{\mathtt{U}}$ of the user. In such cases, verification can be outsourced to a third party verifier $\mathcal{V}$. Otherwise, an MTPOR is said to be *privately-verifiable*.

2.3. Correctness

We consider correctness from the perspective of the provider and the user. From the former perspective, it is necessary to ensure that if two honest users initiate Store with the same file $\widetilde{M}$, then during the protocol run the same file M is outsourced. This notion ensures correct and effective whole-file deduplication.

From a user's perspective, correctness intuitively means that if an *honest* user U uploaded a file to an *honest* provider S and later on runs the POR protocol, the user should accept. For a formal treatment though, one needs to take into account that in the time lapse between the file upload by a user U and the execution of the POR procedure, the provider may have executed other Store protocol runs (i.e., with another user $\mathtt{U}'$) which may have changed the metadata. For example, while the file M would not be stored a second time, information about $\mathtt{U}'$ would be integrated into the metadata $\mathsf{Data}(M)$. Still, the POR protocol executed by U afterwards should be successful although $\mathsf{Data}(M)$ has changed from the initial Store performed by U. We stress that this property has to hold even if some tenants are malicious, e.g., deviate from the protocol, as long as both U and S are honest. We capture this formally by saying that the new content of $\mathsf{Data}(M)$ *evolved* from the last known metadata that has been generated when U executed Store.

Definition 1 (Evolvable Metadata). For a key pair $(\mathrm{sk}, \mathrm{pk})$, a file M, and two metadata $\mathsf{Data}, \mathsf{Data}'$, we denote by $\mathbb{E} = \mathbb{E}(\mathrm{sk}, \mathrm{pk}, M, \mathsf{Data}, \mathsf{Data}')$ the event that when running Store with inputs $\mathrm{sk}, \mathrm{pk}, M$ on the user side and input Data on the provider side where the provider behaves honestly, the metadata with respect to M is updated to Data'. We say that Data' is *evolvable* from Data with respect to $(\mathrm{sk}, \mathrm{pk}, M)$ if:

$$\Pr\left[\mathbb{E}(\mathrm{sk}, \mathrm{pk}, M, \mathsf{Data}, \mathsf{Data}')\right] > 0. \tag{2}$$

This captures that the metadata Data may be possibly updated to Data' while running Store for a file M with keys $(\mathrm{sk}, \mathrm{pk})$. We express this by:

$$\mathsf{Data} \xrightarrow{(\mathrm{sk}, \mathrm{pk}, M)} \mathsf{Data}'. \tag{3}$$

Moreover, we extend this notion to the case where one metadata may be evolved from another metadata also after running Store multiple times. That is, we write $\mathsf{Data} \overset{M}{\Rightarrow} \mathsf{Data}'$ if there exists metadata $\mathsf{Data}_1, \ldots, \mathsf{Data}_\ell$ and key pairs $(\mathrm{sk}_1, \mathrm{pk}_1), \ldots, (\mathrm{sk}_\ell, \mathrm{pk}_\ell)$ such that:

- $\mathsf{Data} \xrightarrow{(\mathrm{sk}_1, \mathrm{pk}_1, M)} \mathsf{Data}_1$
- $\mathsf{Data}_i \xrightarrow{(\mathrm{sk}_{i+1}, \mathrm{pk}_{i+1}, M)} \mathsf{Data}_{i+1}$ for $i = 1, \ldots, \ell - 1$
- $\mathsf{Data}_\ell \xrightarrow{(\mathrm{sk}_{\ell+1}, \mathrm{pk}_{\ell+1}, M)} \mathsf{Data}'$

In this case, we likewise say that Data' is evolvable from Data. Finally, we say that a metadata Data is M-evolvable if $\perp \overset{M}{\Rightarrow} \mathsf{Data}$. Recall that the metadata is initially set to $\perp$. Hence, this notion expresses that Data could be a possible metadata resulting from multiple executions of Store for the same file M.

Note that the definition of evolvable metadata refers to the file M as seen by S (i.e., after the user has uploaded it to the cloud). We are now ready to give the formal definition for correctness of a MTPOR.

Definition 2 (Correctness). An MTPOR scheme is correct if the following holds with overwhelming probability in the security parameter for any key pairs $(\mathrm{sk}, \mathrm{pk})$, any file M, and any M-evolvable metadata Data.
Consider a Store execution with the above mentioned inputs and the outputs vk and Data'. Then, it holds for any metadata Data'' with $\mathsf{Data}' \overset{M}{\Rightarrow} \mathsf{Data}''$, and any protocol execution of POR with inputs $\mathrm{sk}, \mathrm{pk}, \mathrm{vk}$ (user side) and M, Data'' (provider side), the user accepts at the end of the protocol run.

2.4. Security

MTPOR exhibits similar security considerations as in the case of single-tenant POR. Recall that the purpose of executing a POR or an MTPOR is to allow users to verify whether the file is still entirely stored by the provider. Consequently, we do not consider in this paper other security goals besides data retrievability. For example, to ensure the confidentiality of the file $\widetilde{M}$, the user can encrypt the file during the Store protocol by leveraging message-locked encryption using a key derived from the file contents such as [6], [10]. We also do not take into account any privacy leakage resulting from the use of deduplication [24]. More specifically, we assume that the cloud storage employs

Proofs of Ownership [23] in order to ensure that tenants indeed possess the files.

In a nutshell, our attacker model concentrates on a dishonest provider A who aims to mislead the user by storing few (or no) parts of the file but still tries to pass the POR protocol. Hence, security can be captured by a similar notion as in single-tenant POR, i.e., using the notion of an extractor. Recall that an extractor algorithm expresses the notion that if a provider is able to convince a user within the POR protocol, one can extract the file from the provider.

However, as opposed to the case of single-tenant POR, we have to consider a stronger attacker model where A might collude with one or several tenants. We call such tenants *corrupted*. Corrupted tenants are under the full control of the attacker. In particular, the attacker knows all their keys, including the verification key.

To formalize security, we envision a game between an adversary A and an environment E. The task of the environment is to play the role of all honest users and to challenge the adversary. That is, whenever we say that an honest user executes a certain protocol, we mean that the environment honestly executes the protocol on behalf of this user. However, we allow the attacker to initiate protocol runs by honest users at her wish. To keep track of all honest users, we assume that the environment maintains a set $\mathcal{U}$. The adversary can now interact with the environment using four types of queries: a Setup-query, a Store-query, a POR-query, and a Corrupt-query.

In a Setup-query, a new honest user U is created. That is, the environment generates a key pair $(\mathrm{sk}_\mathtt{U}, \mathrm{pk}_\mathtt{U})$ and stores a quadruple $(\mathtt{U}, \mathrm{sk}_\mathtt{U}, \mathrm{pk}_\mathtt{U}, \mathrm{VK}_\mathtt{U})$ in $\mathcal{U}$. The first entry identifies the user, the following two entries give the keys of the user, while $\mathrm{VK}_\mathtt{U}$ represents the set of verification keys stored by the user—each per file stored by U. Initially, $\mathrm{VK}_\mathtt{U}$ is set to $\emptyset$. In addition, we assume that the environment hands the public key to the adversary.

In a Store-query, the adversary presents a user U and a file M to the environment E. The environment stores for each file M that has been part of a Store-request a set $\mathsf{Users}(M)$ that contains the set of users that have executed Store with the attacker for the file M. Upon receiving a Store-query, the environment proceeds as follows:

- If the user has not been created before (in terms of a Setup-request) or if the user is corrupted, the environment aborts the game. This is equivalent to checking if some quadruple $(\mathtt{U}, \mathrm{sk}_\mathtt{U}, \mathrm{pk}_\mathtt{U}, \mathrm{VK}_\mathtt{U})$ is stored in $\mathcal{U}$.
- The environment executes on behalf of the user the Store protocol with the adversary, using the key pair $\mathrm{sk}_\mathtt{U}, \mathrm{pk}_\mathtt{U}$ and the file M as input. Let $\mathrm{vk}_\mathtt{U}$ denote the output for the user. If the set $\mathrm{VK}_\mathtt{U}$ of verification keys stored for user U already contains a tuple $(M, \mathrm{vk}'_\mathtt{U})$, this tuple is replaced by $(M, \mathrm{vk}_\mathtt{U})$. This reflects the case that an honest user runs Store again for the same file. Otherwise, $\mathrm{VK}_\mathtt{U}$ is updated to $\mathrm{VK}_\mathtt{U} \leftarrow \mathrm{VK}_\mathtt{U} \cup \{(M, \mathrm{vk}_\mathtt{U})\}$ and the quadruple stored in $\mathcal{U}$ is updated accordingly.
- Finally, the environment stores U in $\mathsf{Users}(M)$ if this is not the case already.

Observe that this captures the situation where the adversary runs the Store protocol for a given file with an honest user. As he can simulate the protocol with a corrupted user on his own, we do not need a separate oracle query for this case. Similarly, corrupted users do not need to be stored in $\mathcal{U}$—see also below.

In a POR-query, the attacker A gives a user U and a file M to E. The environment first checks if U is recorded in $\mathsf{Users}(M)$ and if a quadruple $(\mathrm{id}, \mathrm{sk}_\mathtt{U}, \mathrm{pk}_\mathtt{U}, \mathrm{VK}_\mathtt{U})$ is stored in $\mathcal{U}$. If this is not the case, E aborts. Otherwise, it executes on behalf of U the POR-protocol with A, using the values $\mathrm{sk}_\mathtt{U}, \mathrm{pk}_\mathtt{U}, \mathrm{vk}_\mathtt{U}$ as input such that $(M, \mathrm{vk}_\mathtt{U}) \in \mathrm{VK}_\mathtt{U}$. The purpose of this query is to allow the attacker to execute the POR-protocol with an honest user of its choice. Also here, POR executions with corrupted users can be simulated by A and hence do not require any interaction with the environment.

Finally, in a Corrupt-query, the attacker A hands a user U to E. If the user is not recorded in $\mathcal{U}$, it aborts the game. Otherwise, it locates the corresponding quadruple $(\mathrm{id}, \mathrm{sk}_\mathtt{U}, \mathrm{pk}_\mathtt{U}, \mathrm{VK}_\mathtt{U}) \in \mathcal{U}$ and hands the secret key $\mathrm{sk}_\mathtt{U}$ and the set of verification keys $\mathrm{VK}_\mathtt{U}$ to the attacker. Furthermore, it removes the quadruple from $\mathcal{U}$, indicating that U is no longer honest.

The attacker is allowed to make a polynomial number of queries (i.e., polynomial in the security parameter λ). We say that the attacker is *static* if he makes no Corrupt-queries and *adaptive* otherwise. Notice that a static adversary can still collude with corrupted users but these are under the control of the attacker right from the start. That is, the attacker can create both honest and corrupted users but cannot corrupt honest users. Eventually, the attacker quits the game by selecting a file M, an honest user U that is recorded in $\mathsf{Users}(M)$, and by providing the description of a cheating prover P. Observe that the attacker cannot quit if no honest user is present at this point in time.

From this point on, the definitions are analogous to the standard POR definitions [32]. Namely, we say that a cheating prover P is ε-admissible if in an ε fraction of POR runs with U, this user accepts at the end of the protocol. Here, the probability is over the coins of the user and the prover.

Definition 3 (Soundness). We say a MTPOR is ε-sound if there exists an efficient extraction algorithm Extr such that, for every adversary A, whenever A executes the game explained above, outputs a ε-admissible cheating prover P for a file M and a user U, the extraction algorithm recovers M from P except with negligible probability. We say that it is *strongly* ε-sound if this property holds in the presence of an adaptive attacker.

2.5. Tag Sharing

Within our model, we distinguish between two types of data that are stored by the provider: the uploaded files and associated metadata. The crucial aspect of multi-tenant POR lies in the deduplication of the files, meaning that duplicate

files/objects are stored only once. For generality purposes, multi-tenant POR, however, does not make any specific assumptions about the storage of the associated metadata.

In fact, a trivial instantiation of multi-tenant POR would be that the metadata $\mathsf{Data}(M)$ is simply a collection of the user data for all users $\mathtt{U} \in \mathsf{Users}(M)$ that are registered for this file M and the respective file tags $\tau_{\mathtt{U}}$ as outlined for single-tenant PORs (cf. Sec. 2.1). Here, each tenant $\mathtt{U}$ constructs his own tags $\tau_{\mathtt{U}}$ using his own secret material; that is, only the file is deduplicated, but the tags are stored separately for each user. While this approach would resist against malicious tenants, its storage overhead is essentially the number of tags times the number of users (cf. Figure 1).

In this section, we address this problem and introduce the definition of *tag sharing*. The overall goal of tag sharing is to enable storage efficiency with respect to the storage of that tags. To motivate the definition, observe that the size $|\mathsf{Data}(M)|$ of the metadata can be expressed as a function of two parameters[2]: the size $|M|$ of M, and the number k of users registered to the file. For example, a straightforward realization of MTPOR based on a single-tenant POR consists of storing the file only once but storing the tags separately for each user. In this case, it would hold that $|\mathsf{Data}(M)| \in \mathcal{O}(k \cdot |M|)$. That is, the number of users and the file size are directly coupled in the storage complexity. The goal of tag sharing is to make the storage consumed for storing the tags *independent* of the number of users. We capture this as follows.

Definition 4 (Tag Sharing MTPOR). We say that an MTPOR achieves *tag sharing* if it holds for all files M (with their respective number of users k) and the maintained metadata, that its size can be expressed by

$$|\mathsf{Data}(M)| = f(|M|) + g(k) \tag{4}$$

for some function f and g. We denote by $\mathsf{Data} = (\mathsf{Data}_{\mathrm{file}}, \mathsf{Data}_{\mathrm{users}})$ the splitting of the metadata such that $|\mathsf{Data}_{\mathrm{file}}| = f(|M|)$ and $|\mathsf{Data}_{\mathrm{users}}| = g(k)$.

We stress at this stage that the storage efforts induced by the number of users and the file size, respectively, are *decoupled*. Notice that we make no assumptions on the functions f and g—for instance, g could be a constant function.

Next, we highlight an intrinsic property of the Store protocol that holds in a tag sharing MTPOR. Recall that in a tag sharing MTPOR, the size of $\mathsf{Data}_{\mathrm{file}}$ is $f(|M|)$ and is therefore independent of the number of users. Hence, even if the number k of users that upload this file continuously increases, the size of $\mathsf{Data}_{\mathrm{file}}$ is bounded by a function that solely depends on $|M|$. This means that, after a certain number of uploads, the information stored in $\mathsf{Data}_{\mathrm{file}}$ may still change when Store is executed but the size will no longer increase. We denote this variant of Store as Store^*:

Definition 5. In a tag sharing MTPOR, we refer to the variant of the Store protocol where the $\mathsf{Data}_{\mathrm{file}}$ part of the metadata does not further increase as Store^*. It formally holds

$$\begin{aligned} \mathsf{Store}^*: \quad & [\mathtt{U} : \widetilde{M};\ \mathtt{S} : \mathsf{Data}_{\mathrm{file}}, \mathsf{Data}_{\mathrm{users}}] \\ \rightarrow \quad & [\mathtt{U} : \mathrm{vk}_{\mathtt{U}};\ \mathtt{S} : M, \mathsf{Data}^*_{\mathrm{file}}, \mathsf{Data}^*_{\mathrm{users}}]. \end{aligned}$$

with $|\mathsf{Data}^*_{\mathrm{file}}| \leq |\mathsf{Data}_{\mathrm{file}}|$.

Notice that Store^* updates the *metadata* associated to a file but not the file itself.

3. SPORT: Sharing POR across Tenants

In this section, we detail SPORT, an efficient publicly verifiable instantiation of tag sharing MTPOR. We start by outlining the main intuition behind our scheme.

3.1. Overview

One straightforward approach to realize a tag sharing MTPOR from existing single-tenant POR schemes would consist of sharing the keys required to verify the POR among all tenants deduplicating the same file. However, this approach would not resist malicious tenants who can share the secret material used to construct the POR with the cloud provider. Namely, in case the secret material is leaked to the provider (e.g., by means of a malicious/colluding tenant), then the provider may be able to correctly answer all POR challenges, without the need to store the file. *This equally applies to existing public-verifiable POR schemes; if the original creator leaks the private keys used to construct the POR, the underlying security of the POR is invalidated. This also rules out the possibility to derive the secret material to instrument POR from the file content itself (e.g., in a way similar to message-locked encryption [10])*. In this case, the entropy of the secret material will be dependent on the predictability of messages; providers can acquire the material e.g., by means of brute-force search, and/or by colluding with malicious tenants.

To overcome these challenges, SPORT extends the publicly verifiable POR scheme of SW [32] based on BLS signatures. Here, tags were essentially signatures of combination of message blocks. Unlike [32] which exploits the message-homomorphism of BLS signatures to allow for compact batch verification of multiple blocks, SPORT goes one step further and leverages key-homomorphism of BLS signatures to combine different tags (protected by different keying material) constructed by multiple untrusted tenants into one tag—thus resulting in tremendous storage savings. Key-homomorphism here ensures that the underlying secret material of the combined tag depends on the contribution of each of the users registered to the same content. As a result, SPORT support public verifiability and provides security against a collusion between tenants/service provider as long as there is one single honest tenant storing the content (and not compromising his own secret material).

SPORT achieves this without requiring any trusted entity (e.g., an identity manager or a proxy) to mediate the *update*

2. We assume here that global values such as the security parameter are fixed.

of tags shared by different tenants. In SPORT, this is realized using the (decentralized) Store* protocol—a variant of the standard Store POR protocol—that is executed by tenants deduplicating the same content. Unlike Store, the Store* routine in SPORT only involves updating the tags; we show that the tags can be efficiently verified in a single batch (for the entire file) by the provider. Moreover, we show that tenants only need to verify the inclusion of their own private keys within the updated keys—while the correctness of the tags is implicitly ensured during the POR protocol. Observe that the service provider does not have to be trusted when creating or updating the tags. This is the main reason why cryptographic accumulators could not be used in our setting.[3] However, most efficient accumulator constricts assume a *trusted setup*, meaning that the party who setups the accumulator needs to be trusted. Although a number accumulators can deal with untrusted setup case [21], [30], these proposals are either inefficient or exhibit other impeding drawbacks.

In SPORT, although the Store* protocol incurs computational load on the clients, we argue that SPORT offers tremendous economic benefits to the clients and the service provider. Namely, SPORT allows clients to trade the *permanent* storage costs associated with the additional storage of file tags, with the *one-time* use of their computational resources to update the tags—thus reducing the total costs borne on users. Moreover, SPORT enables the provider to offer differentiated services, such as proving the integrity of stored files, with minimal additional storage overhead.

In the following subsections, we go into greater detail on the various parts of SPORT, starting with the protocol specification, then moving on to the security analysis of our scheme.

3.2. Protocol Specification

We now detail the specifications for the procedures of SPORT.

Specification of the Setup protocol
SPORT makes use of the multi-signature scheme based on BLS signatures described in [31]. To this end, it deploys a group G of prime order p with generator g, and a computable efficient bilinear map $e : G \times G \rightarrow G_T$. In addition, we require three independent secure hash functions $h_0, h_1, h_2 : \{0,1\}^* \rightarrow G$ from the set of bit strings into the group G (e.g., the BLS hash function). We refer the readers to [31] for more details.

Each user U that participates in the scheme is required to setup a key pair of signature key and verification key. More precisely, the user chooses a private key $\mathrm{sk}_\mathtt{U} \in \mathbb{Z}_p$, and the corresponding public key is then given by $\mathrm{pk}_\mathtt{U} = g^{\mathrm{sk}_\mathtt{U}} \in G$. Moreover, U also needs to provide a proof of possession $\mathrm{pop} = \mathrm{pop}(\mathrm{sk}_\mathtt{U})$ to prove possession of the secret key $\mathrm{sk}_\mathtt{U}$. As we show later, proofs of possession prevent the provider from replacing/creating tags using other keys whose secret-keys are known to him. In a nutshell, our proof of possession (adapted from [31]) is constructed as follows: $\mathrm{pop}(\mathrm{sk}_\mathtt{U}) = h_0(\langle \mathrm{pk}_\mathtt{U} \rangle)^{\mathrm{sk}_\mathtt{U}}$ and $\langle \mathrm{pk}_\mathtt{U} \rangle$ denotes an encoding of $\mathrm{pk}_\mathtt{U}$ as a bit string. As we show later, such a proof of possession is required to protect against a rogue-key attack, where an adversary forges the multi-signature of a group using an invalid key (i.e., a public key whose private key is unknown to the adversary).

Specification of the Store protocol
To store a new file $\widetilde{M}$, the user first has to encode the file with a deterministic erasure code as required by the utilized POR (in order to provide extractability guarantees). As mentioned earlier, we assume that prior to this step the file is encrypted with a deterministic file-key securely obtained, e.g., using the schemes of [6], [10], in order to support deduplication of encrypted content across different users. The result is denoted by M.

Next, the user creates a unique file identifier fid for M, e.g., by hashing the encrypted file, or by computing the root of the Merkle tree of the file [6]. The server checks if the corresponding file with identifier fid has already been stored by another tenant. Now, we distinguish between two cases: *(i)* the file has not been stored previously, and *(ii)* the file is already stored. In the latter case, the user will continue with the Store* protocol (see below).

We start by discussing the former case where the outsourced file has not been stored previously at S. Similar to [32], the file is interpreted as n blocks, each is s sectors long. A sector is an element of $\mathbb{Z}_p$ with p being the order of G and is denoted by m_{ij} with $1 \leq i \leq n$, $1 \leq j \leq s$. That is, the overall number of sectors in the file is $n \cdot s$. For each $1 \leq j \leq s$ a group element $u_j \in G$ is pseudo-randomly extracted from the file as $u_j = h_1(H(M)||j)$ where H denotes a cryptographic hash function.

Similar to the BLS scheme of [32], for each block index $i \in \{1, \ldots, n\}$ a file tag σ_i is computed as follows:

$$\sigma_i = \left(h_2(H(M)||i) \cdot \prod_{j=1}^{s} u_j^{m_{ij}} \right)^{\mathrm{sk}_\mathtt{U}} . \tag{5}$$

Observe that σ_i effectively represents a BLS signature under the key $\mathrm{sk}_\mathtt{U}$ of the following value:

$$w_i = h_2(H(M)||i) \cdot \prod_{j=1}^{s} u_j^{m_{ij}}, \tag{6}$$

The user uploads the file M, the public key $\mathrm{pk}_\mathtt{U}$ along with the proof of possession $\mathrm{pop}(\mathrm{sk}_\mathtt{U})$, and the file tags $\tau = (\sigma_1, \ldots, \sigma_n)$ to the provider S. The user stores locally $\mathrm{pk}_M = \mathrm{pk}_\mathtt{U}$ as the verification key for the file and the hash value $H(M)$ to be able to reconstruct the values $\{u_j\}_{j=1\ldots m}$ later on.

Upon receiving $(M, \mathrm{pk}_\mathtt{U}, \mathrm{pop}(\mathrm{sk}_\mathtt{U}), \tau)$, the provider S first verifies the validity of U's key using $\mathrm{pop}(\mathrm{sk}_\mathtt{U})$. Furthermore, S checks that the tags σ_i in τ are indeed valid

3. Recall that an accumulator is a cryptographic technique that allows to represent a set S of elements by a single value w, the witness, such that membership for any element $x \in S$ can be proved using w and an according membership proof.

for the key $\mathrm{pk}_{\mathtt{U}}$. A straightforward approach would be to check that each tag σ_i indeed represents a signature of w_i, incurring an effort to evaluate n pairings, which incurs considerable computational overhead on the provider. In SPORT, we exploit the fact that the signature scheme is homomorphic and verify all the signatures in a single batch verification. More precisely, S samples for each block index i a random exponent $r_i \in \mathbb{Z}_p^*$ and checks whether the following verification holds:

$$e\left(\prod_i \sigma_i^{r_i}, g\right) \stackrel{?}{=} e\left(\prod_i w_i^{r_i}, \mathrm{pk}_{\mathtt{U}}\right). \tag{7}$$

Observe that this reduces the effort from n bilinear pairings to n exponentiations and one pairing. In Section 4, we show that this verification considerably reduces the computational overhead borne by the provider in verifying the tags.

If the key and tags are correct, S sets $\mathrm{pk}_M := \mathrm{pk}_{\mathtt{U}}$, being the public key associated to a secret key $\mathrm{sk}_M := \mathrm{sk}_{\mathtt{U}}$. Moreover, the provider creates a log file, pklog, that will provide the necessary information needed for any other user to verify that the keys are well formed. Here, pklog is initialized with $\{(\mathrm{pk}_{\mathtt{U}}, \mathrm{pop}(\mathrm{sk}_{\mathtt{U}}))\}$. Following the notation from Section 2.5, the provider stores the file M, the metadata $\mathsf{Data}_{\mathrm{users}} = (\mathrm{pk}_M, \mathrm{pklog})$, and file metadata $\mathsf{Data}_{\mathrm{file}} = \tau$.

Specification of the Store^* Protocol

In case the file M is already stored and will be deduplicated, the Store^* protocol is executed in order to update the file tags τ on the fly, allowing the new user U to obtain guarantees from the POR protocol without the need to trust the correctness of the already stored tags (created by other tenants).

In Store^*, the user proceeds analogously to Store in order to compute the file M, its hash value $H(M)$, and the file tags $\tau = \{\sigma_i\}_{i=1\ldots n}$. Here, U also computes M and the hash value $H(M)$ and stores the same values as in Store. The main difference is that the user does not upload the file M but only $(\mathrm{pk}_{\mathtt{U}}, \mathrm{pop}(\mathrm{sk}_{\mathtt{U}}), \tau)$ to S.

Upon reception of $(\mathrm{pk}_{\mathtt{U}}, \mathrm{pop}(\mathrm{sk}_{\mathtt{U}}), \tau)$, S checks the validity of $\mathrm{pk}_{\mathtt{U}}$ using $\mathrm{pop}(\mathrm{sk}_{\mathtt{U}})$. Moreover, he verifies the correctness of the newly uploaded tags σ_i. In principle, one could apply the same verification given in Equation 7. However, this would require the provider to recompute or store the values w_i. To avoid this, we leverage the bilinear map to combine the two verifications (in the signature) and the fact that the already stored file tags σ_j^* are correct (as they have been previously validated by S).

Namely, let $\tau = (\sigma_i)_{1\le i\le n}$ denote the file tags uploaded by the user and $\tau^* = (\sigma_i^*)_{1\le i\le n}$ be the tags stored by the provider. SPORT applies cross-verification between the uploaded tags τ and the stored tags τ^*. More precisely, an uploaded tag σ_i is correct if and only if it holds that:

$$e\left(\sigma_i, \mathrm{pk}_M\right) \stackrel{?}{=} e\left(\sigma_i^*, \mathrm{pk}_{\mathtt{U}}\right), \tag{8}$$

Similar to the Store procedure, S does a batch cross-verification by sampling for each block index i a random exponent $r_i \in \mathbb{Z}_p^*$ and checks whether Equation 9 holds:

$$e\left(\prod_i \sigma_i^{r_i}, \mathrm{pk}_{\widetilde{M}}\right) \stackrel{?}{=} e\left(\prod_i (\sigma_i^*)^{r_i}, \mathrm{pk}_{\mathtt{U}}\right) \tag{9}$$

If the verification succeeds, S updates pk_M with $\mathrm{pk}_M \cdot \mathrm{pk}_{\mathtt{U}}$, and likewise the already stored tags σ_i^* by replacing them with $\sigma_i^* \cdot \sigma_i$. Due to the homomorphic properties of the BLS signature, the new tags will still be correct signatures under the updated key pk_M with respect to an updated secret key $\mathrm{sk}_M \rightarrow \mathrm{sk}_M + \mathrm{sk}_{\mathtt{U}}$. The metadata about the file, $\mathsf{Data}_{\mathrm{file}}$, is hence updated to the new tags τ^*—observe that the size of $\mathsf{Data}_{\mathrm{file}}$ remains unchanged. Moreover, the key pk_M is updated in $\mathsf{Data}_{\mathrm{users}}$ while the public key and the corresponding proof of possession of the new user $(\mathrm{pk}_{\mathtt{U}}, \mathrm{pop}(\mathrm{sk}_{\mathtt{U}}))$ are appended to pklog.

Finally, the provider sends to the user the log file pklog. Then, U proceeds to check the validity of each pair $(\mathrm{pk}', \mathrm{pop}') \in \mathrm{pklog}$ with $\mathrm{pk}' \neq \mathrm{pk}_{\mathtt{U}}$. In case all these keys pass the verification, U sets $\mathrm{pk}_M := \prod_{(\mathrm{pk}', \mathrm{pop}') \in \mathrm{pklog}} \mathrm{pk}'$.

The core idea here is that, at each point in time, the public key pk_M associated to a file M corresponds to a secret key $\sum_{\mathtt{U}} \mathrm{sk}_{\mathtt{U}}$, which captures the sum of all secret keys belonging to the users who uploaded M (see also the discussion in Section 3.3.2). This ensures that the secret key to pk_M is unknown to the provider as long as at least one user is honest (and hence kept his key secret). In addition, our construct easily allows a user to *unsubscribe* from M. To do so, a user with secret key $\mathrm{sk}_{\mathtt{U}}$ executes Store^* again using the secret key $-\mathrm{sk}_{\mathtt{U}}$ instead. This results into removing (or cancelling) $\mathrm{sk}_{\mathtt{U}}$ from sk_M.

Specification of the POR Protocol

To conduct POR on M for a user U, a verifier $\mathcal{V}$ (who could be the user himself) downloads the current verification key pk_M^* from the provider and compares it with the locally stored key pk_M.[4] If these are not equal, this signals that a new tenant has deduplicated the same content, and updated the corresponding file tags. In this case, $\mathcal{V}$ checks that pk_M^* has been correctly updated and includes information from his local key pk_M.

To this end, S sends to $\mathcal{V}$ all pairs $(\mathrm{pk}', \mathrm{pop}') \in \mathrm{pklog}$ of public keys and proof of possessions for currently registered users that joined since the last time that $\mathcal{V}$ or U executed a protocol with S with respect to M. Here, we assume that the keys are sorted in the order of the executions of $\mathsf{Store}/\mathsf{Store}^*$ by the users so that these values can be easily read out from pklog. We denote the set of these pairs by $\widehat{\mathrm{pklog}}$.

Clearly, this allows U to learn the number of users that uploaded the same file. We point out, however, that this information leakage can be prevented if, for example, S inserts a number of "bogus" keys within pklog; in this case, U can only learn an upper bound on the number of other users that store the same file. Observe that the knowledge

4. If the verifier is not the user, we assume that he received all values from the user before except of the secret key $\mathrm{sk}_{\mathtt{U}}$.

of the number of users deduplicating the same file can prove to be useful in a number of use-cases; for example, a number of studies show that such a level of transparency might motivate new pricing models in the cloud by allowing users to collaboratively share the costs of storing the same file [6].

Additionally, we point out that U might pseudorandomly create different signature keys for each file in order to prevent any entity to associate the user's public key with the stored files.

Starting from the locally stored verification key pk_M, the verifier $\mathcal{V}$ verifies $\mathrm{pk}_M \stackrel{?}{=} \mathrm{pk}_M^* \cdot \prod_{(\mathrm{pk}',\mathrm{pop}')\in\widehat{\mathrm{pklog}}} \mathrm{pk}'$. Furthermore, for each $(\mathrm{pk}', \mathrm{pop}') \in \widehat{\mathrm{pklog}}$, $\mathcal{V}$ checks the validity of the key pk' by verifying the proof of possession pop'. If the verification succeeds, $\mathcal{V}$ accepts the downloaded key pk_M^* as the verification key for M, updates the verification key pk_M^* locally to this value, and proceeds with the actual POR protocol as follows.

Subsequently, $\mathcal{V}$ creates a random challenge C and sends it to S. Here, C contains a random ℓ-element set of tuples (i, ν_i) where $i \in \{1, \ldots, n\}$ denotes a block index, and $\nu_i \stackrel{\mathbb{D}}{\leftarrow} \mathbb{Z}_N$ is a randomly generated integer. These values will be used to verify the retrievability of the data. Upon reception of C, S computes:

$$\sigma \leftarrow \prod_{(i,\nu_i)\in Q} \sigma_i^{\nu_i} \in G, \tag{10}$$

$$\mu_j \leftarrow \sum_{(i,\nu_i)\in Q} \nu_i m_{ij} \in \mathbb{Z}_p, \quad 1 \le j \le s. \tag{11}$$

Finally, these values are sent back to $\mathcal{V}$ who reconstructs u_i checks that:

$$e(\sigma, g) \stackrel{?}{=} e\left(\prod_{(i,\nu_i)\in C} h_2(H(M)||i)^{\nu_i} \cdot \prod_{j=1}^{s} u_j^{\mu_j}, \mathrm{pk}_M\right). \tag{12}$$

If this verification does not pass, $\mathcal{V}$ is certain that his file has been modified, and takes actions, such as informing the user (if different from $\mathcal{V}$) or downloading and repairing the file.

3.3. Correctness and Security

In this section, we show that MTPOR is correct according to Definition 2 and secure according to the soundness property of Definition 3.

3.3.1. Correctness. In the most simple case of exactly one user (which needs to be honest then), correctness follows directly from the correctness of the BLS-based POR of [32].

To address the multi-tenant case, we assume first that all involved users are honest. We say that a tag σ is *correctly formed* with respect to a key pair $(\mathrm{sk}, \mathrm{pk})$ if it fulfills Equation 5. Recall that correctly formed tags are essentially signatures of values that depend on message blocks, i.e., $\sigma_1 = w^{\mathrm{sk}_1}$, with respect to some verification key $\mathrm{pk}_1 = g^{\mathrm{sk}_1}$. This signature scheme is key-homomorphic in the following sense: given a second signature $\sigma_2 = w^{\mathrm{sk}_2}$ of the same value w with respect to $\mathrm{pk}_2 = g^{\mathrm{sk}_2}$, then $\sigma_1 \cdot \sigma_2 = w^{\mathrm{sk}_1+\mathrm{sk}_2}$ is a valid signature of w with respect to $g^{\mathrm{sk}_1+\mathrm{sk}_2} = \mathrm{pk}_1 \cdot \mathrm{pk}_2$. It follows by induction for any file M and for any sequence of key pairs $(\mathrm{sk}_i, \mathrm{pk}_i)$, $i = 1, \ldots, \ell$, that: $\perp \stackrel{(\mathrm{sk}_1,\mathrm{pk}_1,M)}{\longrightarrow} \ldots \stackrel{(\mathrm{sk}_\ell,\mathrm{pk}_\ell,M)}{\longrightarrow}$ Data implies that $\perp \stackrel{(\mathrm{sk}_1+\ldots+\mathrm{sk}_\ell,\mathrm{pk}_1\cdots\mathrm{pk}_\ell,M)}{\longrightarrow}$ Data. Hence, according to the correctness of BLS POR of [32], it follows that POR-protocol accepts with $\mathrm{vk} = \mathrm{pk}_1 \cdots \mathrm{pk}_\ell$ if the tags are correctly formed with respect to $(\mathrm{sk}^*, \mathrm{pk}^*) := (\mathrm{sk}_1 + \ldots + \mathrm{sk}_\ell, \mathrm{pk}_1 \cdots \mathrm{pk}_\ell)$. This shows correctness for the case that all users behaved honestly.

Next, assume that some of the users are malicious (i.e., they deviate from the protocols). Let $(\mathrm{pk}_i, \mathrm{pop}_i, \tau_i)$ denote the values uploaded by the users during Store/Store*. Observe that G is a cyclic group and hence any value pk_i can be expressed as g^{sk_i} for some value $\mathrm{sk}_i \in \mathbb{Z}_p$. This shows that pk_i represents a valid verification key. Moreover, the tag verifications executed by the provider (see Equations 7 and 9) ensure that the tags stored in Data are correctly formed. To see why, assume that a user uploads malicious tags $\sigma_i^* \neq \sigma_i$ for $i \in I \subseteq \{1, \ldots, n\}$. As the user does not know the values $\{r_i\}_{i\in I}$, Equations 7 and 9 would be violated with high probability and hence not be accepted by the honest provider. We now give an explanation for Equation 7 (which also holds for Equation 9). Assume that the following holds:

$$e\left(\prod_i (\sigma_i^*)^{r_i}, g\right) = e\left(\prod_i w_i^{r_i}, \mathrm{pk}_\mathsf{U}\right) = e\left(\prod_i \left(w_i^{\mathrm{sk}_\mathsf{U}}\right)^{r_i}, g\right).$$

It follows that:

$$\prod_i (\sigma_i^*)^{r_i} = \prod_i \left(w_i^{\mathrm{sk}_\mathsf{U}}\right)^{r_i} \Leftrightarrow \prod_{i\in I} \left(\underbrace{\sigma_i^* \cdot \left(w_i^{\mathrm{sk}_\mathsf{U}}\right)^{-1}}_{\neq 1}\right)^{r_i} = 1.$$

Since the values r_i are sampled uniformly at random and independently, the probability for this event is at most $1/|G|$.

Finally, the correctness of the proofs of possession pop_i are each individually tested according to the specifications. This shows that all values uploaded by the users need to be correctly built as they would be otherwise rejected with high probability by the provider.

3.3.2. Security. We now prove that SPORT is ε-sound (cf. Definition 3). Recall that this expresses security with respect to a static attacker. This means that the attacker may collude with corrupted parties which are corrupted right from the start. That is, whenever an attacker creates a new honest user by a Store-query during the challenge game (cf. Section 2.4), it will not be corrupted later on.

Our proof follows in principle the line of arguments given for the BLS POR in [33]. There, the proof was divided into three parts; the first part shows that if a verifier accepts a response to a query, this response was constructed correctly with overwhelming probability. The other two

parts prove that if correctly built responses are given, one can reconstruct a constant fraction of the blocks of the file (part two) and that with the help of erasure codes, one can derive the full original file from these blocks (part three). As the structure of the response in SPORT is exactly the same as in BLS POR and as we likewise suggest the use of an appropriate erasure code, the arguments given for parts two and three in [33] apply directly to SPORT as well. Consequently, it is sufficient to show the first part, i.e., that if a verifier $\mathcal{V}$ accepts a response, it has to be correctly built with overwhelming probability.

Recall that the CDH refers to the problem of deriving the value h^a where a is unknown from a given triple $(g, g^a, h) \in G^3$ and is usually assumed to be hard. We prove the following theorem:

Theorem 1. If the computational Diffie-Hellman problem (CDH) is hard in the bilinear group G, then an adversary in the random oracle model can only cause a verifier $\mathcal{V}$ to accept a POR-instance by responding with correctly computed values $\{\mu_j\}$ and σ—except with negligible probability.

We conduct a sequence of four games adapted from [33].We start with Game 0 which is simply the challenge game defined in Section 2.4. Game 1 is the same as Game 0 except of the following difference. When the adversary eventually outputs a prover P for a user U and a file M, he has to hand to the environment a value sk_i such that $g^{\mathrm{sk}_i} = \mathrm{pk}_i$, for each pair $(\mathrm{pk}_i, \mathrm{pop}_i)$ communicated to $\mathcal{V}$ in the context of M. Owing to the security of the proofs of possession, the adversary has to know these values since, otherwise, the verification of pop_i would fail with high probability. As the environment performs the POR-protocols honestly on behalf of the user $\mathcal{V}$ (i.e., without using this additional knowledge), this does not impact the probability that the verifier accepts. Hence, we assert that the success probability of the attacker does not change.

Game 2 is the same as Game 1 with the difference that the environment keeps a list of all messages communicated within the challenge game. If the adversary is successful in any of the POR instances (i.e., the verifier accepts, but the adversary's signature σ' is not equal to $\sigma = \prod_{(i,\nu_i)\in Q} \sigma_i^{\nu_i}$, see Equation 10), the environment aborts the security game. Let M denote the file considered within this POR-instance and pk_M the used public verification key. As explained above, it is ensured that $\mathrm{pk}_M = g^{a+b}$ for some secret value a (which is the secret key of the user represented by $\mathcal{V}$) and a known value b (being the aggregation of all remaining keys).

Similar to [33], one can now construct from the adversary a simulator that simulates the honest users and the random oracle for the adversary and that solves the CDH for a given triplet $(g, g^a, h) \in G^3$ by using g^a as the public key of simulated honest users. However, since we have to cater for multiple users, one cannot use the same value g^a each time as the public key. Here, we exploit the fact that the tags are signatures and that these signatures are homomorphic. More precisely, whenever a new honest user U needs to be created (in terms of a Setup-query), the simulator chooses a random value $r_{\mathtt{U}} \in \mathbb{Z}_p$ and outputs $\mathrm{pk}_{\mathtt{U}} = g^a \cdot g^{r_{\mathtt{U}}} = g^{a+r_{\mathtt{U}}}$.

Next, the simulator has to compute the following signatures:

$$\left(h_2(H(M)||i) \cdot \prod_{j=1}^{s} u_j^{m_{ij}}\right)^{a+r_{\mathtt{U}}}. \tag{13}$$

Since $r_{\mathtt{U}}$ is known to the simulator and given that the signatures can be homomorphically combined, this task boils down to compute $\left(h_2(H(M)||i) \cdot \prod_{j=1}^{s} u_j^{m_{ij}}\right)^{a}$ which needs to be performed only by the simulator. This is accomplished as follows. First, a random exponent $\rho_i \in \mathbb{Z}_p^*$ is chosen. Since the values u_j and m_{ij} are known and assuming the random oracle model, the simulator programs the random model such that:

$$h_2(H(M)||i) := g^{\rho_i} \cdot \left(\prod_{j=1}^{s} u_j^{m_{ij}}\right)^{-1}. \tag{14}$$

This implies that the signature is equal to $(g^a)^{\rho_i}$ which can be computed since ρ_i is known and g^a part of the given CDH triple.

A further change affects how the values u_j are computed. By definition, it holds that $u_j = h_1(H(M)||j)$. Again, the simulator exploits that he simulates the random oracle and sets $h_1(H(M)||j) := g^{\alpha_j} \cdot h^{\beta_j}$ for some randomly chosen values α_j, β_j.

Now, if $(\sigma', \{\mu_j'\})$ denotes the response given by the adversary and $(\sigma, \{\mu_j\})$ the correctly-built response, then one can deduce from that fact that $\sigma \neq \sigma'$ that there exists at least one index j such that $\mu_j' \neq \mu_j$. Let $\Delta_j := \mu_j' - \mu_j$ for each j. We obtain $e(\sigma' \cdot \sigma^{-1}, g) = e(\prod_j u_j^{\Delta_j}, g)$, which can be rewritten to:

$$e(\sigma' \cdot \sigma^{-1} \cdot (g^a)^{-\sum_j \alpha_j \cdot \Delta_j}, g) = e(h, g^{a+b+r_{\mathtt{U}}})^{\sum_j \beta_j \cdot \Delta_j}, \tag{15}$$

where $g^{a+b+r_{\mathtt{U}}}$ represents the current combined public key. Note that $\sum_j \beta_j \cdot \Delta_j \neq 0$ with high probability as the coefficients β_j have been chosen uniformly at random and independently. It follows that:

$$h^a = \left(\sigma' \cdot \sigma^{-1} \cdot (g^a)^{-\sum_j \alpha_j \cdot \Delta_j}\right)^{\left(\sum_j \beta_j \cdot \Delta_j\right)^{-1}} \cdot h^{-b-r_{\mathtt{U}}}$$

is a solution to the given CDH instance. Recall that $r_{\mathtt{U}}$ has been chosen by the simulator and b is given to the simulator due to the change in Game 1.

As an additional step in our scheme, the proof of possession pop for the public key $\mathrm{pk}_{\mathtt{U}} = g^{a+r_{\mathtt{U}}}$ needs to be provided. As the exponent a is unknown, this cannot be directly computed by the simulator. However, recall that $\mathrm{pop} = h_0(\langle \mathrm{pk}_{\mathtt{U}} \rangle)^{\mathrm{sk}_{\mathtt{U}}}$ where $\mathrm{sk}_{\mathtt{U}} = a + r_{\mathtt{U}}$. Therefore, the simulator chooses a random integer $\rho \in \mathbb{Z}_p$ and computes g^ρ and $\mathrm{pk}_{\mathtt{U}}^\rho$. Obviously, the second value is a valid signature of the first. Hence, we re-use the trick of reprogramming the hash and simply set $h_0(\langle \mathrm{pk}_{\mathtt{U}} \rangle)$ to g^ρ and pop to $\mathrm{pk}_{\mathtt{U}}^\rho$.

The final Game 3 is the same as Game 2 with a single difference. Similar to before, the environment tracks

all messages and observes all POR-instances. We know already with overwhelming probability, that for all responses $(\sigma, \{\mu_i\})$ which are accepted, it holds that σ is correctly computed. Now, the environment checks for all accepted responses whether the aggregated messages μ_i are equal to correctly generated, i.e., are equal to $\sum_{(i,\nu_i)\in Q} \nu_i m_{ij} \in \mathbb{Z}_p, \quad 1 \leq j \leq s$ (see also Equation 11). If this is not the case, the environment aborts the security game. Similar to [33] and the explanations given for Game 2, one can show that this can be used to solve the discrete logarithm problem in G—which would contradict the assumption that this problem is hard in G.

The core idea is the following. Given $g, h \in G$, the simulator honestly generates all values with the exception of the values u_j that are again set to $g^{\alpha_j} \cdot h^{\beta_j}$. Now, assume that the adversary succeeds with a response $(\sigma', \{\mu'_i\})$ instead of the correct response $(\sigma, \{\mu_i\})$. We know already that $\sigma = \sigma'$ needs to hold with high probability. Assume that $\Delta_j := \mu'_j - \mu_j$ for all j. From the fact that both responses are accepted, it follows that $\prod_j u_j^{\mu'_j} = \prod_j u_j^{\mu_j}$, which implies the following:

$$1 = \prod_j u_j^{\Delta_j} = g^{\sum_j \alpha_j \cdot \Delta_j} \cdot h^{\sum_j \beta_j \cdot \Delta_j}. \tag{16}$$

Again, it holds that $\sum_j \beta_j \cdot \Delta_j \neq 0$ with high probability. In such case, we solve the discrete logarithm problem by

$$h = \left(g^{\sum_j \alpha_j \cdot \Delta_j}\right)^{\left(\sum_j \beta_j \cdot \Delta_j\right)^{-1}}. \tag{17}$$

Thus, we conclude that the existence of such an attacker would contradict the hardness assumption with respect to the discrete logarithm problem.

Summing up, a prover can only present responses that are accepted by the verifier if all values are correctly built, i.e., as displayed in Equations 10 and 11—thus concluding the proof of Theorem 1.

4. Implementation & Evaluation

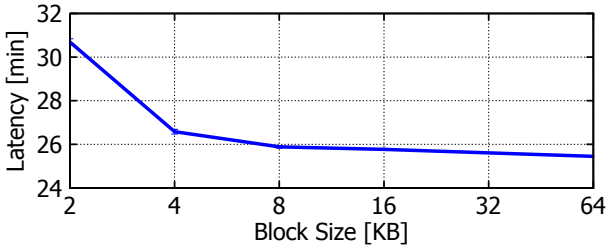
In this section, we evaluate an implementation of SPORT within a realistic cloud setting and we compare the performance of SPORT to the RSA-based and BLS-based POR schemes due to [32]. Note that all three POR schemes are publicly-verifiable.

4.1. Implementation Setup

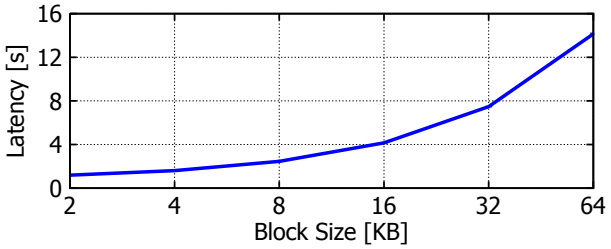
We implemented a prototype of SPORT in Java. In our implementation, we relied on the Backblaze library [3] for constructing the erasure codes (instantiated using Reed-Solomon coding). For a baseline comparison, we also implemented the public POR (with its two BLS and RSA variants) schemes (see Appendix A for a description of these POR schemes). Here, we relied on SHA-256, the Java built-in random number generator, HMAC based on SHA-256, 2048-bit RSA modulus, and the JPBC library [2] (based on the PBC cryptographic library [1]) to implement

Parameter	Default Value
File size	64 MB
Group size |p|	224 bits
RSA modulus size	2048 bits
Elliptic Curve (BLS)	PBC Library Curve F
Challenge size ℓ	100

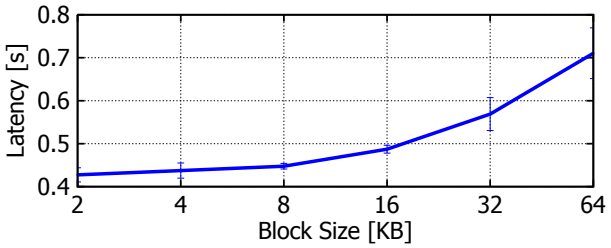
TABLE 1. DEFAULT PARAMETERS USED IN EVALUATION.



(a) Latency in store of SPORT w.r.t. to the block size.



(b) User verification latency in the Proof Verification procedure of SPORT w.r.t. to the block size.



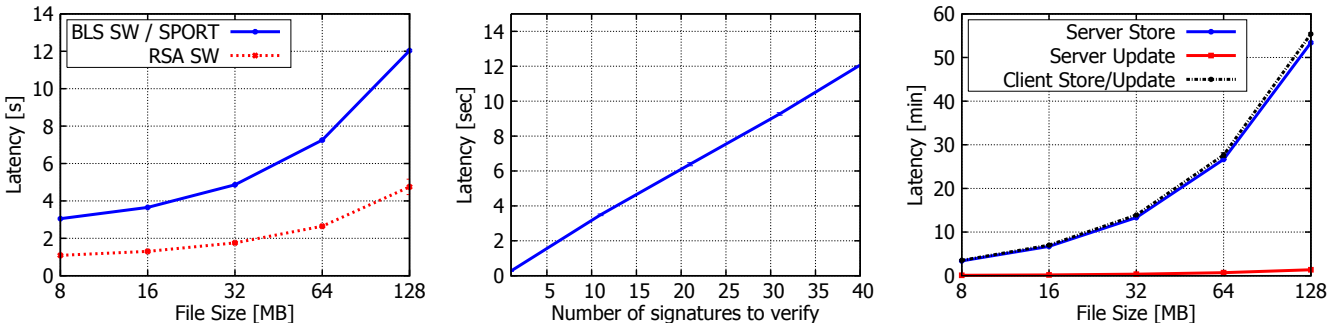
(c) Server proof generation latency in SPORT w.r.t. to the block size.

Figure 2. Performance of SPORT with respect to the block size.

BLS signatures. Table 1 summarizes the default parameters assumed in our setup.

We deployed our implementations on a private network consisting of two 24-core Intel Xeon E5-2640 with 32GB of RAM. In our network, the communication between various machines was bridged using a 100 Mbps switch. The storage server was running on one of the 24-core Xeon E5-2640 machine, whereas the clients were co-located on the second 24-core Xeon E5-2640 machine; this ensures a fair comparison between the overhead incurred on the users and on the server for the different schemes.

To emulate a realistic Wide Area Network (WAN), we shape all traffic exchanged on the networking interfaces with a Pareto distribution with a mean of 20 ms and a variance of 4 ms [19]. In our setup, each client invokes an operation in a closed loop, i.e., a client may have at most one pending



(a) Latency incurred in the verification of POR by the users w.r.t. the file Size.

(b) Latency incurred in updating users' private verification keys in SPORT with respect to the number of tenants deduplicating the same file.

(c) Latency incurred in Store* and Store on the provider w.r.t. the file Size.

Figure 3. Performance evaluation of SPORT in comparison to the POR-SW schemes of [32]. Each data point in our plots is averaged over 10 independent runs. Notice that, due to their small size, 95% confidence intervals are omitted from these plots.

operation. Prior to the setup phase, each client disperses his files with a (9,12) code. Similar to [5], [8], we assume that clients query for the availability of $\ell = 100$ randomly selected blocks in the POR challenge phase.

When implementing SPORT, we spawned multiple threads on the cloud machine, each thread corresponding to a unique audit performed on behalf of a client. Each data point in our plots is averaged over 10 independent measurements; where appropriate, we include the corresponding 95% confidence intervals.

4.2. Evaluation Results

Before evaluating the performance of SPORT, we start by analyzing the impact of the block size on the latencies incurred in the verification of POR in the BLS POR scheme of [32]. Our results (cf. Figure 2) show that modest block sizes of 8 KB yield the most balanced performance, on average, across all procedures. Throughout the rest of our evaluation, we therefore set the block size to 8 KB.

POR protocol performance: In Figure 3(a), we evaluate the time required by the user to verify a single POR in SPORT, when compared to the public SW schemes. As expected, our findings show that users in SPORT and the BLS SW scheme witness comparable performance since the (core) POR verification process in both these schemes are similar. One major difference between SPORT and the BLS SW scheme of [32] is that users need to seldomly update the private key in order to be able to verify the POR responses issued by the provider. This only occurs whenever a new tenant deduplicates the same file of interest in between two consecutive POR performed by users storing the same file (cf. Section 3.2). As shown in Figure 3(b), this process incurs negligible overhead on users; for example, users can update their private keys based on the contributions of 10 additional tenants in around 3 seconds.

Figure 3(a) also shows that the latency witnessed in the POR verification process in SPORT and the BLS-SW scheme is almost double when compared to that of the RSA-SW scheme. This is the case due to the fact that *(i)* pairing operations (to verify tags) are considerably more expensive than RSA exponentiations, and *(ii)* BLS-based schemes result in a larger number of sectors per block, which incurs additional computations in the store, verify, and prove procedures.

That said, the fact that SPORT/BLS-SW exhibits more sectors per block allows the cloud provider to efficiently parallelize the computations required to answer a given POR challenge. In Figure 4, we evaluate the latency with respect to throughout exhibited by the provider in SPORT when issuing POR responses (i.e., when computing Equation 11). We measure throughput as follows: we require that each client performs back to back POR verification operations; we then increase the number of clients in the system until the aggregated throughput attained by all clients is saturated. The peak throughput is then computed as the maximum aggregated number of POR responses that can be issued by the storage server within a period of time. Our results show that the provider can perform up to 3700 POR responses per SPORT within 50 seconds, resulting in a peak throughput 74 operations (POR responses) per second. In the RSA-SW scheme, a maximum of 500 POR responses can be performed within the same time lapse, resulting in a modest peak throughout of 10 operations per second. In other words, the provider can scale more than 7 times better in SPORT when compared to the RSA-SW scheme of [32].

Recall that RSA-based tags are considerably larger in size when compared to their BLS counterparts (almost 5 times larger). Moreover, RSA does not exhibit key homomorphism and therefore does not allow tag sharing. As shown in Figure 1, this incurs considerable storage overhead on the provider when multiple tenants deduplicate the same file. For example, assuming a file size of 64 MB, storing the tags pertaining to 5 tenants in RSA-SW results in an additional overhead of 10 MB; this overhead reduces to only 0.2 MB in SPORT.

Update protocol performance: In Figure 3(c), we evaluate the time incurred on the provider when verifying the ten-

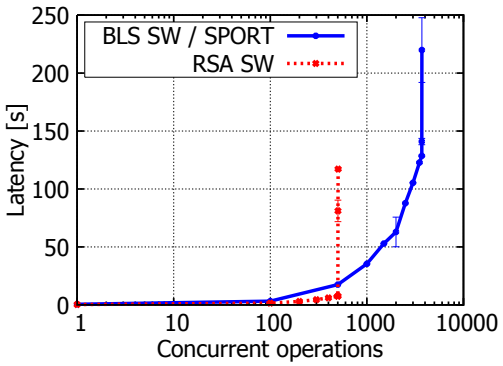


Figure 4. Latency incurred in POR w.r.t. the throughput as witnessed by the provider.

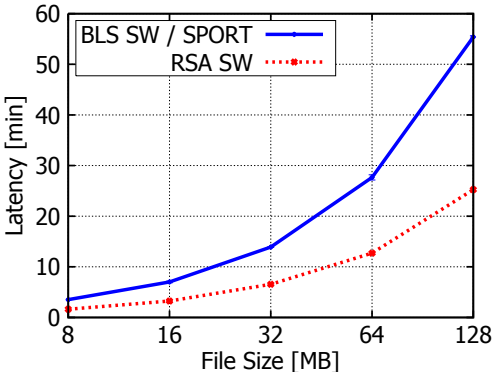


Figure 5. Latency incurred in Store w.r.t. the file size.

ant's tags in the store and update procedures. Here, we also recall the time incurred on the tenants in the store procedure for a baseline comparison. Our results show that the overhead of store (in verifying the first tags uploaded for a given file) incurred on the provider slightly improves that incurred on the tenants in spite of the fact that BLS signature generation (performed by clients) is considerably faster than the signature verification process (performed by the cloud). This is the case since SPORT leverages the homomorphism of the underlying signature scheme in order to verify all the tags with a single batch verification (cf. Section 3.2). As a result of this optimization, the overhead on the provider in this case is approximately 4% lower than that on the tenants.

Notice that this overhead is even further reduced in the update procedure whenever an additional tenant shares the same file/tags. In this case, the provider does not need to partially create the tags (i.e., to reconstruct v_i as given by Equation 6). Instead, S only needs to verify that Equation 9 holds. This verification incurs negligible overhead on S; as shown in Figure 3(c), this process is almost 40 times faster than that required to construct the tags by the tenants.

Recall that the *update* protocol is performed only once by clients (upon upload of a file that is already stored). By doing so, SPORT trades the *permanent* storage costs associated with the additional storage of file tags, with the *one-time* use of their computational resources to update the tags—thus reducing the total costs borne on users.

Store protocol performance: In Figure 5, we measure the latency incurred in the store procedure in SPORT compared to the RSA-SW and BLS-SW schemes of [32]. As expected, the latency incurred in store increases almost linearly with the file size in all POR schemes. Our results show that the store latency incurred in SPORT is the same as that incurred in the BLS SW of [32]. Indeed, both schemes involve the same procedure for creating tags based on BLS signatures; in this respect, SPORT does not incur any additional overhead on the user when compared to the basic BLS SW store procedure. On the other hand, although BLS signatures are faster to generate by the user when compared to their RSA counterpart, BLS-based tags result in considerably smaller sector sizes than the RSA-based scheme, which results in a larger number of sectors per block, and thus in additional computations to create each block tag. This explains the reason why the store latency exhibited by the users of SPORT is almost twice as much larger than that of the RSA-SW protocol. We, however, point out that the store procedure is only performed once per user—after which users simply execute the POR protocol to verify the integrity of their outsourced data.

5. Related Work

In what follows, we briefly overview related work in the area.

Single-tenant POR: Juels and Kaliski [26] introduce a single-tenant POR scheme, which relies on indistinguishable blocks, hidden among regular file blocks in order to detect data modification by the server. This proposal only supports a bounded number of POR queries, after which the storage server can learn all the embedded sentinels. The authors also propose a Merkle-tree construction for constructing public POR, which can be verified by any external party without the need of a secret key. Bowers *et al.* [15] propose various improvements to the original single-tenant POR in [26], which tolerates a Byzantine adversarial model. Shacham and Waters [32] propose private-key-based and public-key-based (single-tenant) POR schemes which utilize homomorphic authenticators to yield compact proofs.

In [8], Ateniese *et al.* introduce a variant of POR called *proofs of data possession* (PDP). It supports an unbounded number of challenge queries and enables public verifiability of the PDP. Unlike other POR schemes, this instantiation does not offer extractability guarantees. This proposal was later extended in [9] to address dynamic writes/updates from the clients. Cash *et al.* [17] propose a dynamic POR scheme which relies on oblivious RAM protocols. In [35], Shi *et al.* propose a dynamic POR scheme that considerably improves the performance of [17] by relying on a Merkle hash tree. Other contributions propose the notion of delegable verifiability of POR; for instance, in [29], [34], the authors describe schemes that enable the user to delegate the verification of POR and to prevent their further re-delegation. In [7],

Armknecht *et al.* introduce the notion of outsourced proofs of retrievability, an extension of the POR model, in which users can task an external auditor to perform and verify POR on their behalf with the cloud provider.

Secure Data De-duplication in Multi-tenant Settings: In [24], Harnik *et al.* describe a number of threats posed by client-side data deduplication, in which an adversary can learn if a file is already stored in a particular cloud by guessing the hashes of predictable messages. This leakage can be countered using Proofs of Ownership schemes (PoW) [18], [23], which enable a client to prove it possesses the file in its entirety. PoW are inspired by Proofs of Retrievability and Data Possession (POR/PDP) schemes [8], [32], with the difference that PoW do not have a pre-processing step at setup time. Halevi *et al.* [23] propose a PoW construct based on Merkle trees which incurs low overhead on the server in constructing and verifying PoW. Xu *et al.* [36] build upon the PoW of [23] to construct a PoW scheme that supports client-side deduplication in a bounded leakage setting. Di Pietro and Sorniotti [18] propose a PoW scheme which reduces the communication complexity of [23] at the expense of additional server computational overhead. Blasco *et al.* [12] propose a PoW based on Bloom filters which further reduces the server-side overhead of [18].

Douceur *et al.* [20] introduced the notion of convergent encryption, a type of deterministic encryption in which a message is encrypted using a key derived from the plaintext itself. Convergent encryption is not semantically secure [11] and only offers confidentiality for messages whose content is unpredictable. To remedy this, a number of proposals introduce the notion of oblivious server-aided encryption to perform data deduplication scheme [6], [10]; here, the encryption key is obliviously computed based on the hash of the file and the private key of the assisting server.

In [37], Zheng and Xu propose the notion of Proof of Storage with Deduplication (POSD). Here, POSD is introduced as the combination of PDP/POR and PoW schemes. The authors claim that publicly-verifiable POR/PDP can be inherently used to verify the integrity of deduplicated files (since the verification can be performed by any entity). As we mention in this paper, such schemes do not resist collusion between malicious tenants and the cloud provider; namely, a cloud provider which has access to the secret material (e.g., leaked by the first user who created) can always construct correct responses to the challenges issued by the clients—even if the cloud provider deletes the oustourced data. As far as we are aware, SPORT is the first secure POR instantiation which addresses multi-tenancy and data deduplication.

6. Conclusion

We introduced the notion of multi-tenant proofs of retrievability (MTPOR), an extension of the traditional single-tenant POR concept, and proposed an efficient instantiation dubbed SPORT. We implemented a prototype based on SPORT, and evaluated its performance in a realistic cloud setting. Our results show that our proposal incurs minimal storage overhead on the cloud provider without deteriorating the performance witnessed by tenants and the cloud provider when compared to existing publicly-verifiable POR schemes. SPORT is provably secure in the random oracle model under the computational Diffie-Hellman problem, assuming static corruptions. In this respect, we see the analysis of the security of SPORT under adaptive corruption as an interesting open question.

We argue that SPORT provides an important stepping stone to reconciliate existing cloud integrity and security primitives with functional requirements (such as resource sharing and multi-tenancy) in the cloud. Namely, SPORT provides considerable incentives *(i)* for end-users to obtain guarantees about the retrievability of their files in the cloud, and *(ii)* for cloud providers to offer differentiated services while preserving the efficiency of their storage system.

7. Acknowledgements

This work was partly supported by the TREDISEC project (G.A. no 644412), funded by the European Union (EU) under the Information and Communication Technologies (ICT) theme of the Horizon 2020 (H2020) research and innovation programme.

References

[1] PBC Library. http://crypto.stanford.edu/pbc/, 2007.

[2] JPBC:Java Pairing-Based Cryptography Library. http://gas.dia.unisa.it/projects/jpbc/#.U3HBFfna5cY, 2013.

[3] Backblaze Open Sources Reed-Solomon Erasure Coding Source Code. https://www.backblaze.com/blog/reed-solomon/, 2015.

[4] Google loses data after lightning strikes. http://money.cnn.com/2015/08/19/technology/google-data-loss-lightning/, 2015.

[5] F. Armknecht, J. Bohli, G. O. Karame, Z. Liu, and C. A. Reuter. Outsourced proofs of retrievability. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 831–843, 2014.

[6] F. Armknecht, J. Bohli, G. O. Karame, and F. Youssef. Transparent data deduplication in the cloud. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 886–900, 2015.

[7] F. Armknecht, J.-M. Bohli, G. O. Karame, Z. Liu, and C. A. Reuter. Outsourced proofs of retrievability. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 831–843, New York, NY, USA, 2014. ACM.

[8] G. Ateniese, R. C. Burns, R. Curtmola, J. Herring, L. Kissner, Z. N. J. Peterson, and D. X. Song. Provable data possession at untrusted stores. In *ACM Conference on Computer and Communications Security*, pages 598–609, 2007.

[9] G. Ateniese, R. D. Pietro, L. V. Mancini, and G. Tsudik. Scalable and efficient provable data possession. *IACR Cryptology ePrint Archive*, 2008:114, 2008.

[10] M. Bellare, S. Keelveedhi, and T. Ristenpart. DupLESS: Server-aided encryption for deduplicated storage. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 179–194, Berkeley, CA, USA, 2013. USENIX Association.

[11] M. Bellare, S. Keelveedhi, and T. Ristenpart. Message-locked encryption and secure deduplication. In Johansson and Nguyen [25], pages 296–312.

[12] J. Blasco, R. Di Pietro, A. Orfila, and A. Sorniotti. A tunable proof of ownership scheme for deduplication using bloom filters. In *Communications and Network Security (CNS), 2014 IEEE Conference on*, pages 481–489, Oct 2014.

[13] D. Boneh, B. Lynn, and H. Shacham. Short signatures from the weil pairing. *J. Cryptology*, 17(4):297–319, 2004.

[14] K. D. Bowers, A. Juels, and A. Oprea. HAIL: a high-availability and integrity layer for cloud storage. In *ACM Conference on Computer and Communications Security*, pages 187–198, 2009.

[15] K. D. Bowers, A. Juels, and A. Oprea. Proofs of retrievability: theory and implementation. In *CCSW*, pages 43–54, 2009.

[16] K. D. Bowers, M. van Dijk, A. Juels, A. Oprea, and R. L. Rivest. How to tell if your cloud files are vulnerable to drive crashes. In *ACM Conference on Computer and Communications Security*, pages 501–514, 2011.

[17] D. Cash, A. Küpçü, and D. Wichs. Dynamic Proofs of Retrievability via Oblivious RAM. In Johansson and Nguyen [25], pages 279–295.

[18] R. Di Pietro and A. Sorniotti. Boosting efficiency and security in proof of ownership for deduplication. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '12, pages 81–82, New York, NY, USA, 2012. ACM.

[19] D. Dobre, G. Karame, W. Li, M. Majuntke, N. Suri, and M. Vukolić. Powerstore: Proofs of writing for efficient and robust storage. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 285–298, New York, NY, USA, 2013. ACM.

[20] J. R. Douceur, A. Adya, W. J. Bolosky, D. Simon, and M. Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In *ICDCS*, pages 617–624, 2002.

[21] E. Ghosh, O. Ohrimenko, D. Papadopoulos, R. Tamassia, and N. Triandopoulos. Zero-knowledge accumulators and set operations. Cryptology ePrint Archive, Report 2015/404, 2015. http://eprint.iacr.org/2015/404.

[22] T. T. W. Group. The notorious nine: Cloud computing top threats in 2013. Report, Cloud Security Alliance, February 2013.

[23] S. Halevi, D. Harnik, B. Pinkas, and A. Shulman-Peleg. Proofs of ownership in remote storage systems. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 491–500, New York, NY, USA, 2011. ACM.

[24] D. Harnik, B. Pinkas, and A. Shulman-Peleg. Side channels in cloud services: Deduplication in cloud storage. *IEEE Security & Privacy*, 8(6):40–47, 2010.

[25] T. Johansson and P. Q. Nguyen, editors. *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*, volume 7881 of *Lecture Notes in Computer Science*. Springer, 2013.

[26] A. Juels and B. S. K. Jr. PORs: Proofs Of Retrievability for Large Files. In *ACM Conference on Computer and Communications Security*, pages 584–597, 2007.

[27] D. T. Meyer and W. J. Bolosky. A study of practical deduplication. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, pages 1–1, Berkeley, CA, USA, 2011. USENIX Association.

[28] D. T. Meyer and W. J. Bolosky. A study of practical deduplication. *Trans. Storage*, 7(4):14:1–14:20, Feb. 2012.

[29] Y. Ren, J. Xu, J. Wang, and J.-U. Kim. Designated-verifier provable data possession in public cloud storage. *International Journal of Security and Its Applications*, 7(6):11–20, 2013.

[30] L. Reyzin and S. Yakoubov. Efficient asynchronous accumulators for distributed pki. Cryptology ePrint Archive, Report 2015/718, 2015. http://eprint.iacr.org/2015/718.

[31] T. Ristenpart and S. Yilek. The power of proofs-of-possession: Securing multiparty signatures against rogue-key attacks. In *Advances in Cryptology - EUROCRYPT 2007, 26th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Barcelona, Spain, May 20-24, 2007, Proceedings*, volume 4515 of *Lecture Notes in Computer Science*, pages 228–245. Springer, 2007.

[32] H. Shacham and B. Waters. Compact Proofs of Retrievability. In *ASIACRYPT*, pages 90–107, 2008.

[33] H. Shacham and B. Waters. Compact Proofs of Retrievability. Cryptology ePrint Archive, Report 2008/073, 2008. http://eprint.iacr.org/.

[34] S.-T. Shen and W.-G. Tzeng. Delegable provable data possession for remote data in the clouds. In S. Qing, W. Susilo, G. Wang, and D. Liu, editors, *ICICS*, volume 7043 of *Lecture Notes in Computer Science*, pages 93–111. Springer, 2011.

[35] E. Shi, E. Stefanov, and C. Papamanthou. Practical dynamic proofs of retrievability. In A.-R. Sadeghi, V. D. Gligor, and M. Yung, editors, *ACM Conference on Computer and Communications Security*, pages 325–336. ACM, 2013.

[36] J. Xu, E.-C. Chang, and J. Zhou. Weak leakage-resilient client-side deduplication of encrypted data in cloud storage. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13, pages 195–206, New York, NY, USA, 2013. ACM.

[37] Q. Zheng and S. Xu. Secure and efficient proof of storage with deduplication. In *Proceedings of the Second ACM Conference on Data and Application Security and Privacy*, CODASPY '12, pages 1–12, New York, NY, USA, 2012. ACM.

Appendix A. Public SW POR

To enable any entity which does not necessarily possess secrets to verify a POR, Shacham and Waters [32] propose two publicly verifiable POR schemes based on BLS signatures [13] and RSA, respectively.

Public BLS SW Scheme: The setup phase requires choosing a group G with support $\mathbb{Z}_p$, and a computable bilinear map $e : G \times G \to G_T$. Additionally, the user chooses a private key $x \in \mathbb{Z}_p$, the corresponding public key $v = g^x \in G$ along with another generator $u \in G$. In the storage phase, a signature on each block i is computed $\sigma_i = \left(H(i)\prod_{j=1}^{s} u_j^{m_{ij}}\right)^x$. For verification, the challenge query Q is generated similarly to PSW and sent to the prover who computes:

$$\sigma \leftarrow \prod_{(i,\nu_i)\in Q} \sigma_i^{\nu_i} \in G, \quad \mu_j \leftarrow \sum_{(i,\nu_i)\in Q} \nu_i m_{ij} \in \mathbb{Z}_p.$$

These values are sent to the verifier who checks that:

$$e(\sigma, g) \stackrel{?}{=} e\left(\prod_{(i,\nu_i)\in Q} H(i)^{\nu_i} \cdot \prod_{j=1}^{s} u_j^{\mu_j}, v\right). \quad (18)$$

Public RSA SW Scheme: The public RSA-based SW scheme is similar to its public counterpart. Here, the block authenticator can be computed by $\sigma_i = (H(i)u^{m_i})^d \bmod N$, where d is the private key of the user. The cloud response is calculated similarly to

the public BLS SW scheme. Given the public RSA key e, the verification unfolds as follows:

$$\sigma^e \stackrel{?}{=} \prod_{(i,\nu_i)\in Q} H(i)^{\nu_i} u^{\mu} \bmod N.$$