Outsourced Proofs of Retrievability

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

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

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

Zongren Liu
NEC Laboratories Europe, Germany
zongren.liu@neclab.eu

Christian A. Reuter
University of Mannheim, Germany
reuter@uni-mannheim.de

ABSTRACT

Proofs of Retrievability (POR) are cryptographic proofs that enable a cloud provider to prove that a user can retrieve his file in its entirety. POR need to be frequently executed by the user to ensure that their files stored on the cloud can be fully retrieved at any point in time. To conduct and verify POR, users need to be equipped with devices that have network access, and that can tolerate the (non-negligible) computational overhead incurred by the verification process. This clearly hinders the large-scale adoption of POR by cloud users, since many users increasingly rely on portable devices that have limited computational capacity, or might not always have network access.

In this paper, we introduce the notion of *outsourced proofs of retrievability* (OPOR), in which users can task an external auditor to perform and verify POR with the cloud provider. We argue that the OPOR setting is subject to security risks that have not been covered by existing POR security models. To remedy that, we propose a formal framework and a security model for OPOR. We then propose an instantiation of OPOR which builds upon the provably-secure private POR scheme due to Shacham and Waters (Asiacrypt'08) and we show its security in our proposed security model. We implement a prototype based on our solution, and evaluate its performance in a realistic cloud setting. Our evaluation results show that our proposal minimizes user effort, incurs negligible overhead on the auditor (compared to the SW scheme), and considerably improves over existing publicly verifiable POR.

1. INTRODUCTION

Cloud services are increasingly gaining importance and applicability in numerous application domains, such as storage, computing services, collaboration platforms, etc. The success of the cloud model is driven by the tremendous economic benefit offered to companies, private individuals, and public organizations to deploy/provision cloud services in a cost effective manner.

The advent of cloud storage and computation services, however, introduces new threats to data security. As a matter of fact, customers of cloud services lose control over their data and how data is processed or stored. Indeed, this has been identified as the main obstacle which makes users reluctant when using cloud services [2,8]. The literature features a number of solutions that enable users to verify the integrity and availability of their outsourced data [14, 16, 24, 35]. Examples include Proofs of Retrievability (POR) [24, 35] which provide end-clients with the assurance that the data is still available and can be entirely downloaded if needed, and Proofs of Data Possession (PDP) [11] which enable a client to verify that its stored data has not undergone any modifications, among others. All existing solutions share a similar system and attacker model, comprising of the cloud user and a rational cloud provider. Here, the 'malicious' cloud aims at minimizing storage costs, e.g., by not deploying the appropriate security measures in their datacenters, or by intentionally modifying (e.g., deleting) user data.

Clearly, the guarantees provided by current solutions therefore largely depend on the users *themselves* who are required to regularly perform verifications (e.g., POR) in order to react as early as possible in case of data loss. Moreover, the verification of a POR requires the user to be equipped with devices that have network access, and that can tolerate the (non-negligible) computational overhead incurred by the verification process.

Therefore, customers either have to *(i)* accept this burden and regularly verify their outsourced data (e.g., by invoking POR with the cloud provider), or *(ii)* entrust the cloud providers to deploy the necessary security mechanisms that ensure data integrity in spite of server failures, exploits, etc. We point out that the integration of such security mechanisms in current clouds often incurs considerable costs on the cloud providers, which explains the reason why none of today's cloud storage services accept liability for data loss in their Service Level Agreements (SLAs) and only guarantee service availability [5,6] – in spite of the plethora of cloud security and dependability solutions that populate the literature [14, 24, 30, 35].

In this paper, we address this problem, and propose a novel solution—*outsourced proofs of retrievability* (OPOR)—which goes one step beyond existing POR and enables an external party, the *auditor*, to execute a POR protocol with the cloud provider on behalf of the data owner. OPOR protects against a malicious auditor, and malicious users/cloud providers (and against collusion among any combination of these parties); we contrast this to existing public (and delegable) POR [11, 32, 35, 35, 36], which allow an external party to *verify* a POR but do not provide any security guarantees when the user and/or external verifier are dishonest. OPOR provides users with the guarantee that their data is entirely stored in the cloud *without* having to verify their data themselves. Although auditors are made (legally) liable to monitor the availability of their files, users can verify the auditor's work at any point in time; we show that this verification can be much less frequent, and is con-

siderably more (computationally) efficient when compared to the verification of existing POR.

We argue that OPOR is technically and economically viable. Indeed, by providing the necessary security guarantees for the auditors, OPOR enables auditors to issue a security SLA for the cloud users attesting that they will correctly verify the availability of outsourced data, in exchange, e.g., of financial remuneration. While the main barriers of wide adoption of the cloud lie in the lack of customer trust and in the high costs of deploying security measures in cloud infrastructures, OPOR bridges these gaps and enables customers and external auditors to establish a financial contract by which customers can rest assured that the security of their files is constantly monitored.

In addition to introducing the notion of OPOR, we make the following additional contributions:

Formal Framework. We propose a formal framework and a security model for outsourced POR involving the cloud provider, the user, and the auditor. Our framework extends the POR model outlined in [35] and addresses security risks that have not been covered so far in existing models.

Concrete Instantiation. We describe a concrete OPOR scheme, dubbed Fortress, which builds upon the private SW POR scheme [35] and that is secure in our enhanced security model. Fortress inherits the security guarantees of this POR scheme but allows to shift most of the computations a user has to bear in a POR to the auditor. Fortress deploys a novel mechanism which enables the user and the auditor to commonly extract pseudo-random bits using a time-dependent source, and without any interaction; we show how this can be efficiently instantiated by leveraging functionality from Bitcoin.

Prototype Implementation. We implement and evaluate a prototype based on Fortress in a realistic cloud setting, and we show that our proposal minimizes user effort, incurs negligible overhead on the auditor when compared to the private-key POR of [35], and considerably improves the performance of existing public-key POR.

The remainder of this paper is organized as follows. In Section 2, we introduce a novel framework for outsourced proofs of retrievability, OPOR. In Section 3, we propose Fortress, an efficient instantiation of OPOR, and analyze its security. In Section 4, we describe a prototype implementation and evaluation of Fortress in realistic cloud settings and compare its performance to the POR schemes of [35]. In Section 5, we overview related work in the area and we conclude the paper in Section 6.

2. OPOR: OUTSOURCED PROOFS OF RETRIEVABILITY

In this section, we introduce a formal model for OPOR. Since OPOR extends POR, we first introduce POR, adapted from [35].

2.1 Proofs of Retrievability

Proofs of Retrievability (POR) are cryptographic proofs that *prove* the retrievability of outsourced data. More precisely, POR assume a model comprising of a user, and a service provider that stores a file pertaining to the user. POR consist basically of a challenge-response protocol in which the service provider proves to the user 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.

A POR scheme consists of four procedures [35], setup, store, verify, and prove:

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

store. This randomized algorithm takes as input the keys of the user and a file $M \in \{0,1\}^*$. The file gets processed and it outputs the produced M^* which will be stored on the server. The algorithm also generates a file tag τ which contains additional information (e.g., metadata, secret information) about M^*.

verify, prove. The randomized proving and verifying 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). While the verifier algorithm takes the secret keys as input, the prover algorithm takes as input the processed file M^* that is output by store. Both verify, prove algorithms also take as input the public key and the file tag τ from store during protocol execution. Algorithm verify 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.

2.2 OPOR Model

Similar to the traditional POR model, an OPOR consists of a user $\mathcal{U}$, the data owner, who plans to outsource his data M to a service provider $\mathcal{S}$. In addition, $\mathcal{U}$ is interested in acquiring regular proofs that his data is correctly stored and retrievable from $\mathcal{S}$. To this end, an OPOR comprises a new entity $\mathcal{A}$, called the auditor, who runs POR with $\mathcal{S}$ on behalf of $\mathcal{U}$. If these POR do not succeed, the auditor takes certain actions, e.g., inform the user immediately. Otherwise, the user is assured that the data are stored correctly.

More specifically, an OPOR scheme comprises five protocols Setup, Store, POR, CheckLog, and ProveLog. The first three protocols resemble the protocols that are represented in a POR scheme (see Section 2.1) but extend them. One major difference is that the POR protocol not only outputs a decision on whether the POR has been correct, but also a log file. The log files serve a twofold purpose. First, they allow the user to *check* (using the CheckLog procedure) if the auditor did his job during the runtime of the OPOR scheme. As the purpose of OPOR is to incur less burden on the user, the verification of the logs by the user should incur less resource consumption on the user when compared to the standard verification of POR directly with $\mathcal{S}$. Second, logs allow the auditor to *prove* (using the ProveLog procedure) that if some problems occur, e.g., the file is no longer stored by $\mathcal{S}$, the auditor must not be blamed. In what follows, we detail each protocol in OPOR.

The Setup *Protocol.*

This randomized protocol generates for each of the different parties a public-private key pair. If a party only deploys symmetric key schemes, the public key is simply set to $\perp$. For the sake of brevity, we implicitly assume for each of the subsequent protocols and procedures that an involved party always uses as inputs its own secret key and the public keys of the other parties.

The Store *Protocol.*

This randomized file-storing protocol takes the secret keys of the parties and a file M from the user to be stored. The output M^* for

the service provider marks the data that it should store. The user also needs a contract c specifying the policy for checks for the auditor. Observe that M^* may not be exactly equal to M, but it must be guaranteed that M can be recovered from M^*. Additionally, the output needs to contain information which *(i)* enables the execution of a POR protocol between $\mathcal{A}$ and $\mathcal{S}$ on the one hand and *(ii)* enables the validation of the log files created by $\mathcal{A}$ on the other hand. This information consists of two tokens represented by $\tau_{\mathcal{A}}$ and $\tau_{\mathcal{U}}$, respectively.

An important distinction from POR comes from the fact that when uploading a file M to $\mathcal{S}$ which should be monitored by $\mathcal{A}$, several agreements need to be established. We denote by $\mathsf{Agree}[\mathcal{P}_1, \mathcal{P}_2, [D]]$ a proof that both parties $\mathcal{P}_1$ and $\mathcal{P}_2$ agreed on a file D. Observe that this does not require that D is given in clear within the agreement. For example, an agreement could be the signed hash of D. Most important, user $\mathcal{U}$ and auditor $\mathcal{A}$ need to agree which file M^* will be monitored. In addition, user and auditor need to agree on the contract c that settles several parameters. For example it may set a maximum interval within which the auditor needs to notify the user in case M^* is (partially) lost and a maximum failure tolerance. Formally, it holds

$$\begin{aligned}\mathsf{Store}\colon\ & [\mathcal{U} : M, \mathsf{c};\ \mathcal{A} : \perp;\ \mathcal{S} : \perp] \\ \longrightarrow\ & [\mathcal{U} : \tau_{\mathcal{U}}, \mathsf{Agree}(\mathcal{U}, \mathcal{A}, [M^*, \tau_{\mathcal{U}}, \tau_{\mathcal{A}}, \mathsf{c}]), \mathsf{Agree}(\mathcal{U}, \mathcal{S}, [M^*]); \\ & \mathcal{A} : \tau_{\mathcal{A}}, \mathsf{Agree}(\mathcal{U}, \mathcal{A}, [M^*, \tau_{\mathcal{U}}, \tau_{\mathcal{A}}, \mathsf{c}]), \mathsf{Agree}(\mathcal{A}, \mathcal{S}, [M^*]); \\ & \mathcal{S} : M^*, \mathsf{Agree}(\mathcal{A}, \mathcal{S}, [M^*])].\end{aligned}$$

The protocol run is accepted by the parties, if the agreements succeed.

The POR *Protocol.*

In the OPOR model, the auditor $\mathcal{A}$ and the provider $\mathcal{S}$ run a POR protocol to convince the auditor that M^* is still retrievable from $\mathcal{S}$. The input of $\mathcal{A}$ is the tag $\tau_{\mathcal{A}}$ given by Store, and the input of the provider $\mathcal{S}$ is the stored copy of the file M^*. Similar to the traditional POR model, on the auditor's side (who plays the role of the verifier), the output contains one binary value $\mathrm{dec}_{\mathcal{A}}$ which expresses whether the auditor accepts the POR or not. In addition, the POR protocol will produce a log file Λ. It holds that:

$$\mathsf{POR}\colon\ [\mathcal{A} : \tau_{\mathcal{A}};\ \mathcal{S} : M^*] \longrightarrow [\mathcal{A} : \Lambda, \mathrm{dec}_{\mathcal{A}}]$$

The protocol run is accepted by the auditor if $\mathrm{dec}_{\mathcal{A}} = \mathrm{TRUE}$.

The CheckLog *Algorithm.*

In an OPOR, the POR protocol only convinces $\mathcal{A}$ that M^* is still retrievable. The CheckLog protocol enables $\mathcal{U}$ to audit the auditor. CheckLog is a deterministic algorithm which takes as input the verification key $\tau_{\mathcal{U}}$ and a log file Λ and outputs a binary variable dec_{Λ} which is either TRUE or FALSE, indicating whether the log file is correct. Formally:

$$\mathrm{dec}_{\Lambda} := \mathsf{CheckLog}(\tau_{\mathcal{U}}, \Lambda).$$

The ProveLog *Algorithm.*

ProveLog is a deterministic algorithm which complements the CheckLog procedure to ensure the correctness of the auditor in case of conflicts. In fact, if the CheckLog algorithm provides certainty about the correctness of the auditor, ProveLog is not necessary. Otherwise, ProveLog can without doubt prove or disprove the honesty of $\mathcal{A}$ as it has access to the secret information of $\mathcal{A}$. The algorithm ProveLog takes as input the tag $\tau_{\mathcal{A}}$ of the auditor and a log file Λ and outputs a binary variable $\mathrm{dec}_{\Lambda}^{\mathrm{corr}}$ which is either TRUE or FALSE, indicating whether the POR protocol run that produced the log file has been correctly executed by the auditor. Formally:

$$\mathrm{dec}_{\Lambda}^{\mathrm{corr}} := \mathsf{ProveLog}(\tau_{\mathcal{A}}, \Lambda).$$

Correctness.

The definition of *correctness* requires that if all parties are honest, then the auditor always, i.e., with probability 1, accepts at the end of each POR protocol run and likewise the user at the end of each CheckLog protocol run. This should hold for any choice of key pairs and for any file $M \in \{0,1\}^*$. Likewise, if the POR protocol has been executed correctly by the auditor based on $\tau_{\mathcal{A}}$ and yielded an output Λ, then the output of $\mathsf{ProveLog}(\tau_{\mathcal{A}}, \Lambda)$ should be TRUE with probability 1.

2.3 Security Model

In the following, we explain how security is defined within the OPOR model. We do not consider confidentiality of the file M, but assume that the user encrypts the file prior to the start the OPOR protocol. In OPOR, we extend the attacker model of traditional POR which only considers malicious service providers, and we assume that any subset of parties can be corrupted.

To define the soundness of an OPOR scheme, we adapt and extend the existing POR security models of [24, 35]. In [24, 35], security is formalized using the notion of an extractor algorithm, that is able to extract the file in interaction with the adversary. This proves the following statement: if the prover convinces the verifier with a sufficient level of probability then the file is actually stored. As already elaborated, the notion of extractability is not sufficient to capture security in OPOR schemes. In addition, an OPOR also should provide security to the auditor $\mathcal{A}$, who is taking over guarantees about the service quality of $\mathcal{S}$. In contrast to the POR security models, this is especially important in situations where something went wrong, e.g., the file has been lost. Consequently, we split the definition of soundness into two parts: soundness if no honest party aborts (we call this ε-extractability) and soundness for the case that one honest party aborted (what we refer to as (δ_1, δ_2)-liability).

Extractability. We start by describing the case where none of the honest parties aborts. More precisely, the honest parties interact with malicious parties. Recall that the ProveLog procedure is only invoked if one of the parties aborted and hence does not contribute to the notion of extractability. Consider the following experiment between an adversary who corrupted the parties specified in $C \subsetneq \{\mathcal{U}, \mathcal{A}, \mathcal{S}\}$ and an environment.

1. Initially, the environment runs the Setup protocol on behalf of all parties and generates the public and private keys. All public keys and the secret keys of the corrupted parties C are given to the adversary.

2. The environment plays the roles of the honest parties and the adversary simulates the roles of the corrupted parties. The adversary is allowed to request executions of Store for any file $M \in \{0,1\}^*$. Likewise, he can request the execution of POR or CheckLog for any stored file, that is for any file that has been input to a previous Store execution. In the protocols, the environment will play the role of honest parties and the adversary the role of the corrupted parties C. The adversary learns only the output provided to corrupted parties and whether the honest party accepted.

3. Finally, the adversary outputs a file M to challenge the environment with and the description of a machine implementing the role of the malicious parties for this file.

Observe that this game differs from the game described in traditional POR models in several aspects. First, it inherently incorporates the fact that several parties can be compromised and that the attacker can initiate also other protocols. Second, the output is not a description of an attacker for a file that has been stored already but for a file that should be stored. In other words, this game also covers the case that an attacker tries to cheat during the Store protocol already.

An attacker is ε-*admissible* if the probability that none of the honest parties aborts is at least ε. Here, the probability is over the coins of the honest and malicious parties. For the definition of security, we adopt the concept of *extractors*. An extractor algorithm Extr takes the values of the honest parties, e.g., their private keys and tokens, and the description of a machine implementing the role of the malicious parties in the OPOR system. The algorithm's output is a file $M \in \{0,1\}^*$. Note that Extr is given non-black-box access to the machine implementing the corrupted parties and can, in particular, rewind them.

DEFINITION 1. *We say that an OPOR scheme is* ε-extractable *with respect to a set of corrupted parties C if there exists an extraction algorithm such that, for any algorithm corrupting the parties in C and playing the aforementioned game, outputs an ε-admissible attacker $\mathcal{Z}$, the following holds: for any honest party that has an agreement for some file M^*, the extraction algorithm recovers M^*—except possibly with negligible probability.*

Naturally, if $\mathcal{S}$ is honest, Extr has the providers view as input and can trivially extract M^* already from its input. For the other two parties, the notion adapts the notion of extractability.

Liability. Next, we address the security definition for the case that one honest party aborts. Let us a call an auditor who generated all values occurring in the protocols (including the generation of $\tau_{\mathcal{A}}$ during Setup and the creation of the challenges during POR) according to the protocol specifications as *well-behaving*. If an auditor is not well-behaving, he is *misbehaving*. We want that any well-behaving auditor can prove that the log files are correct with a high probability while a misbehaving auditor should achieve this with a certain (preferably small) probability only. As this notion is motivated by potential legal issues between the user and the auditor, liability is only defined with respect to a set of corrupted parties that does not include the user *and* the auditor. We formalize this as follows:

DEFINITION 2. *We say that an OPOR scheme is* (δ_1, δ_2)-liable *with respect to a set of corrupted parties C with $\{\mathcal{U}, \mathcal{A}\} \not\subseteq C$ if the following holds. Let $\mathcal{Z}$ be an algorithm that corrupts the parties in C and plays the aforementioned game. Let M denote a file that has been the input to one* Setup *execution and let $\tau_{\mathcal{A}}$ denote the output for the auditor of this* Store *execution and $\mathcal{L}$ be the set of all log files which have been created afterwards based on the outputs of these specific* Store *execution. Then, it holds for any randomly selected log file $\Lambda \in \mathcal{L}$ that $\Pr[\mathsf{ProveLog}(\tau_{\mathcal{A}}, \Lambda)] = \mathrm{TRUE}] \geq \delta_1$ if the auditor has been well-behaving in the protocol executions associated to M and $\Pr[\mathsf{ProveLog}(\tau_{\mathcal{A}}, \Lambda)] = \mathrm{TRUE}] \leq \delta_2$ if the auditor has been misbehaving.*

Relation to the POR Model of Shacham and Waters. The proposed OPOR extends the POR model in [35]. In particular, any POR scheme Π with two parties, a verifier and a prover, can be expressed as an OPOR scheme Π' where the user and the auditor emulate the same party, namely the verifier, while the service provider plays the role of the prover. If Π is ε-sound within the POR model, then Π' provides ε-extractability within the OPOR model if only the service provider is corrupted. Note, however, that since Π does not specify a ProveLog procedure, liability is not automatically ensured in Π.

3. FORTRESS: AN EFFICIENT OPOR

In this section, we introduce and detail an efficient instantiation of OPOR. We analyze the security of our instantiation according to the model outlined in Section 2.

3.1 Overview

Fortress builds upon the private-key unbounded[1] POR scheme of [35] (that we shortly call PSW in the sequel) which minimizes bandwidth overhead, and maximizes performance/scalability. We chose to rely on SW-POR as a starting point for Fortress (instead of any other POR or PDP) based on two properties that are exhibited by the SW-POR and that facilitated the transformation into an OPOR: *(i)* since the setup phase uses only algebraic operations, the correctness of the setup parameters can be shown via a standard zero knowledge proof, and *(ii)* the produced proofs are also homomorphic, allowing the user to batch the verification of several proofs. We however expect that other POR/PDP schemes can be transformed into OPOR as well and leave it as an interesting question for future research. Note that a straightforward approach to achieve an OPOR would be to simply let the auditor instantiate all protocol parameters (e.g., secret keys, verification tags, etc.), and conduct the PSW regularly with the service provider. In the presence of a malicious service provider, this approach would inherit the same security guarantees as already proven for PSW. However, this straightforward solution does not protect against a malicious user, and/or a malicious auditor. In fact, to transform a secure POR into a secure OPOR, we argue that a number of challenges need to be addressed:

Malicious Auditor. Existing POR rely on the assumption that the verifier is honest. As such, these POR cannot be directly ported to an OPOR setting, where the auditor might deviate from the protocol, and/or collude with the service provider. For example, a malicious auditor may share the secret key with the service provider so that both can produce correct POR without having to store the file at all. In fact, the requirement of security against malicious auditors is often one of the main reasons why existing private-verifiable schemes cannot be outsourced by simply handing the secrets to the auditor (see Appendix B for some examples).

Auditing the Auditor. Although users should not be involved in verifying the retrievability of their files, OPOR should enable users to audit the auditors. Clearly, for an OPOR scheme to be effective, such an audit should be much less frequent and considerably more efficient than the act of verifying the files stored at the cloud. Note, however, that verifying the POR response typically requires the knowledge of the secret keys; if these keys are known to the user, a malicious user could reveal this key to a malicious service provider. Hence, one requires that the auditor can be audited to some extent—without revealing his secret keys.

Auditor Liability. Since the auditors want to minimize their liability, an OPOR scheme should *(i)* protect the auditors from

[1]For practical purposes, an OPOR scheme should support an unbounded number of queries by an external auditor – without the need for user interaction.

malicious users, and *(ii)* enable auditors to attest to any party that they did their work correctly, in case of dispute or litigation. This entire process should be efficient, and should scale with the number of the auditor customers.

Parameter Generation. A POR depends on several parameters. To achieve auditor liability, an auditor needs to be able to convincingly prove later on that these parameters have been constructed correctly—even if the file is no longer present.

Challenge Sampling. In the third phase of a POR, the verifier is typically required to construct a number of challenges (e.g., some randomly sampled field elements as in PSW). Clearly, these challenges cannot depend only on any of the involved three parties (the cloud, the user, and the auditor) since they might be malicious; note that interactive sampling among two parties would not solve this problem (since any two parties might collude), and would require user interaction. This problem is further exacerbated by the fact that the challenges cannot be pre-defined, e.g., by agreeing some seed of a pseudo-random bit generator since a malicious auditor might pre-share all the challenges to be queried with a malicious provider. Indeed, the latter can then compute the necessary replies to those challenges and delete the file, while answering correctly to all POR.

The core idea in Fortress (see Figure 1) to overcome the first two challenges is to require that the auditor conducts *two* POR in parallel with the service provider: one POR which can be verified by the auditor himself, and another one which can optionally be verified by the user (who has the right cryptographic keys). Upon the completion of each POR, the auditor logs the responses of the service provider, and the parameters used to conduct the two POR (e.g., block indices used at challenge). This second POR protects on the one hand against malicious auditor/service provider and allows for auditing the auditor. Here, Fortress enables the user to efficiently verify in a single batch a number of conducted POR to verify the work of the auditor. This minimizes communication overhead while achieving the same level of security and efficiency as in PSW.

However, an auditor could still cheat, e.g., by using wrong parameters or by biasing the challenge sampling process according to some strategy. To ensure correct parameter generation, Fortress relies on a sub-protocol which guarantees that the parameters computed by the auditor in the beginning have been correctly generated without revealing his secret parameters.

Moreover, to ensure a truly (pseudo-)random sampling of the challenges, Fortress exploits the fact that any randomized algorithm can be rewritten as a deterministic algorithm where the random bits are provided as additional input. The idea is to deprive the auditor from the right to sample these random bits and to extract them from an external source. To this end, Fortress leverages functionality from Bitcoin in order to provide a time-dependent source of pseudo-randomness to sample the parameters of the POR. An important property is that in case of potential conflicts, e.g., if the file gets lost, the auditor can provide an irrefutable cryptographic proof that he correctly followed the protocol. This can be achieved by opening the auditor's key for which a commitment has been signed in the beginning, i.e., during the Store protocol. Owing to the fact that any random bits extracted from Bitcoin can be uniquely reconstructed at any later point in time, the whole POR can be re-played to check if *(i)* the auditor did send the correct challenges and *(ii)* the response sent by the service provider has been correct. We show that Fortress incurs negligible overhead on the auditor, and scales well with the number of clients.

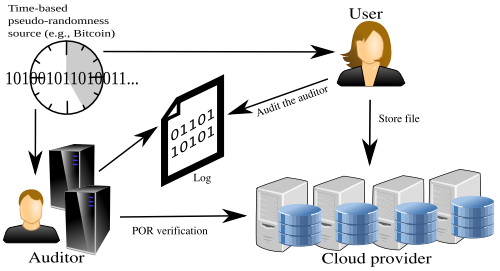


Figure 1: Sketch of Fortress. Fortress relies on a time-dependent source of pseudo-randomness (e.g., Bitcoin) to sample the parameters of the POR.

3.2 Protocol Specification

In this section, we describe, Fortress, a concrete instantiation of an OPOR. We start by outlining the main building blocks that are used in Fortress.

Building Blocks.

Unless otherwise specified, all operations in Fortress are performed in the finite field $\mathbb{F} = \mathbb{Z}_p$. Fortress makes use of a number of established cryptographic building blocks: a pseudo-random function $f : \{0,1\}^* \times \{0,1\}^{\ell_{\text{prf}}} \rightarrow \mathbb{F}$, a cryptographic hash function H, a signature scheme $(\text{KeyGen}, \text{Sign}, \text{Verify})$, and a pseudo-random bit generator:

$$g : \{0,1\}^{\ell_{\text{seed}}} \times \{0,1\}^{\ell_{\text{prbg}}} \rightarrow \{0,1\}^*.$$

Here, we assume that the output of the PRBG is long enough to extract the number of pseudo-random bits as required in the protocol. The bit length of p, ℓ_{prf}, ℓ_{seed}, ℓ_{prbg} are all chosen equal to the intended security level.

The GetRandomness *Procedure.*

In addition, Fortress leverages a time-dependent pseudo-randomness generator:

$$\mathsf{GetRandomness} : \Gamma \rightarrow \{0,1\}^{\ell_{\text{seed}}}.$$

GetRandomness has access to a secure time-dependent source. Let cur denote the current time. On input $t \in \Gamma$ with Γ being a time set, GetRandomness outputs a uniformly random string in $\{0,1\}^{\ell_{\text{seed}}}$ if $t \geq \text{cur}$, otherwise GetRandomness outputs $\perp$. GetRandomness is secure, if the output of $\mathsf{GetRandomness}(t)$ cannot be predicted with probability significantly better than $2^{-\ell_{\text{seed}}}$ as long as $t < \text{cur}$. In Fortress, we elect to instantiate GetRandomness by leveraging functionality from Bitcoin, since the latter offers a secure and convenient way (e.g., by means of API) to acquire time-dependent pseudo-randomness.

Bitcoin [9] relies on blocks, a hash-based Proof of Work (PoW) concept, to ensure the security of transactions. In particular, given the set of transactions that have been announced since the last block, and the hash of the last block, Bitcoin miners need to find a nonce, that would make the hash of the formed block smaller than a 256-bit number target:

$$\texttt{hash}\{\mathsf{BI} \,||\, \text{MR}(\text{TR}_1, \ldots, \text{TR}_n) \,||\, \text{nonce}\} \leq \text{target},$$

where BI denotes the hash of last generated block, $\text{MR}(\overline{x})$ denotes the root of the Merkle tree with elements $\overline{x}$, $\text{TR}_1 \,||\, \ldots \,||\, \text{TR}_n$ is a set of transactions that have been chosen by the miners to be included in the block.

The difficulty of block generation in Bitcoin is adjusted so that blocks are generated once every 10 minutes on average; it was

shown in [25] that the block generation in Bitcoin follows a shifted geometric distribution with parameter $p = 0.19$.

Given this, GetRandomness then unfolds as follows. On input time t, GetRandomness outputs the hash of the latest block that has appeared since time t in the Bitcoin block chain. Clearly, if t is in the future, then GetRandomness will output $\perp$, since the hash of a Bitcoin block that would appear in the future cannot be predicted. On the other hand, it is straightforward to compute GetRandomness(t), for a past time t, by fetching the hash of previous Bitcoin blocks.

We now detail the specifications for the four protocols Setup, Store, POR, CheckLog, and ProveLog in Fortress.

Specification of the Setup *Protocol.*

Each party $\mathcal{P} \in \{\mathcal{U}, \mathcal{A}, \mathcal{S}\}$ executes the key generation algorithm KeyGen of the digital signature scheme to receive a secret signing key $sk_{\mathcal{P}}$ and a public verification key $pk_{\mathcal{P}}$. The public keys are distributed amongst all parties.

Specification of the Store *Protocol.*

This Store protocol is initiated by the user $\mathcal{U}$, holding a file M. First, the user executes an information dispersal algorithm (i.e., erasure code) to disperse M into n blocks (for a given n, and a reconstruction threshold), each s sectors long: $\{m_{ij}\}_{1 \le i \le n, 1 \le j \le s}$. This will be the actual input to the interactive Store protocol. For communication links, we assume that they are authenticated, which can be realized by means of the TLS protocol as public/private key pairs are established.

User-controlled parameters: The user samples the values that are necessary for verifying a POR as mandated by PSW. More precisely, he samples a key for the PRF $k_{\text{prf}} \xleftarrow{\text{R}} \{0,1\}^{\ell_{\text{prf}}}$ and s elements of the finite field, i.e, $\alpha_1, \ldots, \alpha_s \xleftarrow{\text{R}} \mathbb{F}$. Finally, the user computes for each i, $1 \le i \le n$:

$$\sigma_i \leftarrow f_{k_{\text{prf}}}(i) + \sum_{j=1}^{s} \alpha_j m_{ij} \quad \in \mathbb{F}.$$

The user sets $\tau_{\mathcal{U}} := (k_{\text{prf}}, \alpha_1, \ldots, \alpha_s)$ and keeps it secret. We define the processed file $\widetilde{M} := (\{m_{ij}\}, \{\sigma_i\}_{1 \le i \le n})$. The file $\widetilde{M}$ is uploaded to the server $\mathcal{S}$.

Auditor-controlled parameters: The auditor $\mathcal{A}$ also samples secret values to verify a POR in PSW. That is, he samples a key for the PRF $k'_{\text{prf}} \xleftarrow{\text{R}} \{0,1\}^{\ell_{\text{prf}}}$ and s elements of the finite field, i.e, $\alpha'_1, \ldots, \alpha'_s \xleftarrow{\text{R}} \mathbb{F}$. Then, the file M' will be fetched[2] and parsed by the auditor from the service provider $\mathcal{S}$. Finally, the auditor computes for each i, $1 \le i \le n$:

$$\sigma'_i \leftarrow f_{k'_{\text{prf}}}(i) + \sum_{j=1}^{s} \alpha'_j m'_{ij} \quad \in \mathbb{F}.$$

The auditor uploads the values $\{\sigma'_i\}_{1 \le i \le n}$ and $\{\sigma_i\}_{1 \le i \le n}$ to the provider, and sends them also to the user together with a correctness proof (see below). The final file stored at $\mathcal{S}$ is M^*, composed of $\widetilde{M}$ and $\{\sigma'_i\}_{1 \le i \le n}$. The auditor sets $\tau_{\mathcal{A}} := (k'_{\text{prf}}, \alpha'_1, \ldots, \alpha'_s)$ and keeps it secret.

[2]In a practical instantiation, we assume that the auditor has read access rights over $\widetilde{M}$ which is stored at the cloud. If everyone follows the protocol and no errors occur, it holds $M' = \widetilde{M}$.

Proving correctness of σ'_i: The auditor needs now to convince the user that he correctly computed σ'_i. Therefore, user and auditor choose an RSA modulus N. The auditor should not know the factorisation, to ensure that he cannot compute the inverse modulo $\varphi(N)$. Similarly, the user must not be able to compute discrete logarithms in this group. We therefore elect that the user and auditor agree on an external mutually trusted number N, e.g., the value N of the root certificate of a certification authority. Then, both entities pick a generator $g < N$ in $\mathbb{Z}_N$, whose order is unknown (at least) to the auditor.

The auditor commits to the secret values α'_i as well as to the pseudo-random values used in computing σ'_i. In particular, $\mathcal{A}$ computes the following commitments:

$$g_j := g^{\alpha'_j} \bmod N, \qquad \text{for } 1 \le j \le s,$$
$$h_i := g^{f_{k'_{\text{prf}}}(i)} \bmod N, \qquad \text{for } 1 \le i \le n.$$

As the values σ'_i were computed in $\mathbb{F}$, i.e. mod p, the auditor computes for $i \in \{1, \ldots, n\}$ over the integers $\mathbb{Z}$

$$\sigma'^{\mathbb{Z}}_i = f_{k'_{\text{prf}}}(i) + \sum_{j=1}^{s} \alpha'_j m'_{ij} \quad \in \mathbb{Z}$$

and determines by means of integer division the values q_i with $\sigma'_i = \sigma'^{\mathbb{Z}}_i - q_i \cdot p$, where p is the prime used for the finite field $\mathbb{F} = \mathbb{Z}_p$. The auditor also computes commitments g^{q_i} and sends all commitments to the user $\mathcal{U}$.

Next, the user and the auditor execute a zero-knowledge-proof (ZKP) whose purpose is to show that the auditor indeed knows the discrete logarithms of the values g_i, h_j and g^{q_i}. For this purpose, Fortress leverages a non-interactive Schnorr ZKP protocol [34]. Here, to verify the knowledge, e.g., of α'_i, the auditor chooses a random value $r_i \in \mathbb{Z}$, computes $c_i = H(g^{r_i} \bmod N)$, and $d_i = r_i + c_i \cdot \alpha'_i$. The values $\{c_i, d_i\}$ are then sent to the user, who verifies $g^{d_i} = g^{r_i} \cdot (g_i)^{c_i} \mod N$. In our implementation, we sample r_i as a 240-bit random number and we use a 160-bit c_i.

$\mathcal{U}$ can now use all received commitments to check whether:

$$g^{\sigma'_i} \stackrel{?}{=} h_i \cdot \prod_{j=1}^{s} g_j^{m_{ij}} / \left(g^{q_i}\right)^p \text{ for } i \in \{1, \ldots, n\}.$$

If all verifications return TRUE, $\mathcal{U}$ then signs the commitments and sends his signature to $\mathcal{A}$.

Agreements: Besides the agreement on the values σ'_i, Fortress requires additional agreements between the user and the auditor, namely:

- All parties need to agree on the file that is stored. The provider will sign $H(M^*)$ once uploaded by the user and send the signature to the user to confirm reception of the file. The user forwards the receipt to the auditor, who will download the respective file and verify the H and the signature. Additionally, the auditor signs $H(M^*)$ and sends the signature to the user. The user verifies the signature and compares with $H(M^*)$. If any verification fails, user or auditor abort the protocol.

- User and auditor need to further agree on the conditions of their contract. We assume that the user and the auditor agree on the latest block Bl which has appeared in the Bitcoin block chain, and an interval d, which dictates the frequency at which the auditor performs the POR. User and auditor also agree on the sample sizes $\ell_{\mathcal{U}}$ and $\ell_{\mathcal{A}}$ to be checked in the POR. The user then requires the auditor to perform a POR

with the cloud provider whenever d new Bitcoin blocks appear in the Bitcoin chain[3]. This approximately corresponds to conducting a POR every $10d$ minutes starting from block Bl which marks the setup time. The auditor and user sign $H(\mathsf{Bl}, d, \ell_{\mathcal{U}}, \ell_{\mathcal{A}})$ and store it together with the signed file as confirmation of the contract.

Specification of the POR *Protocol.*

Our POR protocol corresponds to two parallel executions of PSW. Similar to the PSW, the auditor starts by generating two random POR challenges of size $\ell \in \{\ell_{\mathcal{A}}, \ell_{\mathcal{U}}\}$ for the two POR schemes established in Store. Here, note that $\ell_{\mathcal{U}} \ll \ell_{\mathcal{A}}$ since the user will batch-verify a number of log entries.

To generate a challenge of length ℓ, the verifier picks a random ℓ-element subset I of the set $\{1, \ldots, n\}$, and for each $i \in I$, a random element $\nu_i \xleftarrow{\mathrm{R}} \mathbb{F}$. The output of this algorithm, denoted by $\mathsf{Sample}(\ell)$, is the set $\{(i, \nu_i)\}_{i \in I}$ of size ℓ. Recall that any probabilistic algorithm can be considered as a deterministic algorithm if we specify the internal random coins θ as input, i.e., $\mathsf{Sample}(\theta, \ell)$. The core idea in our scheme is that the random coins θ are not sampled by the user and/or auditor, but are determined from the pseudo-random number generator g that is initialized with the seed obtained from $\mathsf{GetRandomness}(t)$ for the current time t.

The auditor $\mathcal{A}$ inputs $t \in \Gamma$ to GetRandomness in order to get a seed $y \in \{0,1\}^{\ell_{\mathrm{seed}}}$. Then, the PRBG is invoked on the seed y to get sufficient random bits $\theta = (\theta_{\mathcal{U}}, \theta_{\mathcal{A}})$ for use in the two algorithms $\mathsf{Sample}(\theta_{\mathcal{A}}, \ell_{\mathcal{A}})$ and $\mathsf{Sample}(\theta_{\mathcal{U}}, \ell_{\mathcal{U}})$ to obtain the challenge sets $Q_{\mathcal{A}}$ and $Q_{\mathcal{U}}$. This challenges are sent to the provider who has to respond with two POR responses: one based on the values σ_i that have been provided by the user and one using the auditor's σ'_i values. The provider now behaves exactly a in the SW scheme and computes the values $\mu_j, \mu'_j, \sigma, \sigma' \in \mathbb{F}$, for $1 \le j \le s$:

$$\mu_j \leftarrow \sum_{(i,\nu_i) \in Q_{\mathcal{U}}} \nu_i m_{ij}, \qquad \mu'_j \leftarrow \sum_{(i,\nu_i) \in Q_{\mathcal{A}}} \nu_i m_{ij},$$
$$\sigma \leftarrow \sum_{(i,\nu_i) \in Q_{\mathcal{U}}} \nu_i \sigma_i, \qquad \sigma' \leftarrow \sum_{(i,\nu_i) \in Q_{\mathcal{A}}} \nu_i \sigma'_i.$$

Finally, the service provider sends to the auditor the two responses $\rho := (\mu_1, \ldots, \mu_s, \sigma)$ and $\rho' := (\mu'_1, \ldots, \mu'_s, \sigma')$. Both responses ρ and ρ' are signed by $\mathcal{S}$ to offer non-repudiation, denoted by $\mathrm{Sig}_{\mathcal{S}}$. The auditor checks the signature of ρ and ρ' and verifies the latter POR response using $\tau_{\mathcal{A}}$ by checking the following equality:

$$\sigma' \stackrel{?}{=} \sum_{(i,\nu_i) \in Q_{\mathcal{A}}} \nu_i f_{k'_{\mathrm{prf}}}(i) + \sum_{j=1}^{s} \alpha'_j \mu'_j. \tag{1}$$

If this verification does not pass, the auditor informs the user according to the contract about problems with the storage of M^*. The other POR response ρ cannot be checked by $\mathcal{A}$ as the corresponding secret $\tau_{\mathcal{U}}$ is known to the user only.

The auditor finally creates the log entry comprising of the following information:

$$\Lambda := (\mathsf{Bl}_t, \rho, \mathrm{Sig}_{\mathcal{S}}(\rho), \rho', \mathrm{Sig}_{\mathcal{S}}(\rho')). \tag{2}$$

Specification of the CheckLog *Protocol.*

We first describe how a single entry in log file can be verified. First, the user checks the syntax and verifies the signature of $\mathcal{S}$ on the values ρ and ρ'. Then, the user determines $Q_{\mathcal{U}}$ as described in the POR protocol using $\mathsf{Sample}(\theta, \ell_{\mathcal{U}})$ with pseudo-random coins θ obtained with Bl_t. Afterwards, the correctness of ρ is checked, given $Q_{\mathcal{U}}$ and $\rho = (\mu_1, \ldots, \mu_s, \sigma)$ analogous to the verification of ρ' by the auditor in the POR protocol. Note that the user cannot verify ρ' without $\tau_{\mathcal{A}}$—this stronger verification of ρ' can only be performed in a "forensic" analysis with the protocol ProveLog.

As a minimal check, the user can check the last entry since this reflects the most recent state of retrievability for the file or a subset of entries. In Fortress the user has the possibility to check the POR accumulated over a batch of log entries. $\mathcal{U}$ selects a random subset B of indices of Bitcoin blocks and sends them to the auditor. $\mathcal{A}$ responds by accumulating the responses: $\rho^{(b)} = (\mu_1^{(b)}, \ldots, \mu_s^{(b)}, \sigma^{(b)})$ for $b \in B$ into one response

$$\left(\mu_1^{(B)} := \sum_{b \in B} \mu_1^{(b)},\ \ldots,\ \mu_s^{(B)} := \sum_{b \in B} \mu_s^{(b)},\ \sigma^{(B)} := \sum_{b \in B} \sigma^{(b)}\right).$$

The user reconstructs the challenges $Q_{\mathcal{U}}{}^{\mathsf{Bl}_b}$ for the selected entries from the Bitcoin block Bl_b with GetRandomness and checks:

$$\sigma^{(B)} \stackrel{?}{=} \left(\sum_{b \in B} \sum_{(i,\nu_i) \in Q_{\mathcal{U}}^{\mathsf{Bl}_b}} \nu_i f_{k_{\mathrm{prf}}}(i)\right) + \sum_{j=1}^{s} \alpha_j {\mu_j}^{(B)}.$$

If the user's check fails, the user will assume that either $\mathcal{A}$ or $\mathcal{S}$ is malicious and takes actions such as attempting to download the file or starting an analysis with ProveLog.

Specification of the ProveLog *Protocol.*

The ProveLog algorithm provides stronger means for analyzing the correct behavior of the auditor when compared to CheckLog. ProveLog requires that the auditor must reveal his secret token $\tau_{\mathcal{A}}$ and open the log Λ. In addition to the verifications in the CheckLog protocol, every server response ρ' to the auditor will be verified in ProveLog using $\tau_{\mathcal{A}}$. Additionally, the correctness of $\tau_{\mathcal{A}}$ will be verified, by recomputing commitments and verifying the user's signature generated in the Store protocol during the verification of the auditor's σ_i values. If all verifications pass, the auditor can prove that it has executed all protocols correctly.

3.3 Security Analysis

In the following, we show that Fortress is secure according to Definitions 1 and 2 (cf. Section 2.2) with respect to any constellation of corrupted parties.

ε-extractability: We start by discussing the ε-extractability property of Fortress (Definition 1). Observe that if a scheme is secure with respect to a set of corrupted parties C, it automatically is secure with respect to any subset $C' \subsetneq C$. Hence, to show our claim it suffices to consider the three cases where exactly one party is honest. First, we discuss the scenario where the user (and only the user) is honest. We want to show that if ρ produced by $(\mathcal{A}, \mathcal{S})$ as output of CheckLog is accepted with some probability $\ge \varepsilon$ by the user, then the file can be extracted by a means of an extraction algorithm. Recall that the log files actually contain the responses of the service provider $\mathcal{S}$ for a POR protocol that has been executed by the auditor on behalf of the user but without knowing $\tau_{\mathcal{U}}$. Thus, as long as the auditor followed the POR protocol, security is directly inherited from PSW.

More precisely, one can show similar to [35] that if correct responses to the challenges can be produced, then an extractor can

[3] In case of block forks [9], the auditor can make use of the hashes of one of the block forks that appear at the same height in the block chain.

be described that can extract the file. As neither the auditor nor the service provider know $\tau_{\mathcal{U}}$, this still holds even if both collude. However, we note an important difference: while in [35] the challenge $(I, \{(i, \nu_i)\}_{i \in I})$ is randomly sampled, challenges in Fortress are generated pseudo-randomly using the pseudo-random bit generator g with a seed extracted with GetRandomness. However, we argue that this does not bear any practical implication. To see why, let Chl denote the set of all possible challenges that can be produced using g. Observe that the seed to the g are coming from GetRandomness which derives the output from uniquely determined Bitcoin blocks. This has the consequence that each seed is possible and none of the seeds can be predicted or the adversary would gain the power to create Bitcoins at will. Hence, if an attacker wants to exploit the fact that the challenges are created using g, the attacker could be easily transformed into an efficient distinguisher for the g outputs which would contradict the security of g. Observe that the user checks within the CheckLog procedure that the correct challenges have been used. Hence, any deviation from this process, i.e., if other challenges are used, would be detected by the user who would abort. This concludes the first case. The same argumentation can be applied to the case that the auditor is honest.

It remains to address the final case, where the service provider is honest while the other parties are malicious. We have to show that from the fact that the service provider does not abort, it follows that the file can be reconstructed. First of all, the service provider does not check any responses. Hence, the only situation where he may abort is during the initialization phase of Store, e.g., if the wrong file has been signed, etc. If this has not happened, then $\mathcal{S}$ will follow honestly the protocol and in particular store the whole file M^*. This means of course that the honest party, i.e., the service provider, is able to recover the file.

Summing up, we have argued that for any constellation of malicious parties, it holds that if the probability that the honest user aborts is below a certain threshold, the file can be reconstructed.

Liability: With respect to liability (Definition 2), we have to show that an honest auditor can prove the correctness of the log files with high probability while a misbehaving auditor will fail. Recall that the notion of liability is only defined for scenarios where either the user or the auditor are malicious (but not both). First, we argue that the values σ'_i have been computed correctly. If the auditor is honest, it follows the protocol and computes the values correctly. Assume now that the auditor is malicious. Observe that any choice for $\alpha'_1, \ldots, \alpha'_s$ is valid. Likewise, the values $f_{k'_{\text{prf}}}(i)$ can safely be replaced by some random values $r_i \in \mathbb{F}$. Thus, any choice of these is correct. Moreover, due to the ZKPs it is ensured that the auditor knows these values. The only possibility for an auditor to misbehave is hence to output instead of a correct tuple (σ'_i, q'_i) a different tuple $(\tilde{\sigma}, \tilde{q}) \neq (\sigma'_i, q'_i)$. Here, output means that the first entry is given out in clear while the other is indirectly given by a commitment together with an appropriate ZKP. Clearly, this implies that the auditor knows the tuple $(\tilde{\sigma}, \tilde{q})$. Moreover, as the auditor also knows the values α'_j and r_i, he also knows the correct tuple (σ'_i, q'_i). Summing up, the auditors knows *two different* tuples such that:

$$\sigma'_i + q'_i \cdot p \equiv \tilde{\sigma} + \tilde{q} \cdot p \mod \varphi(N).$$

This is equivalent to:

$$\sigma'_i - \tilde{\sigma} \equiv (\tilde{q} - q_i) \cdot p \mod \varphi(N). \quad (3)$$

We show now that this information allows the auditor to factorize N. Assume first that $\sigma'_i - \tilde{\sigma} \equiv 0$ modulo $\varphi(N)$. As both values are less than $p < \varphi(N)$, it follows that $\sigma'_i = \tilde{\sigma}$ as integers and hence $(\tilde{q} - q_i) \cdot p \equiv 0 \mod \varphi(N)$. If p is not a divisor of $\varphi(N)$ (what we can expect in most cases), it follows that $\tilde{q} \equiv q_i \mod \varphi(N)$. If $\tilde{q} = q_i$ as integers, then it would hold $(\tilde{\sigma}, \tilde{q}) = (\sigma'_i, q'_i)$, contradicting our initial assumption. Hence, it must hold that $\tilde{q} \neq q_i$ as integers, and $\tilde{q} - q_i$ is a multiple of $\varphi(N)$. Using the common arguments, this can be used to deduce $\varphi(N)$ which in turn allows to factorize N. Next, assume that $\sigma'_i \neq \tilde{\sigma}$ as integers and without loss of generality that $\sigma'_i > \tilde{\sigma}$. It holds that $0 < \sigma'_i - \tilde{\sigma} < p$ and hence $(\tilde{q} - q_i) \cdot p - (\sigma'_i - \tilde{\sigma})$ is a multiple of $\varphi(N)$. The same arguments then can be applied as above. In conclusion, in both cases the auditor is either generating the parameters correctly or the user would abort the protocol.

Observe that from now on, all POR executions conducted by the auditor are deterministic in the sense that the auditor has no influence on the involved parameters. The points in time when the protocol execution should take place are dictated by the contract. The random bits used at these points in time are coming from GetRandomness and can be reconstructed later on. The verification of the responses are based on the secrets of the auditor only to which commitments have been done in the beginning. Hence, any party who knows $\tau_{\mathcal{A}}$ can reconstruct what should have happened in the respective POR executions and can compare these with the log files. Hence, as long as none of the underlying cryptographic building blocks are broken, this yields an objective proof for a well-behaving auditor that everything has been conducted correctly while ensuring that any misbehaviour would be detected.

Parameter	Default Value
Erasure-coding rate	(9,12)
File size	64 MB
|p|	80 bits
RSA modulus size	1024 bit
ℓ_A/n	10%
ℓ_U/n	1%

Table 1: Default parameters used in evaluation.

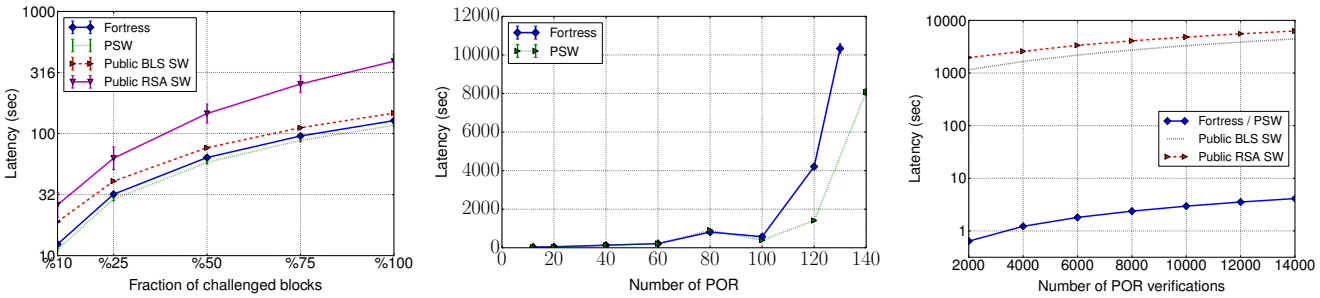
4. IMPLEMENTATION & EVALUATION

In this section, we evaluate an implementation of Fortress within a realistic cloud setting and we compare the performance of Fortress to the private- and public-POR schemes due to [35] and to the PDP scheme of [11].

4.1 Implementation Setup

We implemented a prototype of Fortress in Java. In our implementation, we relied on the Jerasure library [4] for constructing the dispersal codes (instantiated using Reed-Solomon coding). For a baseline comparison, we also implemented PSW and the public POR (with its two BLS and RSA variants) schemes (see Appendix A for a description of the SW POR schemes). Here, we relied on 160-bit SHA1, the Java built-in random number generator, HMAC based on SHA1, and the JPBC library [7] (based on the PBC cryptographic library [3]) to implement 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 and auditors were co-located on the second 24-core Xeon E5-2640 machine; this ensures a fair comparison between the overhead incurred on the user in the SW POR schemes and on the auditor in



(a) Latency incurred in POR w.r.t. the fraction of challenged blocks.

(b) Latency incurred in POR w.r.t. number of concurrent operations.

(c) Latency incurred in POR verifications w.r.t. number of concurrent operations.

Figure 2: Performance of the POR phase in Fortress in comparison to the private and public POR of [35]. Each data point in our plots is averaged over 10 independent runs; where appropriate, we also show the corresponding 95% confidence intervals.

Fortress.

To emulate a realistic Wide Area Network (WAN), we relied on NetEm [27]. For that purpose, we add a Pareto distribution with a mean of 20 ms and a variance of 4 ms, to shape all traffic exchanged on the networking interfaces to emulate the packet delay variance specific to WANs [21].

In our setup, each client invokes an operation in a closed loop, i.e., a client may have at most one pending operation. Prior to the setup phase, each client disperses his files with a (9,12) code. In our evaluation, we abstract away the time to encrypt the file (prior to erasure-coding), and we do not measure the upload/download times of the file to/from the cloud, since this overhead is common to all investigated schemes.

In our implementations, each client queries for the availability of an individual file stored on a local disk in the server. This prevents the need for different clients to synchronize on a common file descriptor when querying pieces from the storage servers. To acquire Bitcoin block hashes, our clients invoke an HTTP request to a getblockhash tool offered by the Bitcoin block explorer[4] [1].

We evaluate the latency incurred in Fortress and in the private and public POR schemes of [35], with respect to the number of challenges, the block size, and the number of sectors. When implementing Fortress, we spawned multiple threads on the auditor 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 Fortress, we start by analyzing the impact of the block size on the latencies incurred in the verification of POR in the private and public SW schemes [35]. Here, we abstract away the network delays and focus on measuring the time required by the clients/auditor to verify the response of the cloud. Note that this verification in Fortress corresponds to that of the private SW scheme. Our results (Figure 6 in Appendix C) show that modest block sizes of 16 KB yield the most balanced performance, on average, across all investigated schemes. Throughout the rest of our evaluation, we therefore set the block size to 16 KB.

POR protocol performance: In Figure 2(a), we evaluate the time required by the auditor to perform a single POR in Fortress, when compared to the SW schemes. For this purpose, we vary in the x-axis the fraction of challenged blocks of the total number of blocks of the file. In Fortress, the number of user-challenges are set by default to 10% of the auditor-challenges. Our results show that Fortress incurs a small additional overhead in time (~10%) in the POR when compared to PSW, due to the need to sample the challenges based on the latest Bitcoin block, and owing to the additional overhead required to create and verify the user-challenges. Recall that these operations enable Fortress to achieves a stronger security level than that of PSW. Moreover, the time to perform POR in Fortress increases linearly with the fraction of challenged blocks; all investigated schemes also exhibited a similar performance. Fortress however improves the performance of the BLS-based SW POR by almost 17%, and the RSA-based SW POR by 50%. Figure 2(b) depicts the latency incurred in Fortress and in PSW w.r.t. to the number of concurrent POR operations in the nominal case where 10% of the stored blocks are challenged in each POR. Our findings show that Fortress results in a tolerable performance degradation when compared to PSW under heavy load, in spite of the additional procedures (e.g., GetRandomness, user-challenges) employed in Fortress.

In a separate experiment that we conducted, we evaluated the peak throughout exhibited by Fortress when verifying POR (i.e., when verifying Equation 1). We measure peak 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 operations that can be performed with the storage servers per second. Our results show that the peak throughput of Fortress is approximately 5200 POR verification operations per second.

In Figure 2(c), we measure the latency incurred in Fortress when compared to the SW POR schemes, w.r.t. the number of concurrent POR verifications that can be performed by the auditor per second. Recall that this verification in Fortress is specular to that of PSW (i.e., verification of Equation 1). Our results show that POR verifications in Fortress is almost 2000 times faster than BLS SW POR, and 3000 times faster than the RSA counterpart. This clearly shows that Fortress scales well with the number of clients when compared to existing unbounded public POR schemes.

Store protocol performance: In Figure 3, we measure the time required in the store procedure of the investigated POR schemes. As expected, the store time increases almost linearly with the file

[4]For example, the hash of Bitcoin block 'X' can be acquired by invoking `https://blockexplorer.com/q/getblockhash/X`.

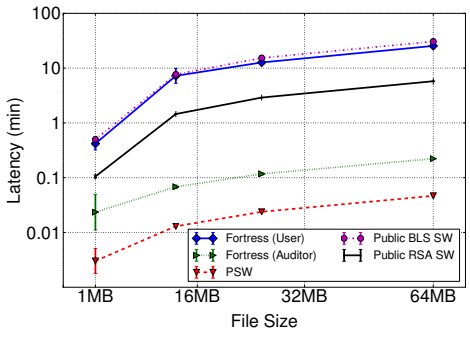


Figure 3: Latency in store w.r.t. to the file size.

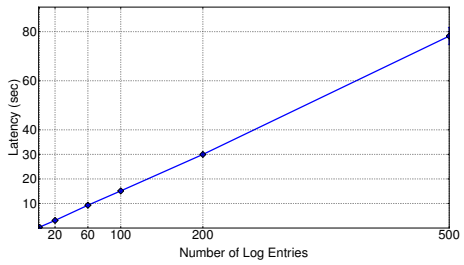


Figure 4: User verification latency in the CheckLog procedure of Fortress w.r.t. the number of verified log entries.

size in all POR schemes (note the logarithmic y-axis scale). Our results show that the store latency incurred on the auditor in Fortress is almost 5 times slower than of PSW; this overhead is mainly due to the generation of the various commitments by the auditor. We point out that this process is 100 times faster than that the store in public BLS SW, and almost 25 times faster than the public RSA scheme. On the other hand, the time required by the user to complete the store procedure is considerably larger, since the user needs to verify the correctness of the ZKP of all the auditor POR parameters. Indeed, this process is almost 100 times slower compared to the store performance of the auditor; the store time incurred on the user in Fortress is however 20% faster than that required by the public BLS SW scheme. Although the store procedure is expensive on the user, this process is only performed once—after which the user does not need to perform any operation.

User verification (CheckLog): In Figure 4, we evaluate the time incurred on the user when verifying the auditor logs with respect to the number of verified log entries. Here, the user requests a number of log entries for verification; these entries are accumulated into one response and sent by the auditor to the user who can batch-verify them in a single computation. In our implementation of Fortress, the size of each log entry stored by the auditor is approximately 32 KB (8 bytes for the Bitcoin block, 20 bytes for the Bitcoin hash, 32800 bytes for the cloud response, and 256 bytes for the signatures). This implies almost 100% storage blowup when compared to the standalone PSW scheme (i.e., the size of cloud response in PSW is around 16400 bytes); this is due to the fact that in Fortress, the auditor needs to store both the response of the cloud to his challenges, and to the user-based challenges. Our results show that the latency incurred on the user increases linearly with the number of verified log entries. Note that this effort is only a small fraction of the burden incurred on the auditor. Indeed, this overhead can be, to a large extent, tolerated in practical settings; recall that this is an optional verification, and is only seldomly performed by the user.

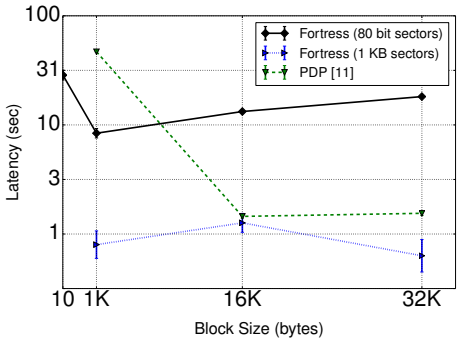


Figure 5: Performance of the POR phase in Fortress in comparison to the PDP of [11] with respect to the block size. We rely on a modulus size of 1024 bits when implementing the PDP.

Comparison with the PDP of [11]: In an additional experiment that we conducted, we measure the performance of Fortress when compared to publicly verifiable PDP scheme of [11]. Note that the latter PDP scheme can be seen as a variant of the public RSA SW scheme in which each block comprises of a single sector. Recall that the PDP of [11] is not secure in the OPOR model (see Appendix B). Our results (cf. Figure 5) show that the latency of performing a POR in Fortress—featuring 80 bits sector size—is considerably faster than that of [11] when the block size is modest (e.g., $\leq$ 8000 bits). As the block size increases, the PDP of [11] results in smaller latencies; we believe that this is due to the large number of I/O operations required by Fortress to fetch the numerous sectors from each challenged block. Indeed, as the number of sectors per block decreases in Fortress (e.g., when the sector size increases to 1 KB), our results show that Fortress considerably improves the performance over the PDP of [11].

5. RELATED WORK

Juels and Kaliski [24] present a POR scheme, which relies on sentinels that are 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 POR in [24], which tolerates a Byzantine adversarial model. Shacham and Waters [35] propose private-key-based and public-key-based POR schemes which utilize homomorphic authenticators to yield compact proofs. Dodis *et al.* [22] generalize the schemes of [24, 35] and introduce the notion of POR codes, which combines concepts in POR and hardness amplification.

In [11], Ateniese *et al.* propose 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. This proposal was later extended in [12] to address dynamic writes/updates from the clients. Erway *et al.* [23] present a *dynamic* PDP which leverages on authenticated dictionaries based on rank information. Similarly, Cash *et al.* [17] propose a dynamic POR scheme which relies on oblivious RAM protocols. In [37], 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 [32, 36], the authors describe schemes that enable the user to dele-

gate the verification of POR and to prevent their further re-delegation. Curtmola *et al.* propose in [19] a multiple-replica PDP, which enables a user to verify that a file is replicated at least across t replicas by the cloud. HAIL [14] enables a set of servers to prove to a client that a stored file is intact and retrievable against a mobile adversary, i.e., one that may progressively corrupt the full set of storage servers. In [16], Bowers *et al.* propose a scheme that enables a user to verify if his data is stored (redundantly) at multiple servers by measuring the time taken for a server to respond to a read request for a set of data blocks.

Proofs of location (PoL) [28, 38] aim at proving the geographic position of data, e.g., if it is stored on servers within a certain country. In [38], Watson *et al.* provide a formal definition for PoL schemes by combining the use of geolocation techniques together with the SW POR schemes. *Proofs of ownership* schemes [29] also share similarities with POR. A proof of ownership scheme aims to provide assurance that a client indeed possesses a given file, which is e.g., already stored at the cloud.

In [30], Popa *et al.* present a scheme which allows to prove the occurrence of violations in the cloud against data integrity, write-serializability and freshness to a third party.

First introduced in 2008, Bitcoin has considerably attracted the attention of the research community [10, 18, 20, 25, 26, 31, 33]. As far as we are aware, this is the first contribution which proposes the reliance on Bitcoin PoW as a secure time-dependent pseudo-randomness source.

6. CONCLUSION

In this paper, we introduced the notion of outsourced proofs of retrievability (OPOR), an extension of the traditional POR concept, and proposed an efficient instantiation of OPOR, dubbed Fortress. We implemented a prototype based on Fortress, and evaluated its performance in a realistic cloud setting. Our results show that our proposal incurs minimal overhead on the user and scales well with the number of users.

We argue that Fortress motivates a novel business model in which customers and external auditors establish a contract by which customers can rest assured about the security of their files. By doing so, Fortress increases the users' trust in the cloud, while incurring minimal user interaction. We therefore argue that our work lays basic foundations for realizing secure external auditing of cloud services; we believe that such auditor-based schemes will provide a stepping stone for establishing a cyber-insurance market for cloud services.

In terms of future work, we plan to explore efficient mechanisms to optimize the store procedure in Fortress, to investigate a generic transformation to turn any POR into an OPOR, and to design an OPOR scheme that supports dynamic updates of the stored file.

7. ACKNOWLEDGEMENTS

The authors would like to thank the anonymous reviewers for their valuable feedback and comments. This work was partly supported by the EU FP7 SECCRIT project, funded by the European Commission under grant agreement no. 312758.

8. REFERENCES

[1] Bitcoin real-time stats and tools. http://blockexplorer.com/q.

[2] Cloud Computing: Cloud Security Concerns. http://technet.microsoft.com/en-us/magazine/hh536219.aspx.

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

[4] Jerasure. https://github.com/tsuraan/Jerasure, 2008.

[5] Amazon S3 Service Level Agreement, 2009. http://aws.amazon.com/s3-sla/.

[6] Micorosfot Corporation. Windows Azure Pricing and Service Agreement, 2009.

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

[8] Protect data stored and shared in public cloud storage. http://i.dell.com/sites/doccontent/shared-content/data-sheets/en/Documents/Dell_Data_Protection_Cloud_Edition_Data_Sheet.pdf, 2013.

[9] SATOSHI NAKAMOTO. Bitcoin: A Peer-to-Peer Electronic Cash System.

[10] ANDROULAKI, E., KARAME, G., AND CAPKUN, S. Evaluating user privacy in bitcoin. http://eprint.iacr.org/2012/596.pdf.

[11] ATENIESE, G., BURNS, R. C., CURTMOLA, R., HERRING, J., KISSNER, L., PETERSON, Z. N. J., AND SONG, D. X. Provable data possession at untrusted stores. In *ACM Conference on Computer and Communications Security* (2007), pp. 598–609.

[12] ATENIESE, G., PIETRO, R. D., MANCINI, L. V., AND TSUDIK, G. Scalable and efficient provable data possession. *IACR Cryptology ePrint Archive 2008* (2008), 114.

[13] BONEH, D., LYNN, B., AND SHACHAM, H. Short signatures from the weil pairing. *J. Cryptology 17*, 4 (2004), 297–319.

[14] BOWERS, K. D., JUELS, A., AND OPREA, A. HAIL: a high-availability and integrity layer for cloud storage. In *ACM Conference on Computer and Communications Security* (2009), pp. 187–198.

[15] BOWERS, K. D., JUELS, A., AND OPREA, A. Proofs of retrievability: theory and implementation. In *CCSW* (2009), pp. 43–54.

[16] BOWERS, K. D., VAN DIJK, M., JUELS, A., OPREA, A., AND RIVEST, R. L. How to tell if your cloud files are vulnerable to drive crashes. In *ACM Conference on Computer and Communications Security* (2011), pp. 501–514.

[17] CASH, D., KÜPÇÜ, A., AND WICHS, D. Dynamic Proofs of Retrievability via Oblivious RAM. In *EUROCRYPT* (2013), pp. 279–295.

[18] CLARK, J., AND ESSEX, A. (Short Paper) CommitCoin: Carbon Dating Commitments with Bitcoin. In *Proceedings of Financial Cryptography and Data Security* (2012).

[19] CURTMOLA, R., KHAN, O., BURNS, R. C., AND ATENIESE, G. MR-PDP: Multiple-Replica Provable Data Possession. In *ICDCS* (2008), pp. 411–420.

[20] DECKER, C., AND WATTENHOFER, R. Information Propagation in the Bitcoin Network. In *13-th IEEE International Conference on Peer-to-Peer Computing* (2013).

[21] DOBRE, D., KARAME, G., LI, W., MAJUNTKE, M., SURI, N., AND VUKOLIĆ, M. Powerstore: Proofs of writing for efficient and robust storage. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security* (New York, NY, USA, 2013), CCS '13, ACM,

pp. 285–298.

[22] DODIS, Y., VADHAN, S. P., AND WICHS, D. Proofs of Retrievability via Hardness Amplification. In *TCC* (2009), pp. 109–127.

[23] ERWAY, C. C., KÜPÇÜ, A., PAPAMANTHOU, C., AND TAMASSIA, R. Dynamic provable data possession. In *ACM Conference on Computer and Communications Security* (2009), pp. 213–222.

[24] JUELS, A., AND JR., B. S. K. PORs: Proofs Of Retrievability for Large Files. In *ACM Conference on Computer and Communications Security* (2007), pp. 584–597.

[25] KARAME, G. O., ANDROULAKI, E., AND CAPKUN, S. Double-spending fast payments in bitcoin. In *Proceedings of the 2012 ACM conference on Computer and communications security* (New York, NY, USA, 2012), CCS '12, ACM, pp. 906–917.

[26] MEIKLEJOHN, S., POMAROLE, M., JORDAN, G., LEVCHENKO, K., MCCOY, D., VOELKER, G. M., AND SAVAGE, S. A fistful of bitcoins: Characterizing payments among men with no names. In *Proceedings of the 2013 Conference on Internet Measurement Conference* (New York, NY, USA, 2013), IMC '13, ACM, pp. 127–140.

[27] NETEM. NetEm, the Linux Foundation. Website, 2009. Available online at `http://www.linuxfoundation.org/collaborate/workgroups/networking/netem`.

[28] PETERSON, Z. N. J., GONDREE, M., AND BEVERLY, R. A position paper on data sovereignty: The importance of geolocating data in the cloud. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Cloud Computing* (Berkeley, CA, USA, 2011), HotCloud'11, USENIX Association, pp. 9–9.

[29] PIETRO, R. D., AND SORNIOTTI, A. Boosting efficiency and security in proof of ownership for deduplication. In *ASIACCS* (2012), H. Y. Youm and Y. Won, Eds., ACM, pp. 81–82.

[30] POPA, R. A., LORCH, J. R., MOLNAR, D., WANG, H. J., AND ZHUANG, L. Enabling Security in Cloud Storage SLAs with CloudProof. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2011), USENIXATC'11, USENIX Association, pp. 31–31.

[31] REID, F., AND HARRIGAN, M. An Analysis of Anonymity in the Bitcoin System. *CoRR* (2011).

[32] REN, Y., XU, J., WANG, J., AND KIM, J.-U. Designated-verifier provable data possession in public cloud storage. *International Journal of Security and Its Applications 7*, 6 (2013), 11–20.

[33] RON, D., AND SHAMIR, A. Quantitative analysis of the full bitcoin transaction graph. `http://eprint.iacr.org/2012/584.pdf`.

[34] SCHNORR, C.-P. Efficient identification and signatures for smart cards (abstract). In *EUROCRYPT* (1989), J.-J. Quisquater and J. Vandewalle, Eds., vol. 434 of *Lecture Notes in Computer Science*, Springer, pp. 688–689.

[35] SHACHAM, H., AND WATERS, B. Compact Proofs of Retrievability. In *ASIACRYPT* (2008), pp. 90–107.

[36] SHEN, S.-T., AND TZENG, W.-G. Delegable provable data possession for remote data in the clouds. In *ICICS* (2011), S. Qing, W. Susilo, G. Wang, and D. Liu, Eds., vol. 7043 of *Lecture Notes in Computer Science*, Springer, pp. 93–111.

[37] SHI, E., STEFANOV, E., AND PAPAMANTHOU, C. Practical dynamic proofs of retrievability. In *ACM Conference on Computer and Communications Security* (2013), A.-R. Sadeghi, V. D. Gligor, and M. Yung, Eds., ACM, pp. 325–336.

[38] WATSON, G. J., SAFAVI-NAINI, R., ALIMOMENI, M., LOCASTO, M. E., AND NARAYAN, S. Lost: location based storage. In *CCSW* (2012), T. Yu, S. Capkun, and S. Kamara, Eds., ACM, pp. 59–70.

APPENDIX

A. POR SCHEMES OF SHACHAM AND WATERS

The literature features a number of POR proposals; we refer the readers to Section 5 for a comprehensive overview of existing proposals. To the best of our knowledge, Juels and Kaliski (JK) [24] propose the first formal security definition of POR; their definition relies on the notion of an extractor algorithm, which interacts with the server provider, and must be able to extract the file with overwhelming probability if the provider passes the POR verification. Shacham and Waters (SW) [35] later built on this model, and proposed POR constructs based on private and public-key cryptography which rely on homomorphic authenticators to reduce the communication costs drastically by aggregating the responses of the service provider.

PSW: The private POR scheme of SW (PSW) leverages pseudo-random functions (PRFs). Here, the user first erasure encodes a file M and then splits it into n blocks $m_1, \ldots, m_n \in \mathbb{Z}_p$, where p is a large prime. The user then chooses a random $\alpha \in_{\mathrm{R}} \mathbb{Z}_p$ and a PRF key k for function f. The user then authenticates each block as follows:

$$\sigma_i = f_k(i) + \alpha m_i \in \mathbb{Z}_p.$$

The blocks $\{m_i\}$ and their authenticators $\{\sigma_i\}$ are all stored at the service provider in M^*.

At the POR verification stage, the verifier (here, the user) chooses a random challenge set $I \subset_{\mathrm{R}} [1, n]$, $|I| = \ell$, and ℓ random coefficients $\nu_i \in_{\mathrm{R}} \mathbb{Z}_p$. The challenge query then is the set $Q := \{(i, \nu_i)\}_{i \in I}$ which is sent to the prover (here, service provider). The prover computes the response (σ, μ) as follows and sends it back to the verifier:

$$\sigma \leftarrow \sum_{(i,\nu_i) \in Q} \nu_i \sigma_i, \quad \mu \leftarrow \sum_{(i,\nu_i) \in Q} \nu_i m_i.$$

Finally, the verifier checks the correctness of the response:

$$\sigma \stackrel{?}{=} \alpha\mu + \sum_{(i,\nu_i) \in Q} \nu_i f_k(i).$$

Public SW POR Scheme: PSW only enables the user, who possesses the secrets α and k, to verify a POR. To enable any entity which does not necessarily possess secrets to verify a POR, SW 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 $\sigma_i = (H(i)u^{m_i})^x$. For verification,

the challenge query Q is generated similarly to PSW and sent to the prover who computes:

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

These values are sent to the verifier who checks that:

$$e(\sigma, g) \stackrel{?}{=} e\left(\prod_{(i,\nu_i)\in Q} H(i)^{\nu_i} u^{\mu}, v\right).$$

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:

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

B. ON THE VULNERABILITY OF EXISTING SCHEMES AGAINST MALICIOUS AUDITORS

As pointed out in Section 3.1, existing private-verifiable POR and PDP schemes cannot be simply outsourced since they cannot deter against malicious auditors. To illustrate this, we briefly show in what follows possible attacks by a malicious auditor in three POR/PDP schemes. In the sequel, we adopt the notation from the original works and refer to these for further details.

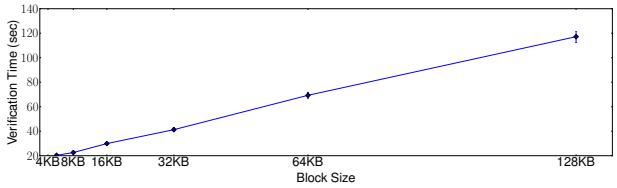
Our first examples addresses SW-POR which has been the basis of Fortress. Recall the private scheme as explained in Appendix A and suppose the secrets of the user are given to a malicious auditor who colludes with the service provider. This allows the provider to generate correct proofs without actually storing the data.

More precisely, if the auditor and therefore the service provider knows k_{prf} and the values α_j, $1 \le j \le s$, he can use the following strategy to pass the verification test: For any challenge $\{(i, \nu_i)\}$, the service provider computes $x := \sum_{(i,\nu_i)\in Q} \nu_i f_{k_{\text{prf}}}(i)$ and σ as specified in the protocol. Then, using the knowledge of α_j, the service provider chooses μ_j such that $\sigma = x + \sum_{j=1}^{s} \alpha_j \mu_j$. This response will pass the verification test but does not depend on the original data.

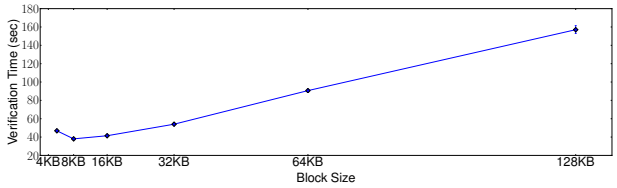
The second example is the dynamic PDP scheme by Ateniese *et al.* in [12]. In the verification phase, the data owner receives a tuple (z, v_i') from the server where v_i' is an authenticated encryption of $v = (i, z)$ where i represents the index and z is some hash value. Observe that the data owner only checks if v_i' is an encryption of (i, z) but does not check z itself. Here, a malicious service provider simply samples some random z and returns z and an encryption of (i, z). Also here, this response would pass verification but does not depend on the file.

Our final example is the PDP by Ateniese *et al.* given in [11]. In details, this PDP scheme contains a procedure called CheckProof which involves three parameters: τ, T, and ρ. In a nutshell, ρ is the hash value of $\tau^s := g^{s\cdot(\alpha_1 m_{i_1}+\ldots+\alpha_c m_{i_c})}$ where the α_i are part of the challenge and the values m_{i_j} represent the entries of the file that are requested within the challenge. The core idea is that in the setup phase, some tags $T_{i,m}$ are generated which allow to compute a value T such that T^e is equal to τ times some publicly known value. Here, e represents an RSA encryption key. A malicious service provider who knows the secrets can construct correct proofs backward. First, the provider chooses a random value τ and computes $\rho = \text{Hash}(\tau^s)$. Next, he computes τ^* as the product of τ and the publicly known values. Finally, the provider computes T such that $T^e = \tau^*$. The response (T, ρ) will pass CheckProof but does not depend on the file.

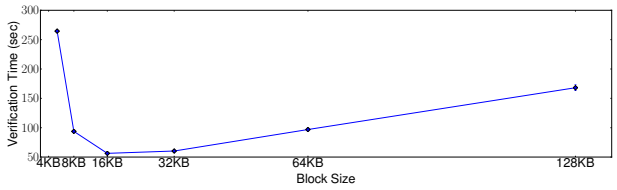
C. IMPACT OF BLOCK SIZE ON POR VERIFICATION TIME



(a) Impact of block size on the verification time of the cloud response in Fortress and in PSW.



(b) Impact of block size on the verification time of the cloud response in the public BLS-based SW scheme.



(c) Impact of block size on the verification time of the cloud response in the public RSA-based SW scheme.

Figure 6: Impact of the block size on the verification time of the cloud response in the private and public SW schemes [35].