Fair and Dynamic Proofs of Retrievability

Qingji Zheng
Department of Computer Science
University of Texas at San Antonio
qzheng@cs.utsa.edu

Shouhuai Xu
Department of Computer Science
University of Texas at San Antonio
shxu@cs.utsa.edu

ABSTRACT

Cloud computing is getting increasingly popular, but has yet to be widely adopted arguably because there are many security and privacy problems that have not been adequately addressed. A specific problem encountered in the context of cloud storage, where clients outsource their data (files) to untrusted cloud storage servers, is to convince the clients that their data are kept intact at the storage servers. An important approach to achieve this goal is called Proof of Retrievability (POR), by which a storage server can convince a client — via a concise proof — that its data can be recovered. However, existing POR solutions can only deal with static data (i.e., data items must be fixed), and actually are not secure when used to deal with dynamic data (i.e., data items need be inserted, deleted, and modified). Motivated by the need to securely deal with dynamic data, we propose the *first* dynamic POR scheme for this purpose. Moreover, we introduce a new property, called *fairness*, which is necessary and also inherent to the setting of dynamic data because, without ensuring it, a dishonest client could legitimately accuse an honest cloud storage server of manipulating its data. Our solution is based on two new tools, one is an authenticated data structure we call *range-based 2-3 trees* (rb23Tree for short), and the other is an incremental signature scheme we call *hash-compress-and-sign* (HCS for short). These tools might be of independent value as well.

Categories and Subject Descriptors

C.2.4 [**Communication Networks**]: Distributed Systems; D.4.6 [**Security and Protection**]: Authentication; H.3.4 [**Information Storage and Retrieval**]: Systems and Software

General Terms

Security

CODASPY'11, February 21–23, 2011, San Antonio, Texas, USA.

Keywords

Cloud computing, cloud storage, cloud security, proof of retrievability, 2-3 tree, ranged-based 2-3 tree, verifiable data integrity, integrity checking, outsourced storage, authenticated data structures

1. INTRODUCTION

Most people believe that cloud computing will be the next paradigm of computing. However, cloud computing has not been widely deployed arguably because there are security and privacy problems that remain to be tackled. The specific problem we focus on in this paper is: How can a cloud storage client (i.e., data owner, user) be assured that its data outsourced to a cloud storage service provider (i.e., server) is kept intact? The trivial solution is to let the client download its whole data periodically, which is unfortunately not acceptable in practice. For this purpose, efficient cryptographic techniques have been proposed. Among existing solutions (see Section 1.2 below for details on related prior work), the arguably most powerful one is the so-called Proof of Retrievability (POR), which was first introduced by Juels and Kaliski (ACM CCS'07) [15] and later significantly improved by Shacham and Waters (Asiacrypt'08) [20]. POR allows a cloud storage provider to cryptographically convince the data owner — via some succinct interactions — that its outsourced data are kept intact.

Existing POR schemes can only deal with static data and are actually not secure when used to deal with dynamic data, as we will show in Section 6.1 through a concrete attack. However, dynamic data are more realistic because they allow to insert, delete, and modify data items. The importance of maintaining dynamic data can also be justified by the fact that researchers have proposed dynamic Proof of Data Possession (PDP) (ACM CCS'09) [13], which follows the static PDP proposed by Ateniese et al. (ACM CCS'07) [2]. Note that PDP is a weaker security property. Indeed, it is already observed in [13] that accommodating dynamic data in the setting of POR is more challenging than accommodating dynamic data in the setting of PDP. Intuitively, the difficulty can be attributed to the fact that the *retrievability* property is more demanding than the *possession* property. More specifically, this is because the former additionally needs to allow the data owner to derive its data from the transcripts of successful sessions of the proof-of-retrievability protocol, which is reminiscent of the cryptographic notion of Proof of Knowledge [4].

Another problem inherent to dynamic POR is what we call *fairness*, which roughly speaking assures that a dishon-

est data owner cannot legitimately accuse an honest cloud storage service provider of manipulating its data. If the service provider has no means to prove (say, to a judge in court) that it is innocent, then the provider could be imposed with a big financial burden because a dispute of this nature may often be biased against the service provider. Note that fairness in the setting of static POR is easily solved by, for example, requiring the client to digitally sign its data before the data are outsourced to the server. In the setting of dynamic POR, however, the problem is challenging because the updated data are not at the owner's end anymore. One trivial solution is to download and sign the whole data after each update operation, which is also clearly not acceptable in practice because of the linear communication cost.

1.1 Contributions

In this paper, we introduce the concept of fair and dynamic proof of retrievability (FDPOR), a useful extension of static POR in practice. Then, we present a formal definition of FDPOR and its security properties called *soundness* and *fairness*. We explain why the extension of static POR to FDPOR is non-trivial. We discuss in detail: (i) The state-of-the-art static POR scheme is insecure when directly used in the setting of dynamic POR; (ii) We need a new authenticated data structure to ensure soundness in the setting of dynamic POR; (iii) We need a special kind of technique to ensure fairness in the setting of dynamic POR; (iv) We need to use error-correcting code in a fashion different from its counterpart in the setting of static POR. These observations guided us in designing an efficient FDPOR scheme, which simultaneously offers both retrievability and fairness in the setting of dynamic data. Specifically, our scheme is built on top of two new building-blocks, which might be of independent value.

- The first building-block is a new authenticated data structure we call *range-based 2-3 tree*, or rb23Tree for short. A rb23Tree not only inherits the properties of 2-3 trees that dynamic maintenance only incurs logarithmic complexity while allowing to prove membership, but also offers an additional important assurance that a specific value is stored at a specific leaf node.
- The second building-block is a new incremental signature scheme called *hash-compress-and-sign* or HCS for short. Our tailored incremental signature scheme is more efficient than the literature ones that operate in a more general setting because our incremental signing incurs constant hash operations (due to the use of flat trees, which are related to but separate from the range-based 2-3 trees) rather than logarithmic many hash operations. Despite the fact that hash functions can be very efficient, the improvement of our scheme would still be significant because of the harddisk input/output operations incurred by the read/write accesses associated with the hash operations.

1.2 Related Work

As mentioned above, the concept of POR was introduced by Juels and Kaliski [15], whose scheme adopted the idea of "spot-checking" [16]. This approach assumes that files are encrypted before outsourcing so that data blocks and "sentinel" blocks are indistinguishable (in a cryptographic sense). Because each query will expose a number of sentinels, this approach is limited by its bounded usage (i.e., when all sentinels have been queried). The scheme was later significantly improved by Shacham and Waters [20] by adapting a tool (implicitly introduced by Ateniese et al. [2]) now known as "homomorphic linear authenticator." Putting informally, this tool allows a data owner to affiliate data blocks with some tags, which will allow the storage provider to produce (via the homomorphism property) a succinct authenticator with respect to any random challenge vector provided by the data owner. Other extensions and improvements include [11], which however still only deals with static data. POR has been experimentally tested [8, 9].

The related concept of PDP was introduced by Ateniese et al. [2], who were motivated by various drawbacks of previous relevant approaches (we refer to [2] for a nice thorough review on earlier relevant studies). Note that the scheme of [2] is insecure in dynamic data setting because of replay attacks. In order to defeat replay attacks, one needs to utilize an authenticated tree structure that incurs logarithmic complexity. This was realized by Erway et al. [13], whose $O(\log n)$ cost is justified by the bound given in [12]. On the other hand, if it is appropriate to assume that the number of challenges issued by data owner is bounded from above and the dynamic data operations are only append-like, then a very efficient PDP scheme can be found in [3].

Organization. We define FDPOR and its security properties in Section 2. We briefly review some preliminary materials in Section 3. We present the two building-blocks in Section 4-5, respectively. We describe the main FDPOR construction in Section 6, and analyze its security in Section 7. We conclude the paper in Section 8.

2. DEFINITIONS

A data file F is divided into blocks $F = (F_1, \ldots, F_n)$ and is updated block-by-block via an update operation (op, i, α), where i is the block index, and α is the new block value, op corresponds to deletion (denoted by $\mathcal{D}$), modification (denoted by $\mathcal{M}$), or insertion (denoted by $\mathcal{I}$). Specifically,

- $(\mathcal{D}, i, \texttt{null})$ means that the i^{th} block F_i will be deleted;
- $(\mathcal{I}, i, \alpha)$ means that a new block will be inserted after the current i^{th} block F_i, namely that the new $F_{i+1} = \alpha$;
- $(\mathcal{M}, i, \alpha)$ means that the i^{th} block F_i will be replaced with new value α.

Note that the initial uploading of a file F can be achieved via a series of $\mathcal{I}$ operation; in specific schemes, it can be simplified as a simpler batch process. We stress that, unlike in the setting of static POR [15, 20], the indices of blocks in dynamic POR or FDPOR are dynamic, namely that block F_{i+1} may become block F_i after deleting block F_i, and block F_{i+1} may become block F_{t+2} after adding a new block F_{i+1}. It is also worthwhile to note that, unlike static POR, we need to divide a file into blocks in the setting of dynamic POR or FDPOR because, otherwise, it is not clear how to define file operations. Nevertheless, this does not jeopardize the resulting scheme because, for performance reason, files are always divided into blocks in the case of static POR [15, 20].

Each file F is identified through an identity fid, which remains unchanged until after F is deleted. Each F is also accompanied with some auxiliary information, denoted by

au, which can be various cryptographic tags that will be used (for example) as part of input in the process of verifying the retrievability of F. Moreover, au may have two variants: au_c is the auxiliary information stored at the client end, and au_s is the auxiliary information stored at the storage server end.

For better clarity, we assume that the communication between a client and a storage server is authenticated. This can be readily realized using a digital signature scheme or a message authentication scheme. Note that in practice the data file F is often encrypted by the client before outsourcing to the server, which justifies why the underlying channel does not have to be private.

The following definition of FDPOR is built on top of previous studies of static POR [15, 20].

Definition 1. (FDPOR; extended from [15, 20]) A FDPOR scheme consists of the following:

- FDPOR.KeyGen. This randomized algorithm takes as input a security parameter κ, and generates a collection of public keys, denoted by pk, and private/secret keys, denoted by sk. pk is publicly known but sk is kept as the client's secret.

- FDPOR.Update. This is a protocol executed between a client C on input $(pk, sk, \mathsf{fid}, \mathsf{au}_c, (op, i, \alpha))$ and a server S on input $(pk, \mathsf{fid}, F, \mathsf{au}_s)$. The protocol allows the client to update its file F according to (op, i, α). At the end of the protocol execution, if the server S accepts (outputs 1), it updates according to (op, i, α) the client's file F (identified by fid) to F' and the auxiliary information au_s to au'_s; otherwise, S aborts and outputs $(0, \perp, \perp)$. In addition, the server S sends to the client C possibly updated auxiliary information au'_c as well as some additional information evi that will allow C to verify that S faithfully updated C's file F according to (op, i, α). The client C verifies whether to accept that the server has faithfully updated the file based on $pk, sk, \mathsf{fid}, \mathsf{au}_c, (op, i, \alpha), \mathsf{au}'_c, \mathsf{evi}$. If C accepts (outputs 1), C also updates its local auxiliary information au_c to au'_c; otherwise, C aborts and outputs $(0, \perp)$. Formally, we denote this by:

  $$((b_C, \mathsf{au}'_c); (b_S, F', \mathsf{au}'_s)) \leftarrow (\mathrm{C}(pk, sk, \mathsf{fid}, \mathsf{au}_c, (op, i, \alpha)) \leftrightarrow \mathrm{S}(pk, \mathsf{fid}, F, \mathsf{au}_s)),$$

  where $b_C \in \{0, 1\}$, $b_S \in \{0, 1\}$, $(b_C = 1, \mathsf{au}'_c)$ is the client C's output and $(b_S = 1, F', \mathsf{au}'_s)$ is the server S's output upon a successful execution of the FDPOR.Update protocol.

- FDPOR.Por. This is a protocol between a prover algorithm P and a verifier algorithm V, by which P (which often is the server S) convinces V (which can be, but not necessarily, the client C) about the retrievability of a data file F. For the execution, P takes as input pk, the file F (corresponding to file identity fid provided by V) and auxiliary information au_s. V takes as input pk, possibly also sk, and the auxiliary information au_c. When the execution of the protocol halts, V outputs 1 (meaning that the file is kept intact at the storage server) and 0 otherwise. Formally, we denote the protocol as $b \leftarrow (\mathrm{P}(pk, \mathsf{fid}, F, \mathsf{au}_s) \leftrightarrow \mathrm{V}(pk, sk, \mathsf{fid}, \mathsf{au}_c))$, where $b \in \{0, 1\}$.

We require a FDPOR protocol to be correct, sound (against dishonest storage provider) and fair (against dishonest client). Intuitively, we say a FDPOR scheme is correct if both the client and the storage provider are honest (i.e., execute according to the protocols), then FDPOR.Update and FDPOR.Por always execute successfully as long as the previous system state and the operation (op, i, α) are legitimate. Formally, we say a FDPOR scheme is correct if:

$$\Pr\left[\begin{array}{l}(pk, sk) \leftarrow \mathsf{FDPOR.KeyGen}(1^\kappa);\\ \forall\ \mathsf{fid}, F, v, \mathsf{au}_c, \mathsf{au}_s \text{ s.t.}\\ 1 \leftarrow (\mathrm{P}(pk, \mathsf{fid}, F, \mathsf{au}_s) \leftrightarrow \mathrm{V}(pk, sk, \mathsf{fid}, \mathsf{au}_c)) :\\ \forall\ (op, i, \alpha),\\ (((1, \mathsf{au}'_c); (1, F', \mathsf{au}'_s)) \leftarrow\\ \quad (\mathrm{C}(pk, sk, \mathsf{fid}, \mathsf{au}_c, (op, i, \alpha)) \leftrightarrow \mathrm{S}(pk, \mathsf{fid}, F, \mathsf{au}_s))) \wedge\\ (1 \leftarrow (\mathrm{P}(pk, \mathsf{fid}, F', \mathsf{au}'_s) \leftrightarrow \mathrm{V}(pk, sk, \mathsf{fid}, \mathsf{au}'_c)))\end{array}\right] = 1$$

The following definition of soundness is built on [20]. Intuitively, a FDPOR scheme is sound if any cheating storage provider, who successfully executed the FDPOR.Por protocol with an honest verifier, is actually keeping the data file F intact. This is captured, as in the case of Proof of Knowledge [4], by that an extractor algorithm that can efficiently derive the data file F. Formally, we consider an adversary $\mathcal{A}$ that operates in an environment, which is bootstrapped via an honest execution of FDPOR.KeyGen and gives the resulting pk to $\mathcal{A}$. Furthermore, $\mathcal{A}$ can query oracles corresponding to the protocol FDPOR.Update, which causes the storage/update of a file, and to the protocol FDPOR.Por, which causes the execution of the proof-of-retrievability protocol with respect to a file. Finally, $\mathcal{A}$ outputs a cheating prover P' with respect to a file F. The cheating prover P' is ϵ-*admissible* if it succeeds in executing FDPOR.Por with an honest verifier V with a non-negligible (in κ) probability ϵ, where the probability is taken over the coins of both the cheating prover P' and the honest verifier V.

Definition 2. (soundness) Let ϵ be a non-negligible function of κ. We say a FDPOR scheme is ϵ-sound if there exists an extraction algorithm EXTRACTOR that, for every adversary $\mathcal{A}$ that outputs an ϵ-*admissible* cheating prover P' for a file F, can recover F from P' except a negligible probability. Formally, we define

$$\mathsf{Retrievability}_{\mathcal{A}}^{\mathsf{FDPOR}}(\kappa) = \Pr\left[\begin{array}{l}(pk, sk) \leftarrow \mathsf{FDPOR.KeyGen}(1^\kappa);\\ \mathrm{P}' \leftarrow \mathcal{A}^{\mathsf{FDPOR.Update}, \mathsf{FDPOR.Por}}(pk);\\ \Pr[1 \leftarrow (\mathrm{P}' \leftrightarrow \mathrm{V}(pk, sk, \mathsf{fid}, \mathsf{au}_c))] \geq \epsilon;\\ F' \leftarrow \mathrm{EXTRACTOR}(pk, sk, \mathsf{au}_c, \mathrm{P}') :\\ F = F'\end{array}\right],$$

where $\mathcal{A}$ may play the role of the server S in the oracle access to FDPOR.Update and the role of the prover P in the oracle access to FDPOR.Por. We say a FDPOR scheme is ϵ-sound if $\mathsf{Retrievability}_{\mathcal{A}}^{\mathsf{FDPOR}}$ is a non-negligible.

The following definition of fairness is newly introduced. Intuitively, a FDPOR scheme is fair if no honest storage server will be legitimately accused of manipulating any client's data. This is captured by ensuring that the dishonest client, who essentially knows everything including the private keys, is still unable to generate $F' \neq F''$, but F' and F'' corresponds to successful executions of the FDPOR.Update protocol starting from the same sever state $(pk, \mathsf{fid}, F, \mathsf{au}_s)$. The idea is that, if a dishonest client can find such F' and F'',

then the client can demonstrate that the server has manipulated its data from F' to F'', or from F'' to F'.

Definition 3. (fairness) Let adversary $\mathcal{A}$ have access to oracle FDPOR.KeyGen for generating keys, to oracle FDPOR.Update while $\mathcal{A}$ may play the role of the client C and choose file operations adaptively, to oracle FDPOR.Por while $\mathcal{A}$ may play the role of the verifier V. Define

$$\mathsf{UnFairness}_{\mathcal{A}}^{\mathsf{FDPOR}}(\kappa) = \Pr\left[\begin{array}{l}(pk, \mathsf{fid}, F, F', F'', \mathtt{au}_s) \leftarrow \\ \quad \mathcal{A}^{\mathsf{FDPOR.KeyGen},\mathsf{FDPOR.Update},\mathsf{FDPOR.Por}}(1^\kappa) : \\ F' \neq F'' \quad \wedge \\ (((1,*);(1,F',*)) \leftarrow \\ \quad (\mathrm{C}(pk, sk, \mathsf{fid}, \mathtt{au}_c, (op, i, \alpha)) \leftrightarrow \mathrm{S}(pk, \mathsf{fid}, F, \mathtt{au}_s)))\wedge \\ (((1,*);(1,F'',*)) \leftarrow \\ \quad (\mathrm{C}(pk, sk, \mathsf{fid}, \mathtt{au}_c, (op, i, \alpha)) \leftrightarrow \mathrm{S}(pk, \mathsf{fid}, F, \mathtt{au}_s)))\wedge \end{array}\right],$$

where $*$ indicates that no restriction on the respective value is imposed. We say a FDPOR protocol is fair, if for every polynomial-time algorithm $\mathcal{A}$, the probability $\mathsf{UnFairness}_{\mathcal{A}}^{\mathsf{FDPOR}}$ is negligible in κ.

3. PRELIMINARIES

A standard 2-3 tree is a tree where all leaves are at the same height and each node (except leaves) has two or three children. It has the nice property that leaf removal and insertion incur only logarithmic complexity because these operations only involve the nodes related to the path from the relevant leaf to the root. We refer to, for example, [1] for a comprehensive treatment of 2-3 trees. For the sake of a modular construction and representation, let us briefly review the maintenance algorithms for a standard 2-3 tree T; the details of these algorithms can be found in, for example, [1].

- 23Tree.M$(\mathcal{T}, i, e)$: This algorithm modifies the value of the i^{th} leaf e.
- 23Tree.I$(\mathcal{T}, i, e)$: This algorithm inserts a leaf storing e immediately after the i^{th} leaf.
- 23Tree.D$(\mathcal{T}, i)$: This algorithm deletes the i^{th} leaf.

Let $h : \{0,1\}^* \rightarrow \{0,1\}^\ell$ be a collision-resistant hash function. Let $\{f_k\}_k$ be a family of secure pseudorandom function (PRF) $f : \{0,1\}^\ell \times \{0,1\}^* \rightarrow \{0,1\}^\ell$. Let $\mathsf{SIG} = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Ver})$ be a secure digital signature scheme, where SIG.KeyGen is the key generation algorithm, SIG.Sign is the signing algorithm, and SIG.Ver is the signature verification algorithm. These primitives and their security properties are standard and can be found for example in [14].

The motivation of incremental cryptography [5] was to speed up the cryptographic computation, especially when dealing with a large amount of data. Notable incremental cryptosystems including [5, 6, 7, 18].

Let $ECC(m, z, d)$ be any error-correcting code that encodes m symbols (e.g., elements in $\mathbb{Z}_p$ for an $\ell+1$ bits prime p) into z symbols such that the distance between any pair of codewords is at least d.

4. BUILDING-BLOCK I: RANGE-BASED 2-3 TREES

There have been some authenticated data structures that are constructed based on 2-3 trees [19, 6, 18]. Specifically, [19] used a variant 2-3 tree for membership query; [6, 18] used another variant 2-3 tree for authenticating leaves while allowing fast searching. However, these data structures are not sufficient for our purpose because we not only need to authenticate the value at a leave, but also want to authenticate the index of the leave (i.e., we want to authenticate that a specific value is exactly stored at a specific leave). This motivates us to propose *range-based 2-3 trees*, or rb23Tree, that have the desired properties.

4.1 Constructing Range-Based 2-3 Trees

For a set of n elements $S = \{e_1, e_2, \ldots, e_n\}$, we can create a 2-3 tree of n leaves such that each node stores $(l(v), r(v), x(v))$, where

- $l(v)$ is the height of node v, with each leaf having height 1.
- $r(v)$ is the *range value* of v, namely the number of leaves corresponding to the subtree rooted at v. If $v = \mathtt{null}$, define $r(v) = 0$. If v is a leaf, define $r(v) = 1$.
- $x(v)$ is the *authentication value* of v. It is defined as

$$x(v) = \begin{cases} h(l(v)||r(v)||x(c_1)||x(c_2)||x(c_3)) & \text{if } l(v) > 1 \\ e_i & v \text{ is the } i^{th} \text{ leaf} \\ 0 & v = \mathtt{null} \end{cases},$$

  where $||$ is the concatenation operation, c_1, c_2, c_3 are the three left-to-right children of an inner node v (when v has two children c_1 and c_2, we treat $c_3 = \mathtt{null}$), and h is a collision-resistant hash function.

Let $\min(v)$ and $\max(v)$ denote the minimum and maximum leaf indices that can be reached via node v, respectively. Given $\min(v)$, $\max(v)$, and the ranges of v's children $r(c_1), r(c_2), r(c_3)$, we have:

$$\begin{aligned} \min(c_1) &= \min(v), \\ \max(c_1) &= \min(c_1) + r(c_1) - 1, \\ \min(c_2) &= \max(c_1) + 1, \\ \max(c_2) &= \min(c_2) + r(c_2) - 1, \\ \min(c_3) &= \max(c_3) - r(c_3) + 1, \\ \max(c_3) &= \max(v). \end{aligned}$$

Observe that a node v is on the path π_i from the i^{th} leaf to the root if and only if $i \in \{\min(v), \max(v)\}$. We call π_i a *proof path* for locating the i^{th} leaf by traversing the path starting at the root.

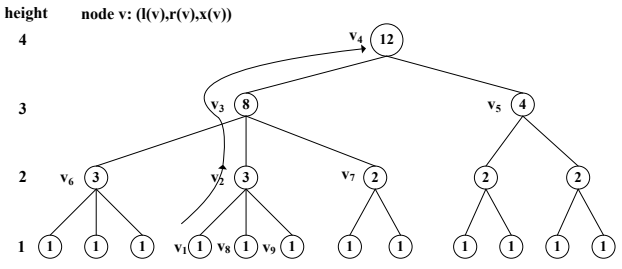

Figure 1: An example of range-based 2-3 tree. Each inner node v stores $(l(v), r(v), x(v))$.

Example. Let us look at the example tree shown in Figure 1. In this example, we have $\min(v_4) = 1$ and $\max(v_4) = 12$. Suppose we want to find the 4^{th} leaf in Figure 1. We observe:

$$\begin{array}{lllll} \min(v_4) = 1 & \wedge & \max(v_4) = 12 & \Longrightarrow & v_4 \in \pi_4 \\ \min(v_3) = 1 & \wedge & \max(v_3) = 8 & \Longrightarrow & v_3 \in \pi_4 \\ \min(v_6) = 1 & \wedge & \max(v_6) = 3 & \Longrightarrow & v_6 \notin \pi_4 \\ \min(v_2) = 4 & \wedge & \max(v_2) = 6 & \Longrightarrow & v_2 \in \pi_4 \\ \min(v_1) = 4 & \wedge & \max(v_1) = 4 & \Longrightarrow & v_1 \in \pi_4. \end{array}$$

As a result, we find the path (represented as an ordered set) $\pi_4 = \{v_1, v_2, v_3, v_4\}$.

4.2 Utilizing Range-Based 2-3 Trees

Now let us put the above tree-construction method into the context of the present paper. Suppose a range-based 2-3 tree with n leaves $e_1, \ldots, e_n$. The tree was constructed by a client using the above method, and was outsourced to the storage server. The client always keeps $x(root)$, which is the authentication value of the root. Suppose further that the client wants to verify not only that e_i is stored at a leaf but also that e_i is stored exactly at the i^{th} leaf, where $1 \leq i \leq n$. For this purpose, the server provides a proof path (or ordered set) $\pi_i = \{v_1, \ldots, v_k\}$, where v_1 is the i^{th} leaf, v_k is the root, and each node $v_j \in \pi_i$ $(1 \leq j \leq k)$ is associated with an 8-element tuple

$$Token(v_j) = (l(v_j), r(v_j), r(c_1), x(c_1), r(c_2), x(c_2), r(c_3), x(c_3)),$$

where c_1, c_2, c_3 are v_j's three left-to-right children, and $r(c_\ell) = -1$ and $x(c_\ell) = -1$ if $c_\ell \in \pi_i$ and $l(c_\ell) > 1$, where $1 \leq \ell \leq 3$. In other words, the proof is a sequence of

$$(Token(v_1), \ldots, Token(v_k)).$$

Example (continued). Take v_1 in Figure 1 as an example. Suppose we want to verify that the value in v_1 is e_4 and the index of v_1 is 4. Then the tuples along the proof path

	$l(v)$	$r(v)$	$r(c_1)$	$x(c_1)$	$r(c_2)$	$x(c_2)$	$r(c_3)$	$x(c_3)$
v_4	4	12	-1	-1	4	$x(v_5)$	0	0
v_3	3	8	3	$x(v_6)$	-1	-1	2	$x(v_7)$
v_2	2	3	-1	-1	1	$x(v_8)$	1	$x(v_9)$
v_1	1	e_4	0	0	0	0	0	0

Table 1: Proof path for that v_1 is the 4^{th} leaf and $x(v_1) = e_4$

are shown in Table 1. Given a proof path, the clients can verify that element e_i of (ordered set) $S = \{e_1, \ldots, e_n\}$ is stored exactly at the i^{th} leaf using Algorithm 1,denoted by rb23Tree.Examine$(x(root), \pi_i)$.

4.3 Verifiably Maintaining Range-Based 2-3 Trees

The maintenance of range-based 2-3 trees is essentially the same as the maintenance of standard 2-3 trees (which was briefly reviewed in Section 3), except that we need to take care of issues relevant to our extension, namely the updating of both range values and authenticated values of the relevant nodes. Specifically, the maintenance algorithms of range-based 2-3 trees are described as follows.

- rb23Tree.P$(\mathcal{T}, i)$: This algorithm allows the client to verify that a specific element is exactly stored at the i^{th} leaf of $\mathcal{T}$. The algorithm traverses a range-based 2-3 tree $\mathcal{T}$ from the root to the i^{th} leaf, and returns a proof path (ordered set) $\pi_i = \{v_1, \ldots, v_k\}$, where each v_j is associated with an 8-element tuple as mentioned above.

- rb23Tree.M$(\mathcal{T}, i, e)$: This algorithm allows the client to update the value of i^{th} leaf of $\mathcal{T}$ to the new value e and refresh $\mathcal{T}$ correctly. It executes as follows.

  1. Call rb23Tree.P$(\mathcal{T}, i)$, which returns (ordered set) $\pi_i = \{v_1, \ldots, v_k\}$, where each v_j is associated with an 8-element tuple as mentioned above.
  2. Call 23Tree.M$(\mathcal{T}, i, e)$ to replace the value of i^{th} leaf of $\mathcal{T}$ with a new value e, then update the range value $r(v_j)$ and authentication value $x(v_j)$ for $j = 1, \ldots, k$. Denote the updated range-based 2-3 tree as $\mathcal{T}'$.
  3. Call 23Tree.P$(\mathcal{T}', i)$, which returns updated proof path $\pi_i' = \{v_1, \ldots, v_k\}$, where each v_j is associated with an updated 8-element tuple.
  4. Return π_i and π_i'.

- rb23Tree.I$(\mathcal{T}, i, e)$: This algorithm allows the client to insert a new leaf with the value e after i^{th} leaf of $\mathcal{T}$ correctly. It executes as follows.

  1. Call rb23Tree.P$(\mathcal{T}, i)$, which returns proof path (ordered set) $\pi_i = \{v_1, \ldots, v_k\}$, where each v_j is associated with an 8-element tuple as mentioned above.

Algorithm 1 rb23Tree.Examine$(x(root), \pi_i)$: This algorithm allows a client, who knows $x(root)$, to verify the i^{th} element e_i of (ordered set) $S = \{e_1, \ldots, e_n\}$ is stored exactly at the i^{th} leaf by examining proof path (ordered set) $\pi_i = \{v_1, \ldots, v_k\}$ provided by the server

```
initialize array position[1..k] = 0
  {position tracks the index of the leaf with x(·) = e_i}
initialize array f[1..k] = 0
  {f[j] tracks x(v_j) where v_j ∈ π_i}
position[1] := 1
f[1] := e_i
for j = 2, ..., k do
    {v_i has three children c_1, c_2, c_3}
  if r(c_1) = −1 and x(c_1) = −1 then
    position[j] := position[j − 1]
    f[j] := h(l(v_i), r(v_i), f_{i−1}, x(c_2), x(c_3))
  end if
  if r(c_2) = −1 and x(c_2) = −1 then
    position[j] := position[j − 1] + r(c_1)
    f[j] := h(l(v_j), r(v_j), x(c_1), f_{i−1}, x(c_3))
  end if
  if r(c_3) = −1 and x(c_3) = −1 then
    position[j] := position[j − 1] + r(c_1) + r(c_2)
    f[j] := h(l(v_j), r(v_j), x(c_1), x(c_2), f_{j−1})
  end if
end for
if f[k] = x(root) and i = position[k] then
  return true
else
  return false
end if
```

2. Call $\mathsf{23Tree.I}(\mathcal{T}, i, e)$ to insert a new leaf with the value e after i^{th} leaf of $\mathcal{T}$, then update range value $r(v_j)$ and authentication value $x(v_j)$ for $j = 1, \ldots, k$. Denote the updated range-based 2-3 tree as $\mathcal{T}'$.
3. Call $\mathsf{23Tree.P}(\mathcal{T}', i+1)$, which returns updated proof path $\pi_i' = \{v_1', \ldots, v_{k'}'\}$, where each v_j is associated with an updated 8-element tuple.
4. Return π_i and π_i'.

- $\mathsf{rb23Tree.D}(\mathcal{T}, i)$: This algorithm allows the client to delete the i^{th} leaf of $\mathcal{T}$ correctly. It executes as follows.
  1. Call $\mathsf{rb23Tree.P}(\mathcal{T}, i)$, which returns proof path (ordered set) $\pi_i = \{v_1, \ldots, v_k\}$, where each v_j is associated with an 8-element tuple as mentioned above.
  2. Call $\mathsf{rb23Tree.P}(\mathcal{T}, i+1)$, which returns proof path (ordered set) $\pi_{i+1} = \{v_1, \ldots, v_k\}$, where each v_j is associated with an 8-element tuple as mentioned above.
  3. Call $\mathsf{23Tree.D}(\mathcal{T}, i)$ to delete the i^{th} leaf of $\mathcal{T}$, then update the range value $r(v_j)$ and authentication value $x(v_j)$ for $j = 1, \ldots, k$. Denote the updated range-based 2-3 tree as $\mathcal{T}'$.
  4. Call $\mathsf{23Tree.P}(\mathcal{T}', i)$, which returns updated proof path $\pi_i' = \{v_1', \ldots, v_{k'}'\}$, where each v_j is associated with an updated 8-element tuple.
  5. Return π_i, π_{i+1}, π_i'.
- $\mathsf{rb23Tree.Ver}((op, i, e), \pi_i, \pi_{i+1}, \pi_i')$: This algorithm allows a client, who knows $x(root)$, to verify π_i, π_{i+1}, π_i' that are received from the server at the end of executing one of the above update algorithms, where possibly $\pi_{i+1} = \texttt{null}$. The client executes as follows.
  1. If $\mathsf{rb23Tree.Examine}(x(root), \pi_i)$ returns true, meaning that π_i is a legitimate proof path, then it executes the following; otherwise, it aborts.
  2. If $op \in \{\mathcal{M}, \mathcal{I}\}$, replace $x(v_1')$ with e where $\pi_i' = (v_1', \ldots, v_{k'}')$, and execute $\mathsf{rb23Tree.Examine}$ $(x(v_k'), \pi_i')$.
  3. If $op = \mathcal{D}$, execute $\mathsf{rb23Tree.Examine}(x(root), \pi_{i+1})$. If $\mathsf{rb23Tree.Examine}(x(root), \pi_{i+1})$ returns true, replace $x(v_1')$ with $x(v_1)$, where $\pi_{i+1} = (v_1, \ldots, v_k)$ and $\pi_i' = (v_1', \ldots, v_{k'}')$, and execute $\mathsf{rb23Tree.Examine}$ $(x(v_k'), \pi_i')$.

5. BUILDING-BLOCK II: A TAILORED INCREMENTAL SIGNATURE SCHEME

Recall that $h : \{0,1\}^* \to \{0,1\}^\ell$ is a collision-resistant hash function, and $\mathsf{SIG} = (\mathsf{SIG.KeyGen}, \mathsf{SIG.Sign}, \mathsf{SIG.Ver})$ is a secure digital signature scheme. Suppose W is a data file that we want to protect via an incremental signature scheme. We divide W into n blocks, namely $W = (W_1, \ldots, W_n)$. Let $(G, \bigodot)$ be a group, which can be, for example, $\mathbb{Z}_p$ such that $W_i \in \mathbb{Z}_p$ for $1 \le i \le n$.

In what follows we specify our incremental signature scheme $\mathsf{HCS} = (\mathsf{HCS.KeyGen}, \mathsf{HCS.InitSign}, \mathsf{HCS.IncSign}, \mathsf{HCS.Ver})$.

- $\mathsf{HCS.KeyGen}(1^\kappa)$: This algorithm executes $\mathsf{SIG.KeyGen}$ by taking as input security parameter κ to generate a pair of public and secret keys (PK, SK).
- $\mathsf{HCS.InitSign}(\text{SK}, W)$: This algorithm executes as follows in the case of a new file:
  1. For $1 \le i \le n$, we define
  $$H_i = h(W_i).$$
  2. Construct a range-based 2-3 tree $\mathcal{T}$ by storing the n values $H_1, \ldots, H_n$ at the corresponding leaves.
  3. Apply h to each pair of $(H_i||H_{i+1})$ for $1 \le i \le n-1$ to obtain
  $$\lambda_i \leftarrow h(H_i||H_{i+1}).$$
  where $||$ is the concatenation operation.
  4. Take group operation $\bigodot$ on $\lambda_1, \ldots, \lambda_{n-1}$ to obtain
  $$\lambda = \bigodot_{i=0}^{i=n-1} \lambda_i.$$
  5. Let $\delta \leftarrow \mathsf{SIG.Sign}(\text{SK}, \lambda)$ and return $\mathcal{T}, \lambda$ and δ.
- $\mathsf{HCS.IncSign}(\text{SK}, \mathcal{T}, \lambda, (op, i, \alpha), H_{i-1}, H_i, H_{i+1})$: This algorithm executes as follows in the case of an update operation (without loss of generality, assume $2 \le i \le n-1$):
  1. $op = \mathcal{D}$: Update the signature as follows:
  $$\lambda' = \lambda \bigodot h^{-1}(H_{i-1}||H_i) \bigodot h^{-1}(H_i||H_{i+1}) \bigodot h(H_{i-1}||H_{i+1}),$$
  $$\delta' = \mathsf{SIG.Sign}(\text{SK}, \lambda').$$
  and call $\mathsf{rb23Tree.D}(\mathcal{T}, i)$. Denote the updated range-based 2-3 tree as $\mathcal{T}'$
  2. $op = \mathcal{I}$: Update the signature as follows:
  $$\lambda' = \lambda \bigodot h^{-1}(H_i||H_{i+1}) \bigodot h(H_i||H(\alpha)) \bigodot h(h(\alpha)||H_{i+1}),$$
  $$\delta' = \mathsf{SIG.Sign}(\text{SK}, \lambda').$$
  and call $\mathsf{rb23Tree.I}(\mathcal{T}, i, \alpha)$. Denote the updated range-based 2-3 tree as $\mathcal{T}'$
  3. $op = \mathcal{M}$: Update the signature as follows:
  $$\lambda' = \lambda \bigodot h^{-1}(H_{i-1}||H_i) \bigodot H^{-1}(H_i||H_{i+1}) \bigodot h(H_{i-1}||h(\alpha)) \bigodot h(H(\alpha)||H_{i+1}),$$
  $$\delta' = \mathsf{SIG.Sign}(\text{SK}, \lambda').$$
  and call $\mathsf{rb23Tree.M}(\mathcal{T}, i, \alpha)$. Denote the updated range-based 2-3 tree as $\mathcal{T}'$

  Then return $\mathcal{T}', \lambda'$ and δ'.
- $\mathsf{HCS.Ver}(\text{PK}, W, \mathcal{T}, \lambda, \delta, (op, i, \alpha))$: This algorithm executes as follows (without loss of generality, assume $2 \le i \le n-1$):
  1. If verifying the file W at the first time:
  $$H_i = h(W_i)\ 1 \le i \le n,$$
  $$\lambda_i = h(H_i||H_{i+1})\ 1 \le i \le n-1,$$
  $$\lambda' = \bigodot_{i=1}^{n-1} \lambda_i.$$
  2. Else the verifier possesses $\mathcal{T}, \lambda, (op, i, \alpha)$
     - $op = \mathcal{D}$: execute the followings:
     $$\lambda' = \lambda \bigodot h^{-1}(H_{i-1}||H_i) \bigodot h^{-1}(H_i||H_{i+1}) \bigodot h(H_{i-1}||H_{i+1}).$$

– $op = \mathcal{I}$: execute the following

$$\lambda' = \lambda \bigodot h^{-1}(H_i||H_{i+1}) \bigodot h(H_i||H(\alpha)) \bigodot h(h(\alpha)||H_{i+1}).$$

– $op = \mathcal{M}$: execute the following

$$\lambda' = \lambda \bigodot h^{-1}(H_{i-1}||H_i) \bigodot H^{-1}(H_i||H_{i+1}) \bigodot h(H_{i-1}||h(\alpha)) \bigodot h(H(\alpha)||H_{i+1}).$$

In either case, execute $\mathsf{SIG.Ver}(\text{PK}, \lambda', \delta)$ and return its outcome.

6. MAIN CONSTRUCTION

Recall that a file F is divided into n blocks, denoted by $F = (F_1, \ldots, F_n)$, where each block F_i contains m symbols of elements in $\mathbb{Z}_p$ for some large prime P, namely $F_i = (F_{i1} \cdots F_{im})$ with $F_{ij} \in \mathbb{Z}_p$ for $1 \leq j \leq m$. To be specific, we can instantiate the $(G, \bigodot)$ mentioned in building-block-II (Section 5) as $(\mathbb{Z}_p, \cdot)$, while noting that $(G, \bigodot)$ can be instantiated as other groups as well.

6.1 Design Rationale

A brief review of static POR. We now briefly review the static POR proposed in [20], which used a kind of homomorphic linear authenticator (HLA) and serves as a starting point for the present paper. Partition each data file into blocks $F = (F_1, \ldots, F_n)$. Assume each block is a single symbol $F_i \in \mathbb{Z}_p$ for some large prime p (this is for the purpose of highlighting the basic idea, while in the actual scheme each F_i will consist of multiple symbols). The client chooses a random $\theta \in \mathbb{Z}_p$ and a key k for pseudorandom function f, which are the client's secrets. The client computes an authentication tag for each block F_i as:

$$\sigma_i = f_k(i) + \theta F_i \ \ (\text{in } \mathbb{Z}_p). \tag{1}$$

Then, the file $F = (F_1, \ldots, F_n)$ and the authentication tags $(\sigma_1, \ldots, \sigma_n)$ are stored on the server, which allow the following proof-of-retrievability protocol. The verifier (or client) chooses and sends to the prover (or server) a subset I of indices, namely $I \subseteq \{1, \ldots, n\}$, along with $|I|$ random challenges in $\mathbb{Z}_p$, denoted by $(random_i)_{i \in I}$. The server responds with

$$\sigma = \sum_{i \in I} random_i \cdot \sigma_i \quad \text{and} \quad \mu = \sum_{i \in I} random_i \cdot F_i \ \ (\text{in } \mathbb{Z}_p),$$

which are very short messages. This works because the verifier, who knows the key k, can verify if:

$$\sigma \stackrel{?}{=} \sum_{i \in I} random_i \cdot f_k(i) + \theta \cdot \mu \ \ (\text{in } \mathbb{Z}_p). \tag{2}$$

Why is the static POR not applicable to the dynamic setting? Note that the above elegant verification Eq. (2) works because the client can compute by itself $\sum_{i \in I} random_i \cdot f_k(i)$, where the i's are block indices that are *fixed* in the case of static POR. Unfortunately, this method is not applicable to the setting of dynamic POR anymore, as we now elaborate.

1. It is not secure when used to deal with dynamic data. To see this, suppose the i^{th} block F_i is updated to new value F_i', where the block index i is kept unchanged. Then, Eq. (1) implies that the server sees:

$$\begin{aligned} \sigma_i &= f_k(i) + \theta F_i \ \ (\text{in } \mathbb{Z}_p) \\ \sigma_i' &= f_k(i) + \theta F_i' \ \ (\text{in } \mathbb{Z}_p), \end{aligned}$$

where $F_i \neq F_i'$ but both are known to the server, and both σ_i and σ_i' are known to the server as well. As a consequence, the server can derive both θ and $f_k(i)$, which are the secrets of the data owner and will allow the server to cheat the data owner via Eq. (2) by (for example) choosing σ and then deriving the corresponding μ.

2. Even under some extreme cases (e.g., only insertion or deletion operations are required), the above idea still does not work because the client has to keep record of the indices of deleted blocks or the indices of yet-to-be inserted indices (otherwise, the verification is not well-defined). This requires significant (which can be proportional to n in the worst case) extra storage at the client end (this is something we want to avoid at the first place in cloud storage). One may suggest, as a rescue, to waste some storage space at the server (e.g., by setting each deleted or yet-to-be inserted block as some special value). But this would require to know the exact upper bound n, which may not be possible in many settings.

The basic ideas underlying our scheme. The core idea, by which we circumvent the above attack, is to use the following verification equation to replace Eq. (1):

$$\sigma_i = f_k(h(F_i)) + \theta F_i \ \ (\text{in } \mathbb{Z}_p), \tag{3}$$

where $h : \{0,1\}^* \to \mathbb{Z}_p$ is a collision-resistant hash function. While this does break the tight binding between the authentication tag σ_i and the block index i, it does introduce new attacks as we now explain. Specifically, let us consider an example of before and after the insertion of block F_i. Because the client should be stateless (i.e., does not keep record of the block indices that have been inserted and deleted), a dishonest server can certainly pass the proof-of-retrievability by rolling back to the system state before the insertion of F_i. Therefore, in order to take full advantage of Eq. (3), we need to make the client keep an (ideally) constant state of the data stored at the server. This calls for a data structure which has the following features:

1. It allows fast update of data, meaning that an update operation only incurs very small amount of computation at the server end, while minimizing the involvement of the client in the process.

2. It allows short "summaries" of dynamic data files. Moreover, the update of a summary incurs only small amount of computation at both the server end and, if necessary, at the client end.

3. It allows fast query of the i^{th} block F_i of a file F. This is important because for updating or querying file, we need to quickly identify F_i.

The above requirements led us to design a new data structure we call *range-based 2-3 trees*, which will be detailed in Section 4. (Note that Merkle trees [17] do not have all the required properties.)

On the other hand, in order to achieve fairness in the setting of dynamic remote data, it is intuitive to use some kind of incremental digital signature scheme. Actually, existing incremental signature schemes [5, 6, 7, 18] used 2-3 trees, which however are not sufficient for the purpose of

FDPOR as we discussed above. Nevertheless, it is possible to further extend our rb23Tree to serve the need of fairness assurance. However, this approach has the following drawback: incremental signing requires $O\log(n)$ hash operations corresponding to the path from the leaf to the root. Instead, we will build another flat tree (i.e., with height 1) directly based on the leaves of the rb23Tree, which will only incur constant hash operations when there is an update to a data file. Although hash functions can be made very efficient, this improved complexity still might be significant in practice because the hash operations will incur correspondingly the same number of read/write accesses. In other words, we use a tailored variant of incremental digital signature scheme for better performance.

Yet another important difference between dynamic POR and static POR is manifested by the way of error correcting codes is used. Specifically, in order to maintain dynamic data, we need to apply an error correct code at the block level, which is in contrast to the setting of static POR where an error correcting code is applied to the whole data file [15, 20]. Applying error-correcting code at the block lever is important because, otherwise, the accommodation of an update operation would always involve the entire data file, which is clearly not acceptable in practice.

Putting the pieces together. The high-level idea is described in Figure 2. Roughly speaking, there are four steps. The first step is to encode the original data file $F = (F_1, \ldots, F_n)$ into $F' = (F'_0, \ldots, F'_{n+1})$. The second step is to compute $H_i = h(F'_i)$ for $0 \le i \le n+1$. The third step is to construct a rb23Tree with leaves $(H_0, \ldots, H_{n+1})$, which leads to obtain the authentication value at root $x(root)$. The fourth step is to construct a flat tree for achieving fairness, where

$$\lambda = \bigodot_{j=0}^{n} \lambda_j = \bigodot_{j=0}^{n} h(H_j||H_{j+1}) = \bigodot_{j=0}^{n} h(h(F'_j)||h(F'_{j+1})).$$

This allows us to construct incremental signatures efficiently because any update operation only incurs constant complexity in deriving the new root λ'.

6.2 Construction

The scheme is described as follows.

FDPOR.KeyGen: Taking as input a primary security parameter κ and a secondary security parameter ℓ, it chooses two secret keys $k_1, k_2 \in_R \{0,1\}^\ell$, executes HCS.KeyGen by taking as input security parameter κ to generate a pair of public and secret keys (PK, SK). The keys (k_1, k_2, SK) will be kept secret by a client, and the key PK will be made public.

FDPOR.Update: This protocol allows a client C to store a file $F = (F_1, \ldots, F_n)$ at the first time at the server S (Case 1 below), or to update a file F that was already stored at the server S (Case 2 below).

Case 1: Uploading a new file F.

1. The client C executes as follows:
   (a) Generate two random blocks F_0 and F_{n+1}, with each consisting of m symbols of random elements in $\mathbb{Z}_p$.
   (b) Insert F_0 in front of F_1 and append F_{n+1} to the tail of F_n. Then, we have $F = (F_0, F_1, \ldots, F_n, F_{n+1})$.
   (c) Apply ECC to $F = (F_0, F_1, \ldots, F_n, F_{n+1})$ in a block-by-block fashion. Denote the output file by $F' = (F'_0, F'_1, \ldots, F'_n, F'_{n+1})$, where each block F'_j for $0 \le j \le n+1$ contains z symbols of elements in $\mathbb{Z}_p$. For $0 \le i \le n+1$, we define
   $$H_i = h(F'_i).$$
   (d) $(\lambda, \delta, \mathcal{T}) \leftarrow \mathsf{HCS.InitSign}(\text{SK}, F')$.
   (e) Compute *integrity tag* for the i^{th} block as follows $(0 \le i \le n+1)$:
   $$\sigma_i = f_{k_1}(H_i) + \sum_{j=1}^{z} f_{k_2}(j) F'_{ij} \qquad (\text{in } \mathbb{Z}_p)$$
   (f) Output the file $F' = (F'_0, \ldots, F'_{n+1})$ with $\mathtt{au}_c = (x(root), \lambda, \mathsf{fid})$, which is kept by the client, and $\mathtt{au}_s = (\mathsf{fid}, (\sigma_0, \ldots, \sigma_{n+1}), \mathcal{T}, \lambda, \delta)$, where fid is a unique string chosen by the client.
   (g) Send $(F', \mathtt{au}_s)$ to the server over the authenticated channel.
2. The server S executes $\mathsf{HCS.Ver}(\text{PK}, F', \mathtt{null}, \mathtt{null}, \delta, \mathtt{null})$. If it returns true, meaning that δ is a valid signature with respect to the client's public key PK and F', the server stores F' and $\mathtt{au}_s = (\mathsf{fid}, (\sigma_0, \ldots, \sigma_{n+1}), \mathcal{T}, \delta, \lambda)$; otherwise, the server outputs 0 and aborts.

Case 2: Updating an existing file F with $(op, i-1, \alpha)$, where $2 \le i \le n+1$.

Note that operation $(op, i-1, \alpha)$ with respect to the raw file F means (op, i, α) with respect to the ECC-encoded file F', where α is an ECC-encoded block of z symbols in $\mathbb{Z}_p$, namely $\alpha = (\alpha_1 \cdots \alpha_z), \alpha_i \in \mathbb{Z}_p$.There are three legitimate subcases.

- Subcase 2.1: $op = \mathcal{D}$.
  1. The client C sends three leaf indices $i-1, i, i+1$ to the server S.
  2. The server S executes $\mathsf{rb23Tree.P}(\mathcal{T}, j)$ for $j = i-1, i, i+1$ to obtain three respective proof pathes $\pi_{i-1}, \pi_i, \pi_{i+1}$, which are returned to the client. Note that $\mathcal{T}$ is the range-based 2-3 tree with leaves $H_0 = h(F'_0), \ldots, H_{n+1} = h(F'_{n+1})$.
  3. The client C executes $\mathsf{rb23Tree.Examine}(x(root), \pi_j)$ for $j = i-1, i, i+1$ in order to verify $\pi_{i-1}, \pi_i, \pi_{i+1}$ are valid with respect to the range-based 2-3 tree $\mathcal{T}$. If C accepts, it derives three block hash values H_{i-1}, H_i, H_{i+1} stored at the three leaves.
  4. The client C applies $\mathsf{HCS.IncSign}(\text{SK}, \mathcal{T}, \lambda, (op, i, \alpha), H_{i-1}, H_i, H_{i+1})$ and obtain δ' and λ'.
  5. The client C sends $(\delta', (op, i, \alpha), \mathsf{fid})$ to the server S and keeps λ'.
  6. The server S verifies δ' by executing $\mathsf{HCS.Ver}(\text{PK}, \mathtt{null}, \mathcal{T}, \lambda, \delta', (op, i, \alpha))$. If accepts, then S deletes the $(i-1)^{th}$ block, and applies $\mathsf{rb23Tree.D}(\mathcal{T}, i)$ to obtain proof pathes $(\pi_i, \pi_{i+1}, \pi'_i)$, which are then returned to C.
  7. The client C verifies $(\pi_i, \pi_{i+1}, \pi'_i)$ by executing $\mathsf{rb23Tree.ver}((op, i, \alpha), \pi_i, \pi_{i+1}, \pi'_i)$. If accepts, C updates $x(root)$ which is included in π'_i.
- Subcase 2.2: $op = \mathcal{I}$.

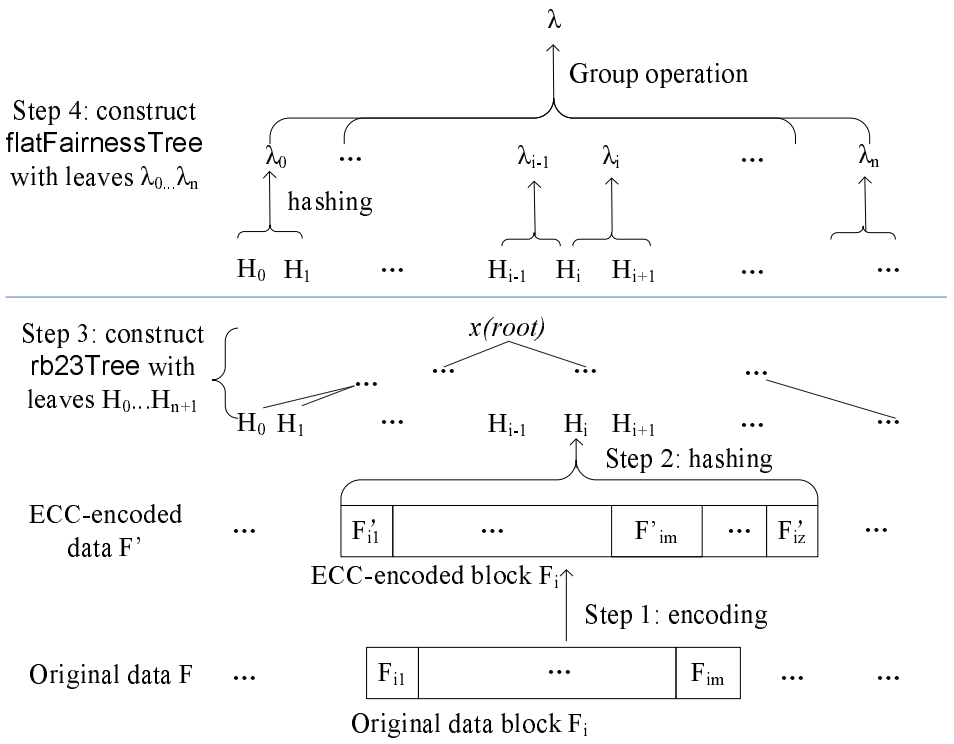

Figure 2: High-level idea behind FDPOR. Steps 1-3 correspond to the maintenance of rb23Tree with leaves $H_0, H_1, \ldots$ (for ensuring the soundness of FDPOR). Once the rb23Tree is created, Step 4 maintains the flatFairnessTree with height 1 and the same leaves $H_0, H_1, \ldots$. This is for the purpose of incremental signatures (for ensuring the fairness of FDPOR).

1. The client C sends two leaf indices $i, i+1$ to the server.
2. The server S returns π_i, π_{i+1}, which are obtained by executing $\mathsf{rb23Tree.P}(\mathcal{T}, j)$ for $j = i, i+1$.
3. The client C executes $\mathsf{rb23Tree.Examine}(\mathsf{x(root)}, \pi_j)$ for $j = i, i+1$, in order to verify π_i, π_{i+1} are valid with respect to $\mathcal{T}$. If accept, it derives two block hash values H_i, H_{i+1} stored at the three leaves.
4. The client C applies $\mathsf{HCS.IncSign}(\mathrm{SK}, \mathcal{T}, \lambda, (op, i, \alpha), \mathtt{null}, H_i, H_{i+1})$ and obtain δ' and λ'. It also computes the integrity tag for the new block α as follows:

$$\sigma_\alpha = f_{k_1}(h(\alpha)) + \sum_{j=1}^{z} f_{k_2}(j)\alpha_j.$$

   It then sends $(\delta', \sigma_\alpha, (op, i, \alpha), \mathsf{fid})$ to the server.
5. The server S verifies δ' by executing $\mathsf{HCS.Ver}(\mathrm{PK}, \mathtt{null}, \mathcal{T}, \lambda, \delta', (op, i, \alpha))$. If accepts, then it inserts the new block α after $(i-1)^{th}$ block, applies $\mathsf{rb23Tree.I}(\mathcal{T}, i, h(\alpha))$ to obtain a pair of proof pathes (π_i, π_i'), and updates δ to δ'.
6. the client C verifies π_i, π_i' by executing $\mathsf{rb23Tree.ver}((op, i, \alpha), \pi_i, \mathtt{null}, \pi_i')$. If accept, C updates $x(root)$.

- Subcase 2.3: $op = \mathcal{M}$.

1. The client C sends three leaf indices $i-1, i, i+1$ to the server.
2. The server S returns $\pi_{i-1}, \pi_i, \pi_{i+1}$, which are obtained by executing $\mathsf{rb23Tree.P}(\mathcal{T}, j)$ for $j = i-1, i, i+1$.
3. The client C execute $\mathsf{rb23Tree.Examine}(\mathsf{x(root)}, \pi_j)$ for $j = i-1, i, i+1$, in order to verify $\pi_{i-1}, \pi_i, \pi_{i+1}$ are valid with respect to $\mathcal{T}$. If C accepts, it derives two block hash values H_{i-1}, H_i, H_{i+1} stored at the three leaves.
4. The client C applies $\mathsf{HCS.IncSign}(\mathrm{SK}, \mathcal{T}, \lambda, (op, i, \alpha), H_{i-1}, H_i, H_{i+1})$ and obtains δ' and λ'. It also computes the integrity tag for the new block α as follows:

$$\sigma_i = f_{k_1}(h(\alpha)) + \sum_{j=1}^{z} f_{k_2}(j)\alpha_j.$$

   It sends $(\delta', \sigma_i, (op, i, \alpha), \mathsf{fid})$ to the server.
5. The server S verifies δ' by executing $\mathsf{HCS.Ver}(\mathrm{PK}, \mathtt{null}, \mathcal{T}, \lambda, \delta', (op, i, \alpha))$. If accept, then it updates the $(i-1)^{th}$ data block with value α, and applies $\mathsf{rb23Tree.M}(\mathcal{T}, h(\alpha))$ to obtain a pair of proof pathes (π_i, π_i').
6. The client C executes $\mathsf{rb23Tree.ver}((op, i, \alpha), \pi_i, \mathtt{null}, \pi_i')$ to verify π_i, π_i'. If accept, C updates $x(root)$.

FDPOR.Por: This protocol is executed between the client C and the server S as follows.

1. The client C chooses a set of c elements: $I = \{\alpha_1, \cdots, \alpha_c\}$, $\alpha_i \xleftarrow{r} [1, n]$, randomly selects β_i from $\mathbb{Z}_p$ for $1 \le i \le c$, and sends the challenge $\mathcal{C} = \{(\alpha_i, \beta_i)\}$ to the server.

2. The server S computes:

$$u_j = \sum_{i \in I} \beta_i F'_{ij} \qquad 1 \le j \le z$$

$$\sigma' = \sum_{i \in I} \beta_i \sigma_i$$

$$H_i = h(F'_i) \qquad i \in I$$

and $\pi_i = \mathsf{rb23Tree.P}(\mathcal{T}, i+1)$ for $i \in I$. It then sends the response $\mathcal{R} = \{u_j, \sigma', H_i, \pi_i\}$ back to the client.

3. The client C verifies the response $\mathcal{R}$ and outputs true if the two following conditions hold, and outputs false otherwise:

   - $\mathsf{rb23Tree.Examine}(x(root), \pi_i)$ returns true for $i \in I$.
   - It holds that

$$\sigma' = \sum_{i \in I} \beta_i f_{k_1}(H_i) + \sum_{j=1}^{z} f_{k_2} u_j$$

7. SECURITY ANALYSIS

The correctness of our scheme can be easily checked. Now we prove the soundness and fairness of our construction in the random model. Specifically, we show the soundness in two steps: First, we prove that no dishonest server can pass the verification with a non-negligible probability. Second, we show that our construction allows to extract the original data file from an ϵ-*admissable* prover with a certain non-negligible probability. We show the fairness of our scheme is ensured by the incremental signature scheme.

THEOREM 1. *Assume $\{f_k\}_k$ is a family of secure PRF, h is collision-resistant hash function, and $ECC(m, z, d)$ is an error-correcting code. Then one can recover the file F from the ϵ-admissable prover with at least the probability $(1 - e^{d/2-\mu}(d/2\mu)^{-d/2})^n$, where $\mu = (1-\epsilon)z/\epsilon$ with the condition $d > 2\mu$.*

PROOF. The theorem is proved by showing:

- A dishonest server cannot execute FDPOR.Por with an honest client successfully with a non-negligible probability (Lemma 1).
- Given a ϵ-*admissable* cheating prover, one can recover the original file F with probability $(1-e^{d/2-\mu}(d/2\mu)^{-d/2})^n$, where $\mu = (1-\epsilon)z/\epsilon$ with the condition $d > 2\mu$ (Lemma 2).

□

LEMMA 1. *If $\{f_k\}_k$ is a family of secure PRF and h is collision-resistant hash function, then no dishonest server can execute* FDPOR.Por *i times to convince an honest client successfully with a non-negligible probability.*

PROOF. According to Definition 2, a dishonest server successfully executes FDPOR.Por to convince the client when the client outputs 1. Let us now consider the probability that event happens. Recall that block F_i is associated with tag σ_i, where $1 \le i \le n$, that $I = \{\alpha_1, \ldots, \alpha_c\}$ is the set of block indices randomly selected by the client and $\mathcal{C} = \{(\alpha_i, \beta_i)\}_{1 \le i \le c}$ is the corresponding challenge, which caused the server to return the response $\{u'_1, \ldots, u'_z, \sigma'\} \bigcup \{H_i | (i \in I)\}$. Let $\{u_1, \ldots, u_z, \sigma\} \bigcup \{H'_i | (i \in I)\}$ be the expected response returned by an honest prover. Here we have $H_i = H'_i$ for all $i \in I$ because of the guarantee in the proof pathes verification. Hence, we have the two equations:

$$\sigma = \sum_{i \in I} \beta_i f_{k_1}(H'_i) + \sum_{j=1}^{z} f_{k_2}(j) u_j \quad (\text{in } \mathbb{Z}_p)$$

$$\sigma' = \sum_{i \in I} \beta_i f_{k_1}(H'_i) + \sum_{j=1}^{z} f_{k_2}(j) u'_j \quad (\text{in } \mathbb{Z}_p)$$

Let $\Delta\sigma = \sigma' - \sigma$ and $\Delta u_j = u'_j - u_j$, $1 \le j \le z$. Then we have

$$\Delta\sigma = \sum_{j=1}^{z} f_{k_2}(j) \Delta u_j \quad (\text{in } \mathbb{Z}_p) \tag{4}$$

In order to successfully execute FDPOR.Por the dishonest server has to provide an instance of the $z+1$ values of the $z+1$ variables $\Delta u_1, \ldots, \Delta u_z$ that satisfy Eq. (4). Because f_{k_2} is a secure PRF, the z random coefficients $f_{k_2}(1), \ldots, f_{k_2}(z)$ are randomly distributed over $\mathbb{Z}_p$ and are not known to the server. This means that any given $\Delta u_1, \ldots, \Delta u_z$ and $\Delta\sigma$ satisfying Eq. (4) with probability $1/p$. Therefore, the probability of convincing the client after i attempts is:

$$1 - (1 - 1/p)(1 - 1/(p-1)) \ldots (1 - 1/(p-i))$$

Let $1 \le i \le p/2$, then we have

$$\begin{aligned} &1 - (1 - 1/p)(1 - 1/(p-1)) \ldots (1 - 1/(p-i)) \\ &\le 1 - (1 - 2/p)^i \\ &\le 1 - (1 - 2/p) = 2/p \end{aligned}$$

For the large $\ell-$bits prime p and within polynomial attempts, the probability of convincing the client will be negligible. □

LEMMA 2. *Given an ϵ-admissible prover, one can recover the original file F with the probability $(1-e^{d/2-\mu}(d/2\mu)^{-d/2})^n$, where $\mu = (1-\epsilon)z/\epsilon$ with the condition $d > 2\mu$..*

PROOF. Since the prover is ϵ-admissible, there are at most $(1-\epsilon)nz$ symbols corrupted in the file. Let us consider the probability that a corrupted symbol exists in a given block. It has the maximum probability while other $(1-\epsilon)nz - 1$ corrupted symbols appear in other blocks. Therefore, the probability of a corrupted symbol in a given block will be less than $z/(nz - (1-\epsilon)nz) = 1/\epsilon n$.

Let us define the event X_i whether i^{th} corrupted symbol will be in a given block ($0 \le i \le (1-\epsilon)nz$) , and $X_i = 1$ means i^{th} corrupted symbol is in the given block. Then $\Pr[X_i = 1] \le 1/\epsilon n$ So $X_1, \ldots, X_{(1-\epsilon)nz}$ are the independent Bernoulli variables. Let us consider the number of corrupted symbols being in the given block, denoted by X. We have $X = \sum_{i=1}^{(1-\epsilon)nz} X_i$ and $\mathbf{E}[X] = \mu \le (1-\epsilon)z/\epsilon$.

According to the Chernoff Bound [10], if $X_1, \ldots, X_N$ are independent bernoulli variables, then for $X = \sum_{i=1}^{N} X_i$ and

any $\delta > 0$, $\Pr[X > (1+\delta)\mu] < (e^{\delta}/(1+\delta)^{(1+\delta)})^{\mu}$ with $\mu = \mathbf{E}[X]$.

For a given block, if it is not recoverable, it should contains at least $d/2$ corrupted symbols. Let $(1+\delta)\mu = d/2$ and have $\delta = d/2\mu - 1$. So $\Pr[X > d/2] = \Pr[X > (1+\delta)\mu] \leq (e^{\delta}/(1+\delta)^{(1+\delta)})^{\mu}$, by letting $\delta = d/2\mu - 1$ and $\mu = (1-\epsilon)z/\epsilon$. By simplification, we have $\Pr[X > d/2] \leq e^{d/2-\mu}(d/2\mu)^{-d/2}$ where $\mu = (1-\epsilon)z/\epsilon$ and under condition $d > 2\mu$. Hence, the probability of recovering(or retrieving) a given block will be at least $1 - e^{d/2-\mu}(d/2\mu)^{-d/2}$. Because the file is composed of n blocks, the probability of retrieving the original file will be at least $(1 - e^{d/2-\mu}(d/2\mu)^{-d/2})^n$ where $\mu = (1-\epsilon)z/\epsilon$ with the condition $d > 2\mu$. □

Fairness of our FDPOR scheme is based on the hardness of the so-called *balance problem*, which is reviewed below.

LEMMA 3. ([7]; the balance problem and its hardness) *Given a group G, and elements $a_1, \ldots, a_n$ randomly selected from G, find disjoint subsets $\mathcal{U}, \mathcal{W} \subseteq \{1, \ldots, n\}$, s.t. $\bigodot_{i\in\mathcal{U}} a_i = \bigodot_{j\in\mathcal{W}} a_j$, $\mathcal{U}, \mathcal{W}$ not both empty, where $\bigodot$ is the group operation. When choosing G as $\mathbb{Z}_p$ for a suitable prime P, the hardness of the balance problem instantiated over $(\mathbb{Z}_p,)$ would be equal to, or even greater than, the discrete logarithm problem in $\mathbb{Z}_p$.*

THEOREM 2. *Assume h is a collision-resistant hash function and* SIG *is a secure signature scheme, our* FDPOR *scheme is fair.*

PROOF. According to Definition 3, a dishonest client may break the fairness of the scheme only when it can generate two files F and F', such that (i) $F \neq F'$, and (ii) the honest server verifies that $1 \leftarrow \mathsf{HCS.Ver}(\text{PK}, F, \mathcal{T}, \lambda, \delta, \texttt{null})$ and $1 \leftarrow \mathsf{HCS.Ver}(\text{PK}, F', \mathcal{T}', \lambda', \delta, \texttt{null})$.

Let us first review the signing process: computing hash value for each file block, $H_i = h(F_i)$, hashing $(H_i H_{i+1})$, i.e.. $\lambda_i = h(H_i||H_{i+1})$, then taking group operation on λ_i, namely $\lambda = \bigodot \lambda_i$. At last Let $\mathsf{SIG.Sign}(\text{SK}, \lambda) \rightarrow \delta$. We denote this process by $F \rightarrow \mathcal{T} \rightarrow \lambda \rightarrow \delta$. Here $\mathcal{T}$ is a range-based 2-3 tree which stores H_i at the leaves. Two range-based 2-3 trees are said to be identical, denoted by $\mathcal{T} = \mathcal{T}'$, if both have the same number of leaves, and the corresponding leaves store the same hash value.

Now Let us consider the probability of the dishonest client breaking the fairness.

Let us consider the probability of the dishonest client successfully generating (F, F', δ). It can achieve this goal with three possible means:

- It can find two different λ and λ' such that
$$1 \leftarrow \mathsf{SIG.Ver}(\text{PK}, \lambda, \delta) \wedge 1 \leftarrow \mathsf{SIG.Ver}(\text{PK}, \lambda', \delta).$$
This is investigate in Case 1 below.
- It can obtain two distinct $\mathcal{T}$ and $\mathcal{T}'$ but get identical $\lambda = \lambda'$. This is investigated in Case 2 below.
- It can obtain two distinct file F and F', but have identical $\mathcal{T}$ and T'. This is investigated in Case 3 below.

Case 1: $\lambda \neq \lambda'$. Let us define event E_1 as

$$\{\lambda \neq \lambda' | 1 \leftarrow \mathsf{SIG.Ver}(\text{PK}, \lambda', \delta) \wedge 1 \leftarrow \mathsf{SIG.Ver}(\text{PK}, \lambda', \delta)\}.$$

Because SIG is a secure signature scheme, $\Pr[E_1]$ is negligible.

Case 2: $\mathcal{T}' \neq \mathcal{T}$ but $\lambda = \lambda'$. Let us define event E_2 as

$$\{\lambda = \lambda' | \mathcal{T} \neq \mathcal{T}' \wedge \ \mathcal{T} \rightarrow \lambda \ \wedge \ \mathcal{T}' \rightarrow \lambda'\}.$$

Let us consider the probability $\Pr[E_2]$.

Assume that there exists a polynomial-time algorithm $\mathcal{A}$ that can obtain two different strings $x = x_1 x_2 \cdots x_n$ and $y = y_1 y_2 \cdots y_m$, which are stored at the leaves of $\mathcal{T}$ and $\mathcal{T}'$ such that $\lambda = \bigodot_{i=1}^{n-1} h(x_i||x_{i+1})$, $\lambda' = \bigodot_{j=1}^{m-1} h(y_j||y_{j+1})$ and $\lambda = \lambda'$. Then we have

$$h(x_1||x_2) \bigodot h(x_2||x_3) \cdots \bigodot h(x_{n-1}||x_n)$$
$$= h(y_1||y_2) \bigodot h(y_2||y_3) \cdots \bigodot h(y_{m-1}||y_m).$$

Let

$$\mathcal{U} = \{\alpha_i = h(x_i||x_{i+1}), 1 \leq i \leq n-1\}$$

and

$$\mathcal{W} = \{\beta_j = h(y_j||y_{j+1}), 1 \leq j \leq m-1\},$$

where α_i, β_j are randomly distributed over the $\mathbb{Z}_p$. Then we have

$$\bigodot_{i=1}^{n-1} \alpha_i = \bigodot_{j=1}^{m-1} \beta_j.$$

Since $x \neq y$, we have $\mathcal{U} \neq \mathcal{W}$. Hence we have

$$\bigodot_{i=1}^{l} \alpha_i = \bigodot_{j=1}^{k} \beta_j, \quad 0 \leq l \leq n-1, 0 \leq k \leq m-1$$

by eliminating identical values at both sides. Note that k and l cannot be zero simultaneously.

If the polynomial time algorithm $\mathcal{A}$ can obtain two different range-based 2-3 trees $\mathcal{T}, \mathcal{T}'$ (meaning two different strings x and y) such that $\lambda = \lambda'$, then we can adopt $\mathcal{A}$ to find a solution to the balance problem by satisfying $\bigodot_{i=1}^{n-1} \alpha_i = \bigodot_{j=1}^{m-1} \beta_j$, where α_i, β_j are randomly selected from the group $\mathbb{Z}_p$, $\bigodot$ is the group operation and l, k cannot be zero simultaneously. In other words, the probability that we find a solution to the balance problem is the same as the probability that $\mathcal{A}$ find two different range-based 2-3 trees such that $\lambda = \lambda'$.

According to Theorem 3, $\Pr[E_2]$ is negligible.

Case 3: $F \neq F'$ but $\mathcal{T}' = \mathcal{T}, \lambda = \lambda'$. Let us consider the event E_3 as

$$\{\mathcal{T} = \mathcal{T}' | F \neq \ F' \wedge \ F \rightarrow \mathcal{T} \ \wedge \ F' \rightarrow \mathcal{T}'\}.$$

The identical range-based 2-3 tree means that the two files have the same length, and the hash values of each block are identical. However, since h is a collision-resistant hash function, $\Pr[E_3]$ is negligible.

In summary, $\Pr[E_1] + \Pr[E_2] + \Pr[E_3]$ is negligible, meaning that a dishonest client can break the fairness of the scheme with only negligible probability. This completes the proof. □

8. CONCLUSION

We presented FDPOR, a useful extension of the static POR. We also presented the first FDPOR scheme, which is proven secure in random oracle. Our scheme is based on two new building-blocks that might be of independent value.

Our future work includes: (i) realizing public verifiability so that a third party can verify the retrievability of data. (ii) utilizing erasure code rather than ECC so as to achieve possibly better performance.

Acknowledgements.

This work was supported in part by an AFOSR MURI grant and a State of Texas Emerging Technology Fund grant.

9. REFERENCES

[1] J. D. U. Alfred V.Aho, John E. Hopcroft. *The Design and Analysis of Computer Algorithms*. Addison Wesley, 1974.

[2] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song. Provable data possession at untrusted stores. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 598–609, 2007.

[3] G. Ateniese, R. Di Pietro, L. V. Mancini, and G. Tsudik. Scalable and efficient provable data possession. In *SecureComm '08: Proceedings of the 4th international conference on Security and privacy in communication netowrks*, pages 1–10, 2008.

[4] M. Bellare and O. Goldreich. On defining proofs of knowledge. In *CRYPTO*, pages 390–420, 1992.

[5] M. Bellare, O. Goldreich, and S. Goldwasser. Incremental cryptography: The case of hashing and signing. In *CRYPTO '94: Proceedings of the 14th Annual International Cryptology Conference on Advances in Cryptology*, pages 216–233, 1994.

[6] M. Bellare, O. Goldreich, and S. Goldwasser. Incremental cryptography and application to virus protection. In *STOC '95: Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*, pages 45–56, 1995.

[7] M. Bellare and D. Micciancio. A new paradigm for collision-free hashing: incrementality at reduced cost. In *EUROCRYPT'97: Proceedings of the 16th annual international conference on Theory and application of cryptographic techniques*, pages 163–192, 1997.

[8] 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.

[9] K. D. Bowers, A. Juels, and A. Oprea. Proofs of retrievability: theory and implementation. In *CCSW '09: Proceedings of the 2009 ACM workshop on Cloud computing security*, pages 43–54, 2009.

[10] H. Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *The Annals of Mathematical Statistics*, 23(4):493–507, 1952.

[11] Y. Dodis, S. Vadhan, and D. Wichs. Proofs of retrievability via hardness amplification. In *TCC '09: Proceedings of the 6th Theory of Cryptography Conference on Theory of Cryptography*, pages 109–127, 2009.

[12] C. Dwork, M. Naor, G. N. Rothblum, and V. Vaikuntanathan. How efficient can memory checking be? In *Theory of Cryptography (TCC'09)*, pages 503–520, 2009.

[13] C. Erway, A. Küpçü, C. Papamanthou, and R. Tamassia. Dynamic provable data possession. In *CCS '09: Proceedings of the 16th ACM conference on Computer and communications security*, pages 213–222, 2009.

[14] O. Goldreich. *The Foundations of Cryptography*, volume 1. Cambridge University Press, 2001.

[15] A. Juels and B. S. Kaliski, Jr. Pors: proofs of retrievability for large files. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 584–597, 2007.

[16] M. Lillibridge, S. Elnikety, A. Birrell, M. Burrows, and M. Isard. A cooperative internet backup scheme. In *ATEC '03: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 3–3, 2003.

[17] R. C. Merkle. Protocols for public key cryptosystems. *IEEE Symposium on Security and Privacy*, pages 122–134, 1980.

[18] D. Micciancio. Oblivious data structures: Applications to cryptography. In *In Proceedings of the 29th Annual ACM Symposium on the Theory of Computing*, pages 456–464, 1997.

[19] M. Naor and K. Nissim. Certificate revocation and certificate update. In *SSYM'98: Proceedings of the 7th conference on USENIX Security Symposium*, pages 17–17, 1998.

[20] H. Shacham and B. Waters. Compact proofs of retrievability. In *ASIACRYPT '08: Proceedings of the 14th International Conference on the Theory and Application of Cryptology and Information Security*, pages 90–107, 2008.