A Symmetric-Key Based Proofs of Retrievability Supporting Public Verification

Chaowen Guan[1], Kui Ren[1], Fangguo Zhang[1,2], Florian Kerschbaum[3], and Jia Yu [1,4]

[1]Department of Computer Science and Engineering, University at Buffalo

[2]School of Information Science and Technology, Sun Yat-sen University, China

[3]SAP, Karlsruhe, Germany

[4]College of Information Engineering, Qingdao University, China

Abstract

Proofs-of-Retrievability enables a client to store his data on a cloud server so that he executes an efficient auditing protocol to check that the server possesses all of his data in the future. During an audit, the server must maintain full knowledge of the client's data to pass, even though only a few blocks of the data need to be accessed. Since the first work by Juels and Kaliski, many PoR schemes have been proposed and some of them can support dynamic updates. However, all the existing works that achieve public verifiability are built upon traditional public-key cryptosystems which imposes a relatively high computational burden on low-power clients (e.g., mobile devices).

In this work we explore *indistinguishability obfuscation* for building a Proof-of-Retrievability scheme that provides public verification while the encryption is based on symmetric key primitives. The resulting scheme offers light-weight storing and proving at the expense of longer verification. This could be useful in applications where outsourcing files is usually done by low-power client and verifications can be done by well equipped machines (e.g., a third party server). We also show that the proposed scheme can support dynamic updates. For better assessing our proposed scheme, we give a performance analysis of our scheme and a comparison with several other existing schemes which demonstrates that our scheme achieves better performance on the data owner side and the server side. At last, we also give a brief discussion about several possible future directions in works on obfuscation.

1 Introduction

Nowadays, storage outsourcing (e.g., Google Drive, Dropbox, etc.) is becoming increasingly popular as one of the applications of cloud computing. It enables clients to access the outsourced data flexibly from any location. However, the storage provider (i.e., server) is not necessarily trusted. This situation gives rise to a need that a data owner (i.e., client) can efficiently verify that the server indeed stores the entire data. More precisely, a client can run an efficient *audit* protocol with the untrusted server where the server can pass the audit only if it maintains knowledge of the client's *entire* outsourced data. Formally, this implies two guarantees that the client wants from the server: *Authenticity* and *Retrievability*. Authenticity ensures that the client can verify the correctness of the data fetched from the server. On the other hand, Retrievability provides assurance that the client's data on the server is intact and no data loss has occurred. Apparently, the client should not need to download the entire data from server to verify the data's integrity, since this may be prohibitive in terms of bandwidth and time. Also, it is undesirable for the server to read all of the client's outsourced data during an audit protocol.

One method that achieves the above is called Proofs of Retrievability (PoR) which was initially defined and constructed by Juels and Kaliski [32]. Mainly, PoR schemes can be categorized into two classes: privately verifiable ones and publicly verifiable ones. Note that privately verifiable PoR systems normally only involve symmetric key primitives, which are cheap for the data owner in encrypting and uploading its files. However, in such systems the guarantees of the data's authenticity and retrievability largely depend on the data owners themselves due to the fact that they need to regularly perform verifications (e.g., auditing) in order to react as early as possible in case of a data loss. Nowadays, users create and upload data everywhere using low power devices, such as mobile phones. Obviously, such privately verifiable PoR system would inevitably impose expensive burdens on low power data owners in the long run. On the other hand, in this scenario with low power users, it is reasonable to have a well equipped server (trusted or semi-trusted) to perform auditing on behalf of data owner which requires publicly verifiable PoR systems. However, all of the existing PoR schemes that achieve public verifiability are constructed based on traditional public key cryptography which implies more complex and expensive computations compared to simple and symmetric key cryptographic primitives. That means a PoR scheme using public-key cryptographic primitives incurs relatively expensive overheads on low-capability clients. One might want to construct a public verifiable PoR scheme without relying on traditional public-key cryptographic primitives. One cryptographic primitive that can help to overcome this constraint is *indistinguishability obfuscation* ($i\mathcal{O}$) which achieves that obfuscations of any two distinct (equal-size) programs that implement the same functionality are computationally indistinguishable from each other. $i\mathcal{O}$ has become so important since the recent breakthrough result of Garg, Gentry, Halevi, Raykova, Sahai, and Waters in [24]. Garg et al. proposed the first candidate construction of an efficient indistinguishability obfuscator for general programs which are written as boolean circuits. Subsequently, Sahai and Waters [37] showed the power of $i\mathcal{O}$ as a cryptographic primitive: they used $i\mathcal{O}$ to construct public-key encryption, selectively-secure short signatures, denial encryption, and much more from pseudorandom functions. Most recently, by exploiting $i\mathcal{O}$, Ramchen et al. [36] built a fully secure signature scheme with fast signing and Boneh et al. [14] proposed a multiparty key exchange protocol, an efficient traitor tracing system and more.

Our work. In this paper, we explore this new primitive, $i\mathcal{O}$, for building PoR. In particular, we modify Shacham and Waters' privately verifiable PoR scheme [38] and apply $i\mathcal{O}$ to construct a publicly verifiable PoR scheme. Our results share a similar property with Ramchen et al.'s short signature scheme, that is, storing and proving are fast at the expense of longer public verification. Such "imbalance" could be useful in applications where outsourcing files is usually done by low-power client and verifications can be done by well equipped machines (a third party server). Our contributions are summarized as follows:

1. We explore building proof-of-retrievability systems from obfuscation. The resulting PoR scheme offers light-weight outsourcing, because it requires only symmetric key operations for the data owner to upload files to the cloud server. Likewise, the server also requires less workload during an auditing compared to existing publicly verifiable PoR schemes.

2. We show that the proposed PoR scheme can support dynamic updates by applying the Merkle hash tree technique. We first build a modified B+ tree over the file blocks and the corresponding block verification messages σ. Then we apply the Merkle hash tree to this tree for ensuring authenticity and freshness.

3. Note that the current $i\mathcal{O}$ construction candidate will incur a big amount of overhead for generating obfuscation, but it is only a one-time cost during the preprocessing stage of our system. Therefore its cost can be amortized over plenty of future auditing operations. Except for this one-time cost, we show that our proposed scheme achieves good performance on the data owner side and the cloud server side by analysis and comparisons with other recent existing PoR schemes.

4. Indistinguishability obfuscation indeed provides attractive and interesting features, but the current $i\mathcal{O}$ candidate construction offers impractical generation and evaluation. Given the fact that the development of $i\mathcal{O}$ is still in its nascent stages, we discuss several possible future directions in works on obfuscation in addition to those discussed in [24], including joint and outsourced generation of Indistinguishability Obfuscation, reusability and universality of Indistinguishability Obfuscation and obfuscation for specific functions.

1.1 Related Work

Proof of Retrievability and Provable Data Possession. The first PoR scheme was defined and constructed by Juels and Kaliski [32]. A closely related line of research is called Provable Data Possession (PDP) which was concurrently defined by Ateniese et al. [26]. The main difference between PoR and PDP is the notion of security that they achieve. Concretely, PoR provides stronger security guarantees than PDP does. A successful PoR audit guarantees that the server maintains knowledge of *all* of the client's outsourced data, while a successful PDP audit only ensures that the server is retaining *most* of the data. That means, in a PDP system a server that lost a small amount of data can still pass an audit with significant probability. Some PDP schemes [11] indeed provide full security. However, those schemes requires the server to read the client's entire data during an audit. If the data is large, this becomes totally impractical. A detailed comparison can be found in [33, 34]. Since the introduction of

PoR and PDP they have received much research attention. Roughly, the works on those two fields can be categorized into two classes: static data and dynamic data.

The first PoR work by Juels and Kaliski and the first PDP work by Ateniese et al. were considering static data. On the one hand, subsequent works [38, 19, 15, 6] for static data focused on the improvement of communication efficiency and exact security. On the other hand, the works of [5, 21, 41] showed how to construct dynamic PDP scheme supporting efficient updates. The techniques used in those works were closely related to the works on memory checking [12, 35, 20]. These techniques provide efficient authentications on remotely stored dynamic data and simultaneously enable verifications on the authenticity of the latest version of data.

Although many efficient PoR schemes have been proposed since the work of Juels and Kaliski (CCS 2007), only a few of them supports efficient dynamic update. The first work of dynamic PoR was proposed by Stefanov et al. [40] as part of a cloud-based file system called Iris. Recently, Cash et al. [17] showed how to construct dynamic PoR by using Oblivious RAM (ORAM), while Shi et al. [39] proposed a light-weight dynamic PoR construction that achieves comparable bandwidth and client side computation with a standard Merkle hash tree.

Observe that in publicly verifiable PoR systems, an external verifier (called auditor) is able to perform an auditing protocol with the cloud server on behalf of the data owner. However, public PoR systems do not provide any security guarantees when the user and/or the external verifier are dishonest. To address this problem Armknecht et al. recently introduced the notion of *outsourced proofs of retrievability* (OPoR) [4]. In particular, OPoR protects against the collusion of any two parties among the malicious auditor, malicious users and the malicious cloud server. Armknecht et al. proposed a concrete OPoR scheme, named Fortress, which is mainly built upon the private PoR scheme in [38]. In order to be secure in the OPoR security model, Fortress also employs a mechanism that enables the user and the auditor to extract common pseudorandom bits using a time-dependent source without any interaction.

Indistinguishability Obfuscation. Program obfuscation aims to make computer programs "unintelligible" while preserving their functionality. The formal study of obfuscation was started by Barak et al. [8, 9] in 2001. In their work, they first suggested a quite intuitive notion called *virtual black-box* obfuscation, for which they also showed impossibility. Motivated by this impossibility, they proposed another important notion of obfuscation called *indistinguishability obfuscation* ($i\mathcal{O}$), which was further studied by Goldwasser and Rothblum in [28]. Indistinguishability obfuscation asks that obfuscations of any two distinct (equal-size) programs that implement the same functionalities are computationally indistinguishable from each other. A recent breakthrough result by Garg, Genry, Halevi, Raykova, Sahai, and Waters [24] presented the first candidate construction of an efficient indistinguishability obfuscator for general programs, which are written as boolean circuits. The proposed construction was build on the multilinear map candidates [22, 18].

The works of Garg et al. [24] also showed how to apply indistinguishability obfuscation to the construction of functional encryption schemes for general circuits. In subsequent work, Sahai and Waters [37] formally investigated what can be built from indistinguishability obfuscation and showed the power of indistinguishability obfuscation as a cryptographic primitive. Since then, many new applications of general-purpose obfuscation have been explored [1, 29, 10, 25, 27, 7, 16, 31, 23]. Most recently, the works of Boneh et al. [14] and Ramchen et al. [36] re-explore the constructions of some existing cryptographic primitives through the lens of obfuscation, including broadcast encryption, traitor tracing and signing. Those proposed constructions indeed obtain some attractive features, although current obfuscation candidates incur prohibitive overheads. Precisely, Boneh et al.'s broadcast encryption achieves that ciphertext size is independent of the number of users, and their traitor tracing system achieves full collusion resistance with short ciphertexts, secret keys and public keys. Additionally, a generalization of their traitor tracing system can lead to a private broadcast encryption scheme with optimal size ciphertexts. On the other hand, Ramchen et al's work [36] proposed an *imbalanced* signing algorithm, which is ideally significantly faster than comparable signatures that are not built upon obfuscation. Here imbalanced means the signing is fast at the expense of longer verification. Observe that the work by Boneh et al. [14] also shared this imbalance feature in a different way, where their schemes achieved short ciphertexts or short keys instead. This imbalance (i.e., asymmetrical speed-up) feature is also the goal of our work. We are using symmetric key cryptographic primitives to construct PoR schemes that supports public verification at the expense of longer verification.

2 Preliminaries

In this section we define erasure codes, proof-of-retrievability, indistinguishability obfuscation, and variants of pseudorandom functions (PRFs) that we will make use of. All the variants of PRFs that we consider will be constructed from one-way functions.

2.1 Erasure Codes

Let Σ be a finite alphabet. We say that $(\mathsf{encode}, \mathsf{decode})$ is an efficient $(m, k, d)_\Sigma$ erasure code over an alphabet Σ if the original message can always be recovered from a corrupted codeword with at most $d-1$ erasures. A code is maximum distance separable (MDS), if $k + d = m + 1$.

2.2 Proofs of Retrievability

Below, we give the definition of publicly verifiable PoR scheme in a way similar to that in [38]. A proof of retrievability scheme defines four algorithms, KeyGen, Store, $\mathcal{P}$ and $\mathcal{V}$, which are specified as follows:

$(pk, sk) \leftarrow$**KeyGen**(1^λ). On input the security parameter λ, this randomized algorithm generates a public-private keypair (pk, sk).

$(M^*, t) \leftarrow$**Store**(sk, M). On input a secret key sk and a file $M \in \{0,1\}^*$, this algorithm processes M to produce M^*, which will be stored on the server, and a tag t. The tag t contains information associated with the file being stored.

$(0, 1) \leftarrow$ **Audit**(**Prove**, **Verify**). The randomized proving and verifying algorithms together define an Audit-protocol for proving file retrievability. During protocol execution, both algorithms take as input the public key pk and the file tag t output by Store. **Prove** algorithm also takes as input the processed file description M^* that is output by Store, and **Verify** algorithm takes as input public verification key VK. At the end of the protocol, **Verify** outputs 0 or 1, with 1 indicating that the file is being stored on the server. We denote a run of two parties executing such protocol as:

$$\{0,1\} \leftarrow (\textbf{Verify}(pk, VK, t) \rightleftharpoons \textbf{Prove}(pk, t, M^*)).$$

Correctness. For all keypairs (pk, sk) output by KeyGen, for all files $M \in \{0,1\}^*$, and for all (M^*, t) output by $\mathsf{Store}(sk, M)$, the verification algorithm accepts when interacting with the valid prover:

$$(\textbf{Verify}(pk, VK, t) \rightleftharpoons \textbf{Prove}(pk, t, M^*)) = 1.$$

2.3 Obfuscation Preliminaries

We recall the definition of indistinguishability obfuscation from [24, 37].

Definition 1. Indistinguishability Obfuscation ($i\mathcal{O}$). A uniform PPT machine $i\mathcal{O}$ is called an *indistinguishability obfuscator* for a circuit class $\{\mathcal{C}_\lambda\}_{\lambda \in \mathbb{N}}$ if the following conditions are satisfied:

- For all security parameters $\lambda \in \mathbb{N}$, for all $C \in \mathcal{C}_\lambda$, for all inputs x, we have that

$$\mathsf{Pr}[C'(x) = C(x) : C' \leftarrow i\mathcal{O}(\lambda, C)] = 1.$$

- For any (not necessarily uniform) PPT distinguisher $(Samp, D)$, there exists a negligible function $negl(\cdot)$ such that the following holds: if for all security parameters $\lambda \in \mathbb{N}$

$$\Pr[\forall x, C_0(x) = C_1(x) : (C_0; C_1; \tau) \leftarrow Samp(1^\lambda)] > 1 - negl(\lambda),$$

then we have

$$|\Pr[D(\tau, i\mathcal{O}(\lambda, C_0)) = 1 : (C_0; C_1; \tau) \leftarrow Samp(1^\lambda)] - \Pr[D(\tau, i\mathcal{O}(\lambda, C_1)) = 1 : (C_0; C_1; \tau) \leftarrow Samp(1^\lambda)]| \leq negl(\lambda).$$

2.4 Puncturable PRFs

A pseudorandom function (PRF) is a function $F : \mathcal{K} \times \mathcal{M} \to \mathcal{Y}$ such that the function $F(K, \cdot)$ is indistinguishable from random when $K \xleftarrow{\$} \mathcal{K}$.

A constrained PRF [13] is a PRF $F(K, \cdot)$ that is able to evaluate at certain portions of the input space and nowhere else. A *puncturable* PRF [13, 37] is a type of constrained PRF that enables the evaluation at all bit strings of a certain length, except for any polynomial-size set of inputs.
PRF $F(K, \cdot)$ is defined with two PPT algorithms $(\mathsf{Eval}_F, \mathsf{Puncture}_F)$ such that the following two properties hold

- **Functionality preserved under puncturing.** For every PPT adversary $\mathcal{A}$ which on input 1^λ outputs a set $S \subseteq \{0,1\}^n$, for all $x \in \{0,1\}^n \backslash S$, we have

$$\Pr[\mathsf{Eval}_F(K\{S\}, x) = F(K, x) : K \xleftarrow{\$} \mathcal{K}, K\{S\} \leftarrow \mathsf{Puncture}_F(K, S)] = 1$$

- **Pseudorandom at punctured points.** For every pair of PPT adversary $(\mathcal{A}_1, \mathcal{A}_2)$ and polynomial $m(\lambda)$ such that $\mathcal{A}_1(1^\lambda)$ outputs a set $S \subseteq \{0,1\}^n$ of cardinality $m(\lambda)$ and a state σ it holds that

$$|\Pr[\mathcal{A}_2(\sigma, K\{S\}, F(K, S)) = 1 : (S, \sigma) \leftarrow \mathcal{A}_1(1^\lambda), K\{S\} \leftarrow \mathsf{Puncture}_F(K, S)] -$$
$$\Pr[\mathcal{A}_2(\sigma, K\{S\}, Y) = 1 : (S, \sigma) \leftarrow \mathcal{A}_1(1^\lambda), K\{S\} \leftarrow \mathsf{Puncture}_F(K, S), Y \xleftarrow{\$} \{0,1\}^{m \cdot |S|}]| \leq negl(\lambda)$$

3 Security Definitions

The security definitions of *Authenticity* and *Retrievability* in [17, 39] are essentially equivalent to Shacham and Waters' security definition of *Soundness* in [38]. Note that the security definitions in [17, 39] are for dynamic PoR systems, while the security definition in [38] considers only static PoR systems. The only difference between a static PoR scheme and a dynamic PoR scheme is that the latter one supports secure dynamic updates, including modification, deletion and insertion. This affects the access to oracles in the security game. Roughly speaking, in a static PoR scheme, the adversary is provided with Store and Audit oracle access, while the attacker can get access to Read, Write and Audit oracles.

In this section, we present the security definitions for a static PoR system and a dynamic PoR system, respectively, in the same way as [17, 39].

3.1 Security Definitions on static PoR

Authenticity. Authenticity requires that the client can always detect if any message sent by the server deviates from honest behavior. More precisely, consider the following game between a challenger $\mathcal{C}$, a malicious server $\widetilde{\mathcal{S}}$ and an honest server $\mathcal{S}$ for the adaptive version of authenticity:

- The challenger initializes the environment and provides $\widetilde{\mathcal{S}}$ with public parameters.
- The malicious sever $\widetilde{\mathcal{S}}$ specifies a valid protocol sequence $P = (op_1, op_2, \cdots, op_{\text{poly}(\lambda)})$ of polynomial size in the security parameter λ. The specified operations op_t can be either Store or Audit. $\mathcal{C}$ executes the protocol with both $\widetilde{\mathcal{S}}$ and an honest server $\mathcal{S}$.

If at execution of any op_j, the message sent by $\widetilde{\mathcal{S}}$ differs from that of the honest server $\mathcal{S}$ and $\mathcal{C}$ does not output reject, the adversary $\widetilde{\mathcal{S}}$ wins and the game results in 1, else 0.

Definition 2. A static PoR scheme is said to satisfy adaptive $\mathsf{Authenticity}$, if any polynomial-time adversary $\widetilde{\mathcal{S}}$ wins the above security game with probability no more than $\mathsf{negl}(\lambda)$.

Retrievability. Retrievability guarantees that whenever a malicious server can pass the audit test with non-negligible probability, the server must know the entire content of $\mathcal{M}$; and moreover, $\mathcal{M}$ can be recovered by repeatedly running the Audit-protocol between the challenger $\mathcal{C}$ and the server $\widetilde{\mathcal{S}}$. More precisely, consider the following security game:

- The challenger initializes the environment and provides $\widetilde{\mathcal{S}}$ with public parameters.
- The malicious server $\widetilde{\mathcal{S}}$ specifies a protocol sequence $P = (op_1, op_2, \cdots, op_{\text{poly}(\lambda)})$ of polynomial size in terms of the security parameter λ. The specified operations op_t can be either Store or Audit. Let $\mathcal{M}$ be the correct content value.

- The challenger $\mathcal{C}$ sequentially executes the respective protocols with $\widetilde{\mathcal{S}}$. At the end of executing P, let $st_{\mathcal{C}}$ and $st_{\widetilde{\mathcal{S}}}$ be the final configurations (states) of the challenger and the malicious server, respectively.
- The challenger now gets black-box rewinding access to the malicious server in its final configuration $st_{\widetilde{\mathcal{S}}}$. Starting from the configurations $(st_{\mathcal{C}}, st_{\widetilde{\mathcal{S}}})$, the challenger runs the Audit-protocol repeatedly for a polynomial number of times with the server $\widetilde{\mathcal{S}}$ and attempts to extract out the content value as $\mathcal{M}'$.

If the malicious server $\widetilde{\mathcal{S}}$ passes the Audit-protocol with non-negligible probability and the extracted content value $\mathcal{M}' \neq \mathcal{M}$, then this game outputs 1, else 0.

Definition 3. A static PoR scheme is said to satisfy Retrievability, if there exists an efficient extractor $\mathcal{E}$ such that for any polynomial-time $\widetilde{\mathcal{S}}$, if $\widetilde{\mathcal{S}}$ passes the Audit-protocol with non-negligible probability, and then after executing the Audit-protocol with $\widetilde{\mathcal{S}}$ for a polynomial number of times, the extractor $\mathcal{E}$ outputs content value $\mathcal{M}' \neq \mathcal{M}$ only with negligible probability.

The above says that the extractor $\mathcal{E}$ will be able to extract out the correct content value $\mathcal{M}' = \mathcal{M}$ if the malicious server $\widetilde{\mathcal{S}}$ can maintain a non-negligible probability of passing the Audit-protocol. This means the server must retain full knowledge of $\mathcal{M}$.

3.2 Security Definitions on Dynamic PoR

The security definitions for dynamic PoR systems are the same as those for static PoR systems, except that the oracles which the malicious server $\widetilde{\mathcal{S}}$ has access to are including Read, Write and Audit. Similarly, we define Authenticity and Retrievability for dynamic PoR systems separately.
Authenticity. This security game is the same as that for static PoR systems, except that the malicious server $\widetilde{\mathcal{S}}$ can get access to Read, Write and Audit oracles. This means that the specified operations op_t by $\widetilde{\mathcal{S}}$ in the protocol sequence $P = (op_1, op_2, \cdots, op_{\text{poly}(\lambda)})$ can be either Read, Write or Audit.
The winning condition remains unchanged. That is, if at execution of any op_j, the message sent by $\widetilde{\mathcal{S}}$ differs from that of the honest server $\mathcal{S}$ and $\mathcal{C}$ does not output reject, the adversary $\widetilde{\mathcal{S}}$ wins and the game results in 1, else 0.

Definition 4. A dynamic PoR scheme is said to satisfy adaptive Authenticity, if any polynomial-time adversary $\widetilde{\mathcal{S}}$ wins the above security game with probability no more than $\mathsf{negl}(\lambda)$.

Retrievability. Again, this security game is the same as that for static PoR systems, except that the malicious server $\widetilde{\mathcal{S}}$ can get access to Read, Write and Audit oracles. This changes the kind of operations that the malicious server $\widetilde{\mathcal{S}}$ can specify in the game.
The winning condition is the same. If the malicious server $\widetilde{\mathcal{S}}$ passes the Audit-protocol with non-negligible probability and the extracted content value $\mathcal{M}' \neq \mathcal{M}$, then this game outputs 1, else 0.

Definition 5. A dynamic PoR scheme is said to satisfy Retrievability, if there exists an efficient extractor $\mathcal{E}$ such that for any polynomial-time $\widetilde{\mathcal{S}}$, if $\widetilde{\mathcal{S}}$ passes the Audit-protocol with non-negligible probability, and then after executing the Audit-protocol with $\widetilde{\mathcal{S}}$ for a polynomial number of times, the extractor $\mathcal{E}$ outputs content value $\mathcal{M}' \neq \mathcal{M}$ only with negligible probability.

4 Constructions

In this section we first give the construction of a static publicly verifiable PoR system. Then we discuss how to extend this static PoR scheme to support efficient dynamic updates.

Before presenting our proposed constructions, we analyze a trivial construction of a publicly verifiable PoR scheme using $i\mathcal{O}$. Let n be the number of file blocks, λ_1 be the size of a file block (here assume every file block is equally large), λ_2 be the size of a block tag σ and I be the challenge index set requested by the verifier. Since $i\mathcal{O}$ can hide secret information, which is embedded into the obfuscated program, from the users, one might construct a scheme as: 1) set the tag for a file block m_i as the output of a PRF $F(k, m_i)$ with secret key k; 2) embed key k into the public verification program and obfuscate it; 3) this verification program simply checks the tags for the challenged file blocks to see if they are valid outputs of the PRF. Observe that this verification program takes as inputs a challenge index set, the challenged file blocks and the corresponding file tags. Therefore, the circuit for this verification program will be of size $O(poly(|I| \cdot \log n + |I| \cdot \lambda_1 + |I| \cdot \lambda_2))$, where $|I|$ is the size of index set I and $poly(x)$ represents a polynomial in terms of x. Clearly, this method also costs much a lot of bandwidth due to the fact that it doesnot provide an aggregated proof.

While in our construction we modify the privately-verifiable PoR scheme in [38], for consistency with the above analysis, file blocks are not further divided into sectors. Then the verification program will take as input a challenge index set I, an aggregation of the challenged file blocks μ and an aggregated file tags σ'. Consequently the circuit for the verification program will have size $O(poly(|I| \cdot \log n + \lambda_1 + \lambda_2))$, which is much smaller than that in the above trivial construction. Clearly, the trivial construction will lead to a significantly larger obfuscation of the verification program.

Similarly, we analyze the circuit's size when a file block is further split into s sectors, as the scheme in [38] did. Let the size of a sector in a file block be λ_3. The circuit size in the trivial construction will remain unchanged, $O(poly(|I| \cdot \log n + |I| \cdot \lambda_1 + |I| \cdot \lambda_2))$. While the circuit in our construction will has size $O(poly(|I| \cdot \log n + s \cdot \lambda_3 + \lambda_3)) \approx O(poly(|I| \cdot \log n + \lambda_1 + \lambda_3))$, which is still much smaller than that in the trivial construction. As we can see, exploiting $i\mathcal{O}$ is not trivial although it is a powerful cryptographic primitive.

4.1 Static publicly verifiable PoR scheme

We modify Shacham and Waters' privately verifiable PoR scheme in [38] and combine the modified PoR scheme with $i\mathcal{O}$ to give a publicly verifiable PoR scheme.

Recall that in Shacham and Waters' scheme [38], a file F will be processed using erasure encoding and then divided into n blocks. Also note that each block is split into s sectors. This allows for a tradeoff between storage overhead and communication overhead, which has already been discussed in [38].

Before presenting the construction of the proposed static PoR scheme, we give a brief discussion on how we apply indistinguishability obfuscation to the PoR scheme in [38]. For doing that, we need to utilize a key technique, named *punctured programs*. At a very high-level, the idea of this technique is to modify a program (which is to be obfuscated) by surgically removing a key element of the program, without which the adversary cannot win the security game it must play, but in a way that does not change the functionality of the program. Note that, in Shacham and Waters' PoR scheme, for each file block, σ_i is set as $f_{prf}(i) + \sum_{j=1}^{s} \alpha_j m_{ij}$, where the secret key k_{prf} for PRF f is specific for one certain file M. That means for different files, it uses different PRF key k_{prf}'s. As to make it a *punctured* PRF that we want in the obfuscated program, we eliminate this binding between PRF key k_{prf} and file M, and the same PRF key k_{prf} will be used in storing many different files. Thus, the PRF key k_{prf} will be randomly chosen in client KeyGen step, not in Store step. The security will be maintained after this modification, due to the fact that it still provides σ_i with randomness without adversary getting the PRF key. We will see this in the security proof.

The second main change is related to the construction of a file tag t. Note that, in Shacham and Waters' scheme, $t = n\|c\|\mathsf{MAC}_{k_{mac}}(n\|c)$, where $c = \mathsf{Enc}_{k_{enc}}(k_{prf}\|\alpha_1\| \cdots \|\alpha_s)$. In our proposed scheme, the randomly selected elements $\alpha_1, \alpha_2, \cdots, \alpha_s$ will be removed from the file tag t. Instead, we use another PRF key $f_{prf'}$ to generate s pseudorandom numbers, which will reduce the communication cost by $(s \cdot \lceil \log p \rceil)$, where $\log p$ means each element α_i is from group $\mathbb{Z}_p$. As a consequence of these two changes, we no longer need to build the symmetric encryption component c here, which means k_{enc} becomes useless, and also σ_i will be made as $f_{prf}(i) + \sum_{j=1}^{s} f_{prf'}(j) \cdot m_{ij}$. The detailed construction is specified as follows:

Let $F_1(k_1, \cdot)$ be a puncturable PRF mapping $\lceil \log N \rceil$-bit inputs to $\lceil \log \mathbb{Z}_p \rceil$. Here N is a bound on the number of blocks in a file. Let $F_2(k_2, \cdot)$ be a puncturable PRF mapping $\lceil \log s \rceil$-bit inputs to $\lceil \log \mathbb{Z}_p \rceil$. Let $\mathsf{SSig}_{ssk}(x)$ be the algorithm generating a signature on x.

KeyGen(). Randomly choose two PRF key $k_1 \in \mathcal{K}_1$, $k_2 \in \mathcal{K}_2$ and a random signing keypair $(svk, ssk) \overset{R}{\leftarrow} \mathsf{SK}_g$. Set the secret key $sk = (k_1, k_2, ssk)$. Let the verification key VK be svk along with an indistinguishability obfuscation of the program Check defined as below.

Store(sk, M)**.** Given inputs the file M and secret key $sk = (k_1, k_2, ssk)$, proceed as follows:

1. apply the erasure code to M to obtain M';
2. split M' into n blocks, and each block into s sectors to get $\{m_{ij}\}$ for $1 \le i \le n, 1 \le j \le s$;
3. set the file tag $t = n\|\mathsf{SSig}_{ssk}(n)$
4. for each i, $1 \le i \le n$, compute $\sigma_i = F_1(k_1, i) + \sum_{j=1}^{s} F_2(k_2, j) \cdot m_{ij}$;

5. set as the outputs the processed file $M' = \{m_{ij}\}$, $1 \leq i \leq n, 1 \leq j \leq s$, the corresponding file tag t and $\{\sigma_i\}, 1 \leq i \leq n$.

Verify(pk, VK, t)**.** Given inputs the file tag t, first parse $t = n \| \mathsf{SSig}_{ssk}(n)$ and use svk to verify the signature on t; if the signature is invalid, reject and halt. Otherwise, pick a random l-element subset I from $[1, n]$, and for each $i \in I$, pick a random element $v_i \in \mathbb{Z}_p$. Let Q be the set $\{(i, v_i)\}$, and sent Q to the prover.

Parse the prover's response to obtain $\mu_1, \cdots, \mu_s, \sigma \in \mathbb{Z}_p^{s+1}$. If parsing fails, reject and halt. Otherwise, output $\mathrm{VK}(Q = \{(i, v_i)\}_{i \in I}, \mu_1, \cdots, \mu_s, \sigma)$.

```
Check:
Inputs: $Q = \{(i, v_i)\}_{i \in I}, \mu_1, \cdots, \mu_s, \sigma$
Constants: PRF keys $k_1, k_2$
    if $\sigma = \sum_{(i,v_i) \in Q} v_i \cdot F_1(k_1, i) + \sum_{j=1}^{s} F_2(k_2, j) \cdot \mu_j$ then output 1
    else output $\perp$
```

Prove(pk, t, M')**.** Given inputs the processed file M', $\{\sigma_i\}, 1 \leq i \leq n$ and an l-element set Q sent by the verifier, first parse $M' = \{m_{ij}\}, 1 \leq i \leq n, 1 \leq j \leq s$ and $Q = \{(i, v_i)\}$. Then compute

$$\mu_j = \sum_{(i,v_i) \in Q} v_i m_{ij} \text{ for } 1 \leq j \leq s, \quad \text{and} \quad \sigma = \sum_{(i,v_i)} v_i \sigma_i,$$

and send to the prove in response the values $\mu_1, \cdots, \mu_s$ and σ.

4.2 PoR scheme Supporting Efficient Dynamic Updates

A PoR scheme supporting dynamic updates means that it enables modification, deletion and insertion over the stored files. Note that, in the static PoR scheme, each σ_i associated with ${m_{ij}}_{1 \leq j \leq s}$ is also bound to a file block index i. If an update is executed in this static PoR scheme, it requires to change every σ_i corresponding to the involved file blocks, and the cost could probably be expensive. Let's say the client needs to insert a file block F_i into position i. We can see that this insertion manipulation requires to update the indices in σ_j's for all $i \leq j \leq n$. On average, a single insertion incurs updates on $n/2$ σ_j's.

In order to offer efficient insertion, we need to disentangle σ_i from index i. Concretely, $F_1(k_1, \cdot)$ should be erased in the computing of σ_i, which leads to a modified $\sigma'_i = \sum_{j=1}^{s} F_2(k_2, j) \cdot m_{ij}$. However, this would make the scheme insecure, because the malicious server can always forge, e.g., $\sigma'_i/2 = \sum_{j=1}^{s} F_2(k_2, j) \cdot (m_{ij}/2)$ for file block $\{m_{ij}/2\}_{1 \leq j \leq s}$ with this σ'_i.

Instead, we build σ_i as $F_1(k_1, r_i) + \sum_{j=1}^{s} F_2(k_2, j) \cdot m_{ij}$, where r_i is a random element from $\mathbb{Z}_p$. Clearly, we can't maintain the order of the stored file blocks without associating each σ_i with index i. To provide the guarantee that every up-to-date file block is in the designated position, we use a modified B+ tree data structure with standard Merkle hash tree technique.

Observe that, unlike Shacham and Waters' scheme where the file is split into n blocks after being erasure encoded, the construction above assumes that each file block is encoded 'locally'. (Cash et al.'s work [17] also started with this point.) That is, instead of using an erasure code that takes the entire file as input, we use a code that works on small blocks. More precisely, the client divides the file M into n blocks, i.e., $M = (m_1, m_2, \cdots, m_n)$, and then encodes each file block m_i individually into a corresponding codeword block $c_i = \mathsf{encode}(m_i)$. Next, the client performs the following PoR scheme to create σ_i for each c_i. Auditing works as before: The verifier randomly selects l indices from $[1, n]$ and l random values, and then challenges the server to respond with a proof that is computed with those l random values and corresponding codewords specified by the l indices. Note that, in this construction, each codeword c_i will be further divided into s sectors, $(c_{i1}, c_{i2}, \cdots, c_{is})$ during the creation of σ_i. One may wonder whether we should divide the file M into blocks and then into sectors before being encoded, that is, $\mathsf{encode}(m_{ij})$. A more detailed discussion about this and analysis of how to better define block size can be found in the appendices in [38, 17].

Let $F_1(k_1, \cdot)$ be a puncturable PRF mapping $\lceil \log N \rceil$-bit inputs to $\lceil \log \mathbb{Z}_p \rceil$. Here N is a bound on the number of blocks in a file. Let $F_2(k_2, \cdot)$ be a puncturable PRF mapping $\lceil \log s \rceil$-bit inputs to $\lceil \log \mathbb{Z}_p \rceil$. Let $\mathsf{Enc}_k/\mathsf{Dec}_k$ be a symmetric encryption/decryption algorithm, and $\mathsf{SSig}_{ssk}(x)$ be the algorithm generating a signature on x.

KeyGen(). Randomly choose puncturable PRF keys $k_1 \in \mathcal{K}_1$ $k_2 \in \mathcal{K}_2$, a symmetric encryption key $k_{enc} \in \mathcal{K}_{enc}$ and a random signing keypair $(svk, ssk) \xleftarrow{R} \mathsf{SK}_g$. Set the secret key $sk = (k_1, k_2, k_{enc}, ssk)$. Let the verification key VK be svk along with an indistinguishability obfuscation of the program CheckU defined as below.

Store(sk, M)**.** Given inputs the file M and secret key $sk = (k_1, k_2, k_{enc}, ssk)$, proceed as follows:

1. apply the erasure code to M to obtain M';
2. split M' into n blocks, and each block into s sectors to get $\{m_{ij}\}$ for $1 \leq i \leq n, 1 \leq j \leq s$;
3. for each i, $1 \leq i \leq n$, choose a random element $r_i \in \mathbb{Z}_p$ and compute $\sigma_i = F_1(k_1, r_i) + \sum_{j=1}^{s} F_2(k_2, j) \cdot m_{ij}$;
4. set $c = \mathsf{Enc}_{k_{enc}}(r_1 \| \cdots \| r_n)$ and the file tag $t = n \| c \| \mathsf{SSig}_{ssk}(n \| c)$;
5. set as the outputs the processed file $M' = \{m_{ij}\}$, $1 \leq i \leq n, 1 \leq j \leq s$, the corresponding file tag t and $\{\sigma_i\}, 1 \leq i \leq n$.

Verify(pk, VK, t)**.** Given inputs the file tag t, first parse $t = n \| c \| \mathsf{SSig}_{ssk}(n \| c)$ and use svk to verify the signature on t; if the signature is invalid, reject and halt. Otherwise, pick a random l-element subset I from $[1, n]$, and for each$i \in I$, pick a random element $v_i \in \mathbb{Z}_p$. Let Q be the set $\{(i, v_i)\}$, and sent Q to the prover.

Parse the prover's response to obtain $\mu_1, \cdots, \mu_s, \sigma \in \mathbb{Z}_p^{s+1}$. If parsing fails, reject and halt. Otherwise, output VK$(Q = \{(i, v_i)\}_{i \in I}, \mu_1, \cdots, \mu_s, \sigma, t)$.

```
CheckU:
Inputs: $Q = \{(i, v_i)\}_{i \in I}, \mu_1, \cdots, \mu_s, \sigma, t$
Constants: PRF keys $k_1$, $k_2$, symmetric encryption key $k_{enc}$
    $n \| c \| \mathsf{SSig}_{ssk}(n \| c) \leftarrow t$
    $r_1, \cdots, r_n \leftarrow Dec_{k_{enc}}(c)$
    if $\sigma = \sum_{(i,v_i) \in Q} v_i \cdot F_1(k_1, r_i) + \sum_{j=1}^{s} F_2(k_2, j) \cdot \mu_j$ then output 1
    else output $\perp$
```

Prove(pk, t, M')**.** Given inputs the processed file M', $\{\sigma_i\}, 1 \leq i \leq n$ and an l-element set Q sent by the verifier, first parse $M' = \{m_{ij}\}, 1 \leq i \leq n, 1 \leq j \leq s$ and $Q = \{(i, v_i)\}$. Then compute

$$\mu_j = \sum_{(i,v_i) \in Q} v_i m_{ij} \text{ for } 1 \leq j \leq s, \quad \text{and} \quad \sigma = \sum_{(i,v_i)} v_i \sigma_i,$$

and send to the prove in response the values $\mu_1, \cdots, \mu_s$ and σ.

Modified B+ Merkle tree. In our construction, we organize the data files using a modified B+ tree, and combine it with a standard Merkle Hash tree to provides guarantees of freshness and authenticity. In this modified B+ tree, each node has at most three entries. Each entry in leaf node is data file's σ and is linked to its corresponding data file in the additional bottom level. The internal nodes will no longer have index information. Before presenting the tree's construction, we first define some notations. We denote an entry's corresponding computed σ by $label(\cdot)$, the rank of an entry (i.e., the number of file blocks that can be reached from this entry) by $rank(\cdot)$, descendants of an entry by $child(\cdot)$, left/right sibling of an entry by $len(\cdot)/ren(\cdot)$.

- entry w in leaf node: $label(w) = \sigma$, $len(w)$ (if w is the leftmost entry, $len(w) = 0$) and $ren(w)$ ((if w is the rightmost entry, $ren(w) = 0$);
- entry v in internal node and root node: $rank(v)$, $child(v)$ $len(v)$ and $ren(v)$, where $len(v)$ and $ren(v)$ conform to the rules above.

An example is illustrated in Figure 1a. Following the definitions above, entry v_1 in the root node contains: (1) $rank(v_1) = 3$, because w_1, w_2 and w_3 can be reached from v_1; (2) $child(v_1) = w_1 \| w_2 \| w_3$; (3) $len(v_1) = 0$; (4) $ren(v_1) = v_2$. Entry w_2 in the leaf node contains: (1) $label(w_2) = \sigma_2$; (2) $len(w_2) = w_1$; (3) $ren(w_2) = w_3$. Note that the arrows connecting the entries in leaf nodes with F's means that each entry is associated with its corresponding file block. Precisely, e.g., entry w_1 is associated with the first data block F_1 and $label(w_1) = \sigma_1$.

To search for a σ and its corresponding data file on this modified B+ tree, we need two other values of each entry $low(\cdot)$ and $high(\cdot)$. $low(\cdot)$ gives the lowest-position data block that can be reached from an entry, and $high(\cdot)$ defines the highest-position data block that can be reached from an entry. Observe that these two values need not be stored for every entry in the tree. We can compute them on the fly using the ranks. For the current entry r, assume we know $low(r)$ and $high(r)$. Let $child(r) = v_1 \| v_2 \| v_3$. Then $low(v_i)$'s and $high(v_i)$'s can be computed with entry's $rank$ value in the following way:

$$\begin{aligned}
low(v_1) &= low(r), \\
high(v_1) &= low(v_1) + rank(v_1) - 1, \\
low(v_2) &= high(v_1) + 1, \\
high(v_2) &= low(v_2) + rank(v_2) - 1, \\
low(v_3) &= high(v_2) + 1, \\
high(v_3) &= high(r).
\end{aligned}$$

Using the ranks stored on the entries, we can reach the $i-th$ data block (i.e., $i-th$ entry) in the leaf nodes. The search starts with entry v_1 in root node (internal node). Clearly, for the start entry of the tree, we have $low(v_1) = 1$. On each entry v during the search, we check whether $i \in [low(v), high(v)]$, and we proceed the search along the pointer from v to its children; otherwise, check the next entry on v's right side. We continue until we reach the *i-th* data block. For instance, say we want to read the 6-*th* data block in Figure 1a. We start with root node's entry v_1, and then the search proceeds as follows:

1. compute $high(v_1) = low(v_1) + rank(v_1) - 1 = 3$;
2. $i = 6 \notin [low(v_1), high(v_1)]$, then check the next entry, v_2;
3. compute $low(v_2) = high(v_1) + 1 = 4$ and $high(v_2) = low(v_2) + rank(v_2) - 1 = 6$;
4. $i \in [low(v_2), high(v_2)]$, then follow the pointer leading to v_2's children;
5. get $child(v_2) = w_4 \| w_5 \| w_6$;
6. now in leaf node, check each entry from left to right, and find w_6 be the entry connecting to the wanted data block.

Now it is only left to define the Merkle hash tree on this modified B+ tree. Note that in our modified B+ tree, each node have at most 3 entries. Let upper case letter denote node and lower case one denote entry. Below we define the hashing value for node and entry, respectively.

For each entry, the hashing value is computed as follows:

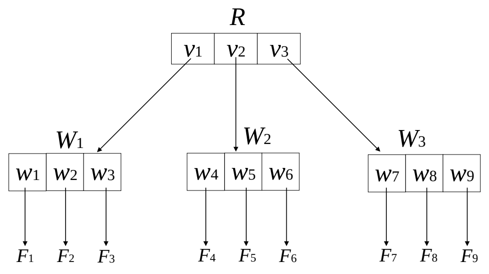
- **Case 0**: w is an entry in a leaf node
$$f(w) = h(label(w)) = h(\sigma),$$
- **Case 1**: v is an entry in an internal node and it's descendent is node V'
$$f(v) = h(rank(v) \| f(V')).$$

For each node (internal node or leaf node) consisting of entries v_1, v_2, v_3, we define

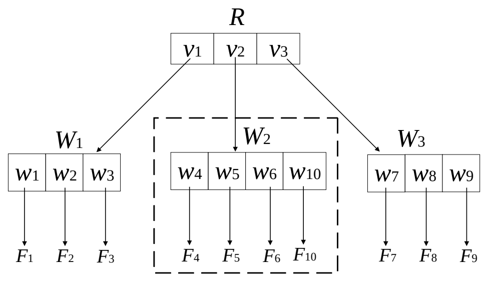
$$f(V) = h(f(v_1) \| f(v_2) \| f(v_3)).$$

For instance, in Figure 1a's example, the hashing value for the root node is $f(R) = h(f(v_1) \| f(v_2) \| f(v_3))$, where $f(v_i) = h(rank(v_i) \| f(W_i))$ and $f(W_i) = h(f(w_{(i-1)*3+1}) \| f(w_{(i-1)*3+2}) \| f(w_{(i-1)*3+3}))$.

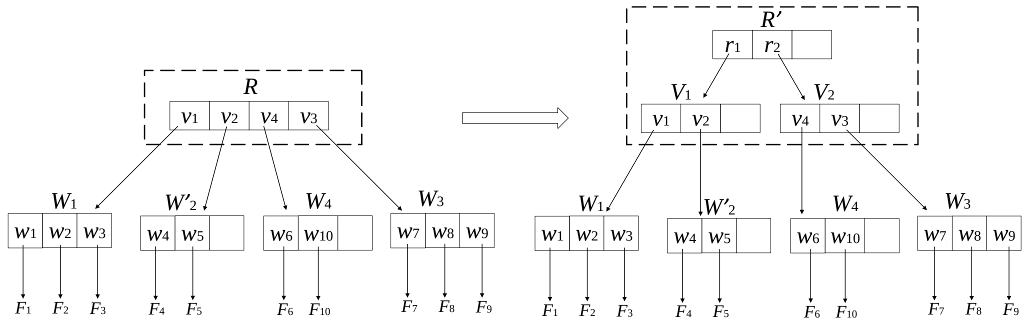
With this Merkle hash tree built over the modified B+ tree, the client keeps track of the root digest. Every time after fetching a data block, the client fetches its corresponding σ as well. Also the client receives the hashing values associated with other entries in the same node along the path from root to the data block. Then the client can verifies the authenticity and freshness with the Merkle tree. Say the client needs to verifies the authenticity and freshness of block F_3 in Figure 1a, where the he/she possesses the root digest $f(R)$. The path from root to F_3 will be $(R \rightarrow W_1)$. For verification, besides σ_3, the client also receives $f(w_1), f(w_2)$ in node W_1 and $f(v_2), f(v_3)$ in node R.



(a) Initial Tree.


R
v1 v2 v3
W1 W2 W3
w1 w2 w3 w4 w5 w6 w10 w7 w8 w9
F1 F2 F3 F4 F5 F6 F10 F7 F8 F9

(b) Insertion.


R
v1 v2 v4 v3
W1 W'2 W4 W3
w1 w2 w3 w4 w5 w6 w10 w7 w8 w9
F1 F2 F3 F4 F5 F6 F10 F7 F8 F9
R'
r1 r2
V1 V2
v1 v2 v4 v3
W1 W'2 W4 W3
w1 w2 w3 w4 w5 w6 w10 w7 w8 w9
F1 F2 F3 F4 F5 F6 F10 F7 F8 F9

(c) Splitting Internal Node and Root Node.

Figure 1: An Example of a Modified B+ tree.

Update. The main manipulations are updating the data block and updating the Merkle tree. Note that the update affects only nodes along a path from a wanted data block to root on the Merkle tree. Therefore, the running time for updating the Merkle tree is $O(logn)$. Also to update the Merkle tree, some hashing values along the path from a data block to root are needed from the server. Clearly, the size of those values will be $O(logn)$. Update operations include Modification, Deletion and Insertion. The update operations over our modified B+ tree mostly conform to the procedures of standard B+ tree. A slight difference lies in the Insertion operation during splitting node, due to the fact that our modified B+ tree doesn't have index information.

First, we discuss Modification and Deletion. To modify a data block, the client simply computes the data block's new corresponding σ and update the Merkle tree with this σ to obtain a new root digest. Then the client uploads the the new data block and the new σ. After receiving this new σ, the server just needs to update the Merkle tree along the path from the data block to root. To delete a data block, the server simply deletes the unwanted data block by the client and then updates the Merkle tree along the path from this data block to root.

Next, we give the details of Insertion. If the leaf node where the new data block will be inserted is not full, the procedure is the same as Modification. Otherwise, the leaf node needs to be split, and then the entry that leads to this leaf node will also be split into two entries, with one entry leading to each leaf node. Note that unlike operations on standard B+ tree, we don't copy the index of the third entry (i.e., the index of the new generated node) to its parent's node. Instead, we just create a new entry simply with a pointer leading to the node and record the corresponding information as defined above. If the node being devided is root node, the depth of this Merkle tree will increment by 1. An example of updating is shown as Figure 1b and 1c. Say the client wants to insert a new file block F_{10} in the 7-th position. First, it locates the position in the way mentioned above. Note that we can locate the 6-th position or the 7-th position. Here we choose to locate the 6-th position and insert a new entry w_{10} behind w_6 in left node W_2 . (If choosing to locate the 7-th position, one should put the new entry before w_7.) Next, the information corresponding to this new file block F_{10} will be written into entry w_{10} with a pointer pointing from w_{10} to F_{10}, as shown in Figure 1b. Since it exceeds the maximum number of entries that a node can have, this leaf node W_2 needs to be split into two leaf nodes, W_2' and W_4 with two non-empty entries in each node (this conforms to the rules of updating on B+ tree), as shown in Figure 1c. At the same time, a new entry v_4 is created in the root node R with a pointer leading v_4 to leaf node W_4. Again, this root node R should be split into two nodes. As shown in the example, this old root node R is divided into two internal nodes, V_1 and V_1. Finally, a new root note R' is built, which has two entries and two pointers leading to V_1 and V_2, respectively. Note that, now the root node has entries r_1 and r_2, where r_1 is the start entry of this tree, meaning $low(r_1) = 1$. We also have $rank(r_1) = rank(V_1) = rank(v_1) + rank(v_2) = 5$ and $rank(r_2) = 5$.

5 Security Proofs

Theorem 1. The proposed static PoR scheme is sound with respect to attacks as specified in Section 3.1, assuming the existence of secure indistinguishability obfuscators, existentially unforgeable signature schemes and secure puncturable PRFs.

Proof. We prove the theorem in a series of games.

Game 0. This original game, Game 0, is simply the challenge game defined in Section 3.1. The following game is played:

1. Pick $k_1 \xleftarrow{\$} \mathcal{K}_1$, $k_2 \xleftarrow{\$} \mathcal{K}_2$ and $k_{mac} \xleftarrow{\$} \mathcal{K}_{mac}$.
2. The public verification key VK is given out as an obfuscation of the verification program Check. Here the circuit Check is padded to be the maximum of itself and the verification program CheckA if necessary.
3. The malicious server $\widetilde{\mathcal{S}}$ specifies a valid Audit-protocol sequence $P = (op_1, op_2, \cdots, op_{\text{poly}(\lambda)})$.
4. The challenger executes the protocol with both $\widetilde{\mathcal{S}}$ and an honest server $\mathcal{S}$.

$\widetilde{\mathcal{S}}$ succeeds, if at any execution of an Audit-protocol, the message sent by $\widetilde{\mathcal{S}}$ differs from that of $\mathcal{S}$ and the challenger does not output reject.

Game 1. This game is the same as Game 0, with one difference. The challenger keeps a list of all ever issued signed tags as part of a store query. If the adversary ever submits a tag t either in initiating a proof-of-storage protocol or as the challenge tag, that is verified as valid under ssk but is not a tag signed by the challenger, the challenger declares failure and aborts.

Game 2. In this game we change the winning condition. First the challenger choose index $\hat{i}$ in $[1, n]$. Then the winning condition enforces an additional check that $\hat{i}$ is in the challenge set I during the Audit protocol.

1. Choose $\hat{i}$ in $[1, n]$ at random.
2. Pick $k_1 \xleftarrow{\$} \mathcal{K}_1$, $k_2 \xleftarrow{\$} \mathcal{K}_2$ and $k_{mac} \xleftarrow{\$} \mathcal{K}_{mac}$.
3. The public verification key VK is given out as an obfuscation of the verification program Check. Here the circuit Check is padded to be the maximum of itself and the verification program CheckA if necessary.
4. The malicious server $\widetilde{\mathcal{S}}$ specifies a valid Audit-protocol sequence $P = (op_1, op_2, \cdots, op_{\text{poly}(\lambda)})$.
5. The challenger executes the protocol with both $\widetilde{\mathcal{S}}$ and an honest server $\mathcal{S}$.

$\widetilde{\mathcal{S}}$ succeeds, if at any execution of an Audit-protocol, (1) $\hat{i} \in I$, and (2) the message sent by $\widetilde{\mathcal{S}}$ differs from that of $\mathcal{S}$ and the challenger does not output reject.

Game 3. In this game the challenger creates the public verification key as an obfuscation of an alternate verification circuit CheckA. First, the challenger computes a puncturing of the secret key k_1 at index $\hat{i}$. Let $y = F_1(k_1, \hat{i})$. The challenger uses the punctured key $k_1\{\hat{i}\}$, punctured value y and the injective one-way function f to generate CheckA.

1. Choose $\hat{i}$ in $[1, n]$ at random.
2. Pick $k_1 \xleftarrow{\$} \mathcal{K}_1$, $k_2 \xleftarrow{\$} \mathcal{K}_2$ and $k_{mac} \xleftarrow{\$} \mathcal{K}_{mac}$. Let $k_1\{\hat{i}\} \leftarrow \mathsf{Puncture}_{F_1}(k_1, \hat{i})$. Let $y = F_1(k_1, \hat{i})$. Let $z = f(y)$.

3. The public verification key VK is given out as an obfuscation of the verification program CheckA.
4. The malicious server $\widetilde{\mathcal{S}}$ specifies a valid Audit-protocol sequence $P = (op_1, op_2, \cdots, op_{\text{poly}(\lambda)})$.
5. The challenger executes the protocol with both $\widetilde{\mathcal{S}}$ and an honest server $\mathcal{S}$.

$\widetilde{\mathcal{S}}$ succeeds, if at any execution of an Audit-protocol, (1) $\hat{i} \in I$, and (2) the message sent by $\widetilde{\mathcal{S}}$ differs from that of $\mathcal{S}$ and the challenger does not output reject.

```
CheckA:
Inputs: $Q = \{(i, v_i)\}_{i \in I}, \mu_1, \cdots, \mu_s, \sigma, t$
Constants: Punctured PRF key $k_1\{\hat{i}\}$, PRF key $k_2$ and MAC key $k_{mac}$. Strings $\hat{i}, z$.
    $n \| \mathsf{MAC}_{k_{mac}}(n) \leftarrow t$
    $\mu_{sum} \leftarrow \sum_{j=1}^{s} F_2(k_2, j)\mu_j$
    if $\hat{i} \in I$ then
        if $f((\sigma - \mu_{sum} - \sum_{(i,v_i)\in Q, i\neq\hat{i}} v_i F_1(k_1, i))/v_{\hat{i}}) = z$ then output 1 else output $\perp$
    else
        if $\sigma - \mu_{sum} = \sum_{(i,v_i)\in Q} v_i F_1(k_1, i)$ then output 1 else output $\perp$
    end if
```

Game 4. In this game the constant y, used to create z in CheckA, is replaced with a random $\lceil \log \mathbb{Z}_p \rceil$-bit string. The other parts of the game do not change.

1. Choose $\hat{i}$ in $[1, n]$ at random.
2. Pick $k_1 \xleftarrow{\$} \mathcal{K}_1$, $k_2 \xleftarrow{\$} \mathcal{K}_2$ and $k_{mac} \xleftarrow{\$} \mathcal{K}_{mac}$. Let $k_1\{\hat{i}\} \leftarrow \mathsf{Puncture}_{F_1}(k_1, \hat{i})$. Choose y at random in $\{0,1\}^{\lceil \log \mathbb{Z}_p \rceil}$. Let $z = f(y)$.
3. The public verification key VK is given out as an obfuscation of the verification program CheckA.
4. The malicious server $\widetilde{\mathcal{S}}$ specifies a valid Audit-protocol sequence $P = (op_1, op_2, \cdots, op_{\text{poly}(\lambda)})$.
5. The challenger executes the protocol with both $\widetilde{\mathcal{S}}$ and an honest server $\mathcal{S}$.

$\widetilde{\mathcal{S}}$ succeeds, if at any execution of an Audit-protocol, (1) $\hat{i} \in I$, and (2) the message sent by $\widetilde{\mathcal{S}}$ differs from that of $\mathcal{S}$ and the challenger does not output reject.

First, we argue that the advantage of the poly-time adversary $\widetilde{\mathcal{S}}$ must be negligibly close in Games 0 and 1. We observe that if there is a difference in the adversary $\widetilde{S}$'s success probability between Games 0 and 1, we can use $\widetilde{S}$ to construct a forger against the signature scheme.

We now argue that the advantage of the adversary $\widetilde{\mathcal{S}}$ in Game 2 is identical to that in Game 1, since the challenger can always include the randomly selected $\hat{i}$ in the challenge set I and clearly, this does not affect the adversary's view.

We then argue that the advantage of the poly-time adversary $\widetilde{\mathcal{S}}$ must be negligibly close in Games 2 and 3. Otherwise, we can construct an attacker $\mathcal{B}$ that breaks the $i\mathcal{O}$ security of the indistinguishability obfuscator. We first demonstrate the functional equivalence of circuits Check and CheckA. Consider a set $Q = \{(i, v_i)\}_{i \in I}$ which is input to the latter circuit. If it holds that $\hat{i} \notin I$, then $F_1(k_1\{\hat{i}\}, i) = F_1(k_1, i)$ for all $i \in I$, since the punctured PRF preserves functionality outside the punctured point. Thus $\sigma = \sum_{(i,v_i)\in Q} v_i F_1(k_1, i) + \sum_{j=1}^{s} F_2(k_2, j)\mu_j \Leftrightarrow \sigma - \mu_{sum} = \sum_{(i,v_i)\in Q} v_i F_1(k_1, i)$. On the other hand, if $\hat{i} \in I$, then $\sigma = \sum_{(i,v_i)\in Q} v_i F_1(k_1, i) + \sum_{j=1}^{s} F_2(k_2, j)\mu_j \Leftrightarrow (\sigma - \mu_{sum} - \sum_{(i,v_i)\in Q, i\neq\hat{i}} v_i F_1(k_1, i))/v_i = F_1(k_1, \hat{i}) \Leftrightarrow f((\sigma - \mu_{sum} - \sum_{(i,v_i)\in Q, i\neq\hat{i}} v_i F_1(k_1, i))/v_i) = f(y)$, since f is injective. Consider the adversary $\mathcal{B} = (Samp, D)$. The algorithm $Samp$ on 1^λ randomly picks $\hat{i}$ in $[1, n]$ and outputs $C_0 = \mathsf{Check}$ and $C_1 = \mathsf{CheckA}$. It sets state $\delta = (k_1, k_2, \hat{i})$. Now the distinguisher D on input $(\delta, i\mathcal{O}(\lambda, C_z))$ sends VK $= i\mathcal{O}(\lambda, C_z)$. If $z = 0$ then $\mathcal{B}$ simulates Game 2. Otherwise, $\mathcal{B}$ simulates Game 3. D will will output 1 if $\widetilde{\mathcal{S}}$ succeeds at some execution of an Audit-protocol such that (1) $\hat{i} \in I$, and (2) the message sent by $\widetilde{\mathcal{S}}$ differs from that of an honest $\mathcal{S}$ and the challenger does not reject. Any attacker $\widetilde{\mathcal{S}}$ with different advantages in the Games leads to $\mathcal{B}$ as an attacker on $i\mathcal{O}$ security.

Next, we argue that the advantage of the poly-time adversary $\widetilde{\mathcal{S}}$ must be negligibly close in Games 3 and 4. Otherwise, we can construct an attacker $\mathcal{B}$ distinguishing the output of the puncturable PRF $F_1(k_1, \cdot)$. Simulator $\mathcal{B}$ interacts with the puncturable PRF challenger while acting as a challenger to $\widetilde{\mathcal{S}}$. First $\mathcal{B}$ picks $\hat{i}$ at random from $[1, n]$. Next $\mathcal{B}$ submits point $\hat{i}$ to the PRF challenger and receives

Scheme	Write Cost on Server	Write Bandwidth	Auditing Cost Server Read	Verifiability	Dynamic Update
Iris[40]	$O(\beta)$	$O(\beta)$	$O(\beta\lambda\sqrt{n})$	private	YES
Cash et al.[17]	$O(\beta\lambda(\log n)^2)$	$O(\beta\lambda(\log n)^2)$	$O(\beta\lambda(\log n)^2)$	private	YES
Shi et al.[39]	$O(\beta \log n) + O(\lambda \log n)$	$O(\beta) + O(\lambda \log n)$	$O(\beta\lambda \log n)$	public	YES
This paper	$O(\beta) + O(\lambda \log n)$	$O(\beta) + O(\lambda \log n)$	$O(\beta\lambda)$	public	YES

Table 1: Comparison with existing dynamic PoRs.

punctured key $k_1\{\hat{i}\}$ and PRF challenge y' in return. $\mathcal{B}$ then computes $z = f(y')$ and computes an obfuscation of CheckA. It sends VK $= i\mathcal{O}(\lambda, \mathsf{CheckA})$. If $y' = F_1(k_1, \hat{i})$ then $\mathcal{B}$ simulates Game 3. Otherwise y' is a random $\lceil \log \mathbb{Z}_p \rceil$-bit string and $\mathcal{B}$ simulates Game 4. $\mathcal{B}$ will output 1 if $\widetilde{\mathcal{S}}$ succeeds at some execution of an Audit-protocol such that (1) $\hat{i} \in I$, and (2) the message sent by $\widetilde{\mathcal{S}}$ differs from that of an honest $\mathcal{S}$ and the challenger does not reject. Any attacker $\widetilde{\mathcal{S}}$ with different advantages in the Games leads to $\mathcal{B}$ as an attacker on the puncturable PRF security.

Finally, if there is an attacker $\widetilde{\mathcal{S}}$ in Game 4, we can construct an attacker $\mathcal{B}$ inverting the one-way function. Simulator $\mathcal{B}$ interacts with the one-way function challenger while acting as a challenger to $\widetilde{\mathcal{S}}$. First $\mathcal{B}$ receives the challenge $z' = f(a)$ as an input, where a is a $\lceil \log \mathbb{Z}_p \rceil$-bit random string. Then $\mathcal{B}$ picks $\hat{i}$ at random from $[1, n]$. Next $\mathcal{B}$ computes punctured key $k_1\{\hat{i}\}$ and an obfuscation of CheckA with z' hardwired in place of z. Since z' is identically distributed to z, the view of $\widetilde{\mathcal{S}}$ is identical to its view in Game 4. $\mathcal{B}$ sends VK. Then with some advantage, say ϵ, $\widetilde{\mathcal{S}}$ succeeds at some execution of an Audit-protocol such that (1) $\hat{i} \in I$, and (2) the message sent by $\widetilde{\mathcal{S}}$ differs from that of an honest $\mathcal{S}$ and the challenger does not reject. Thus $\mathcal{B}$ can computes $a^* = (\sigma - \sum_{j=1}^{s} F_2(k_2, j)\mu_j - \sum_{(i,v_i)\in Q, i\neq \hat{i}} v_i F_1(k_1, i))/v_i$ which satisfies $a^* = f^{-1}(z')$. Therefore, if the one-way function is secure, no poly-time attacker $\widetilde{\mathcal{S}}$ can succeed with non-negligible advantage.

Suppose that there exists a PPT adversary $\widetilde{\mathcal{S}}$ with non-negligible advantage ϵ in breaking the Authenticity of the PoR scheme, i.e. in winning Game 0. Let the maximal advantage of any adversary in distinguishing the output of the indistinguishability obfuscator and in distinguishing the output of the puncturable PRF be $\epsilon_{i\mathcal{O}}$ and ϵ_{PRF}, respectively. The analyses above then imply that $\widetilde{\mathcal{S}}$ has probability at least $\epsilon l/n - \epsilon_{i\mathcal{O}} - \epsilon_{PRF}$ in inverting the one-way function f. Since n and l are polynomially bounded, the first term is non-negligible, and since $i\mathcal{O}$ is a secure indistinguishability obfuscator and F_1 is a secure puncturable PRF, the second and third terms are negligible. This contradicts the non-invertibility of f. It follows that the advantage of $\widetilde{\mathcal{S}}$ in Game 0 must be negligible. This concludes the proof of Theorem 1. ■

Theorem 2. The proposed static PoR scheme satisfies Retrievability as specified in Section 3.1.

Proof Sketch. The proof will be identical to that in [38]. In their work, the proof was divided into two parts, Section 4.2 and 4.3 in [38]. The first part was to prove that the extraction procedure can efficiently reconstruct a ρ fraction of the file blocks when interacting with a prover that provides correctly-computed $\{\mu_j\}$ responses for a non-negligible fraction of the query space. The second part was to prove that a ρ fraction of the blocks of the erasure-coded file suffice for reconstructing the original file. ■

6 Analysis and Comparisons

In this section, we give an analysis of our proposed scheme and then compare it with other two recently proposed schemes.

For the data owner, our scheme requires to generate an obfuscated program during the preprocessing stage of the system. With the current obfuscator candidate, it indeed costs the data owner a somewhat large amount of overhead, but this is a one-time effort which can be amortized over plenty of operations in the future. Thus, we focus on the analysis on the computation and communication overheads incurred during writing and auditing operations rather than those in the preprocessing step. Like SW's privately-verifiable PoR system the data owner can efficiently store files on the cloud server, and it takes the cloud server less overhead during an auditing protocol than in a public-key-based scheme. The cost on the client device is mainly incurred by the operations over symmetric key primitives, which are known to be much faster than public key cryptographic primitives. The cost analysis on the server side is shown as Table 1.

In Table 1 showing a comparison with existing dynamic PoR schemes we let β be the block size in number of bits, λ be the security parameter and n be the number of blocks. We compare our scheme with the state-of-the-art scheme [39], since a comparison between Shi et al.'s scheme and Cash et al.'s scheme is given in [39]. Note that Shi et al.'s scheme needs amortized cost $O(\beta \log n)$ for writing on the server side, due to the fact that an erasure-coding needs to be done on the entire data file after $\Theta(n)$ updates, while our scheme uses an erasure code that works on file blocks, instead of taking the entire file as inputs (more details and discussions can be found in Section 4). That means, in our system modifying a block doesnot require a change of the erasure codes of the entire file. Thus, the cost for writing is only proportional to the block size being written. On the other hand, during an auditing protocol, Shi et al.'s scheme incurs overhead $O(\beta\lambda \log n)$ on the server side, due to the features of the server-side storage layout. In their scheme, one single file will be stored as three parts, including raw data part R, erasure-coded copy of the entire file C and hierarchical log structure part H that stores the up-to-date file blocks in erasure-coded format. Thus, during one auditing operation, Shi et al.'s scheme needs to check $O(\lambda)$ random blocks from C and $O(\lambda)$ random blocks from each filled level in H. While, in our scheme, the server performs every writing over the wanted block directly, not storing the update block separately. Thus, our scheme only requires $O(\lambda)$ random blocks of one file to check authenticity during auditing. (Note that this $O(\lambda)$ usually would be $\Omega(\sqrt{n\beta})$ if no pseudorandom permutation over the locations of the file blocks is performed, because a small number proportional to $O(\lambda)$ might render the system insecure. Please refer to [17] for more details.) Note that it is most likely that the auditing protocol is executed between a well-equipped verification machine and the server, and the operations on server side only involve symmetric key primitives. Therefore, it will not have noticeable effects on the system's overall performance.

Clearly, the improvement in our work mainly results from $i\mathcal{O}$'s power that secret keys can be embedded into the obfuscated verification program without secret keys being learnt by user. However, the current obfuscator candidate [24] provides a construction running in impractical, albeit polynomial, time. (Note that it is reasonable and useful that the obfuscated program is run on well-equipped machines.) Although $i\mathcal{O}$'s generation and evaluation is not fast now [3], studies on implementing practical obfuscation are developing fast [2]. It is plausible that obfuscations with practical performance will be achieved in the not too distant future. Note that the improvement on obfuscation will directly lead to an improvement on our schemes.

7 Discussions and Future Directions

As pointed out in [24], the current obfuscation constructions runs in impractical polynomial-time, and it is an important objective to improve the efficiency for $i\mathcal{O}$ usage in real life applications. Also Apon et al.'s have shown the inefficiency in $i\mathcal{O}$'s generation and evaluation in [3] CRYPTO'14. In this section, we will give some discussions on the above three possible future directions in Obfuscation, in addition to those discussed in [24].

7.1 Outsourced and Joint Generation of Indistinguishability Obfuscation

Image the scenario in our proposed publicly verifiable PoR system, where users store their data on the same cloud server using the same PoR scheme but with different private keys. One naive approach with $i\mathcal{O}$ would be requiring each user to generate his/her own individual obfuscated program for public verification. This means that each user needs to afford the prohibitively expensive overhead for $i\mathcal{O}$'s generation on his/her own. Note that for the same PoR scheme, the verification procedures are the same but with different user's private key. Also note that each user "embeds" his/her own private keys into the obfuscated verification in a way that anyone else can't learn anything about the embedded private information. Hence, we can have several users jointly and securely generate one obfuscated verification program, where each user uses his/her own private key as part of the input to the generation. One promising way to do that could be Secure multiparty computation technique. Observe that this generated obfuscated program has almost the same computation as the one with only one user's private key embedded. The only differences between this jointly generated obfuscation and the individual-user-generated obfuscation are that (1) the jointly generated obfuscation is implanted with more than one user's private key; (2) the jointly generated obfuscation needs one more step to identify which user's private key it will use.

On the other hand, outsourced computing is becoming more and more popular nowadays. It is useful in applications where relatively low-power devices need to compute expensive and time-consuming functions. To address this, one can use computation delegation schemes to securely outsource those functions' computations to a powerful cloud server. Clearly, as for relatively low-power individual computers, the overhead caused by the current $i\mathcal{O}$ construction candidate is impractical. Thus, it would be promising to find a specific way to efficiently outsource $i\mathcal{O}$'s generation.

7.2 Reusability and Universality of Indistinguishability Obfuscation

Reusability is related to $i\mathcal{O}$'s joint generation to some extent. In the scenario considered above, the jointly generated obfuscated program is embedded with a group of users' private key. This means that the same obfuscated program can be used by verifiers on behalf of different users in this group.

Universality is relevant to an obfuscated program's functionalities. More Concretely, an universal $i\mathcal{O}$ is supposed to support multiple functionalities. A *straightforward* example would be the obfuscated-based functional encryption scheme in [24]. Recall that in their construction, the secret key sk_f for a function f is an obfuscated program. For this obfuscated program to become universal, sk_f would need to be associated with more than one function. In this case, e.g., an universal obfuscated program sk_f can be associated with a class of similar functions $f = (f_1, f_2, \cdots, f_k)$. This means that sk_f's holder can obtain $f_1(m), f_2(m), \cdots, f_k(m)$ from an encryption of m.

Recently, Hohenberger et al. [30] has shown that $i\mathcal{O}$ can provide some other cryptographic primitives with universality. They employed $i\mathcal{O}$ to construct universal signature aggregators, which can aggregate across schemes in various algebraic settings (e.g., RSA, BLS). Prior to this universal signature aggregator, the aggregation of signatures can only be built if all the signers use the same signing algorithm and shared parameters. On the contrary, the universal signature aggregator enables the aggregation of the users' signatures without requiring them to execute the same signing behavior, which indicates a compressed authentication overhead.

7.3 Obfuscation for Specific Functions

The current $i\mathcal{O}$ construction candidate provides a way for obfuscating general circuits and runs in impractical polynomial-time. Instead, an obfuscation designed for some particular simple function with practical performance, such as computing two vectors' inner product, can also be wanted. (Wee's work in STOC'05 [42] is one of them.) This means that we can't use Garg et al's obfuscator, and we want to obfuscate such simple functions in a practical way that might be specific for those functions. Note that, for example, a practical obfuscated program computing the inner product of two vectors, where one vector is an input to this program and the one is embedded into the program without user learning its knowledge, could be useful in applications like computational biometrics. Also, note that it is really likely that such a practical obfuscation for a specified function can be used as a black-box to construct an obfuscation supporting more complex functionalities by combining with other existing practical cryptographic primitives.

8 Conclusions

In this paper, we explore *indistinguishability obfuscation* to construct a publicly verifiable Proofs-of-Retrievability (PoR) scheme that is mainly built upon symmetric key cryptographic primitives. We also extend the proposed scheme to support dynamic updates using a combination of a modified B+ tree and a standard Merkle hash tree. By analysis and comparisons with other existing schemes, we show that our scheme is efficient on the data owner side and the cloud server side. Although it consumes a considerable amount of overheads to generate an obfuscation, it is only a one-time effort during the preprocessing stage of the system. Therefore, this cost can be amortized to all of future auditing operations. After that, we discuss three possible future directions in obfuscation: (I) Outsourced and Joint Generation of Indistinguishability Obfuscation, (II) Reusability and Universality of Indistinguishability Obfuscation and (III) Obfuscation for Specific Functions, in addition to those discussed in [24]. Given $i\mathcal{O}$'s power as a cryptographic primitive and its nascent stage in development, studies on constructing more practical obfuscation are developing fast [2]. It is plausible that obfuscations with practical performance will be achieved in the not too distant future.

9 Acknowledgments

References

[1] Ananth, P., Boneh, D., Garg, S., Sahai, A., and Zhandry, M. (2013). Differing-inputs obfuscation and applications. *IACR Cryptology ePrint Archive*, 2013:689.

[2] Ananth, P., Gupta, D., Ishai, Y., and Sahai, A. (2014). Optimizing obfuscation: Avoiding barrington's theorem. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 646–658. ACM.

[3] Apon, D., Huang, Y., Katz, J., and Malozemoff, A. J. (2014). Implementing cryptographic program obfuscation. `http://eprint.iacr.org/`.

[4] Armknecht, F., Bohli, J.-M., Karame, G. O., Liu, Z., and Reuter, C. A. (2014). Outsourced proofs of retrievability. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 831–843. ACM.

[5] Ateniese, G., Di Pietro, R., Mancini, L. V., and Tsudik, G. (2008). Scalable and efficient provable data possession. In *Proceedings of the 4th international conference on Security and privacy in communication netowrks*, page 9. ACM.

[6] Ateniese, G., Kamara, S., and Katz, J. (2009). Proofs of storage from homomorphic identification protocols. In *Advances in Cryptology–ASIACRYPT 2009*, pages 319–333. Springer.

[7] Barak, B., Bitansky, N., Canetti, R., Kalai, Y. T., Paneth, O., and Sahai, A. (2014). Obfuscation for evasive functions. In *Theory of Cryptography*, pages 26–51. Springer.

[8] Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., and Yang, K. (2001). On the (im) possibility of obfuscating programs. In *Advances in CryptologyCRYPTO 2001*, pages 1–18. Springer.

[9] Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., and Yang, K. (2012). On the (im) possibility of obfuscating programs. *Journal of the ACM (JACM)*, 59(2):6.

[10] Bellare, M., Stepanovs, I., and Tessaro, S. (2014). Poly-many hardcore bits for any one-way function and a framework for differing-inputs obfuscation. In Sarkar, P. and Iwata, T., editors, *Advances in Cryptology ?ASIACRYPT 2014*, volume 8874 of *Lecture Notes in Computer Science*, pages 102–121. Springer Berlin Heidelberg.

[11] Benabbas, S., Gennaro, R., and Vahlis, Y. (2011). Verifiable delegation of computation over large datasets. In *Advances in Cryptology–CRYPTO 2011*, pages 111–131. Springer.

[12] Blum, M., Evans, W., Gemmell, P., Kannan, S., and Naor, M. (1994). Checking the correctness of memories. *Algorithmica*, 12(2-3):225–244.

[13] Boneh, D. and Waters, B. (2013). Constrained pseudorandom functions and their applications. In *Advances in Cryptology-ASIACRYPT 2013*, pages 280–300. Springer.

[14] Boneh, D. and Zhandry, M. (2014). Multiparty key exchange, efficient traitor tracing, and more from indistinguishability obfuscation. In *Proceedings of CRYPTO 2014*.

[15] Bowers, K. D., Juels, A., and Oprea, A. (2009). Proofs of retrievability: Theory and implementation. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, pages 43–54. ACM.

[16] Brakerski, Z. and Rothblum, G. N. (2014). Virtual black-box obfuscation for all circuits via generic graded encoding. In *Theory of Cryptography*, pages 1–25. Springer.

[17] Cash, D., Küpçü, A., and Wichs, D. (2013). Dynamic proofs of retrievability via oblivious ram. In *Advances in Cryptology–EUROCRYPT 2013*, pages 279–295. Springer.

[18] Coron, J.-S., Lepoint, T., and Tibouchi, M. (2013). Practical multilinear maps over the integers. In *Advances in Cryptology–CRYPTO 2013*, pages 476–493. Springer.

[19] Dodis, Y., Vadhan, S., and Wichs, D. (2009). Proofs of retrievability via hardness amplification. In *Theory of Cryptography*, pages 109–127. Springer.

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

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

[22] Garg, S., Gentry, C., and Halevi, S. (2013a). Candidate multilinear maps from ideal lattices. In *Eurocrypt*, volume 7881, pages 1–17. Springer.

[23] Garg, S., Gentry, C., Halevi, S., and Raykova, M. (2014a). Two-round secure mpc from indistinguishability obfuscation. In *Theory of Cryptography*, pages 74–94. Springer.

[24] Garg, S., Gentry, C., Halevi, S., Raykova, M., Sahai, A., and Waters, B. (2013b). Candidate indistinguishability obfuscation and functional encryption for all circuits. In *Foundations of Computer Science (FOCS), 2013 IEEE 54th Annual Symposium on*, pages 40–49. IEEE.

[25] Garg, S., Gentry, C., Halevi, S., and Wichs, D. (2014b). On the implausibility of differing-inputs obfuscation and extractable witness encryption with auxiliary input. In Garay, J. and Gennaro, R., editors, *Advances in Cryptology ?CRYPTO 2014*, volume 8616 of *Lecture Notes in Computer Science*, pages 518–535. Springer Berlin Heidelberg.

[26] Giuseppe, A., Randal, B., Reza, C., Herring, J., Kissner, L., Peterson, Z., and Song, D. (2007). Provable data possession at untrusted stores. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 598–609. ACM.

[27] Goldwasser, S., Gordon, S. D., Goyal, V., Jain, A., Katz, J., Liu, F.-H., Sahai, A., Shi, E., and Zhou, H.-S. (2014). Multi-input functional encryption. In *Advances in Cryptology–EUROCRYPT 2014*, pages 578–602. Springer.

[28] Goldwasser, S. and Rothblum, G. N. (2007). On best-possible obfuscation. In *Theory of Cryptography*, pages 194–213. Springer.

[29] Goyal, V., Jain, A., Koppula, V., and Sahai, A. (2013). Functional encryption for randomized functionalities. *IACR Cryptology ePrint Archive*, 2013:729.

[30] Hohenberger, S., Koppula, V., and Waters, B. (2014a). Universal signature aggregators. `http://eprint.iacr.org/`.

[31] Hohenberger, S., Sahai, A., and Waters, B. (2014b). Replacing a random oracle: Full domain hash from indistinguishability obfuscation. In *Advances in Cryptology–EUROCRYPT 2014*, pages 201–220. Springer.

[32] Juels, A. and Kaliski Jr, B. S. (2007). Pors: Proofs of retrievability for large files. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 584–597. ACM.

[33] Küpçü, A. (2010a). *Efficient Cryptography for the Next Generation Secure Cloud.* PhD thesis, Brown University.

[34] Küpçü, A. (2010b). *Efficient Cryptography for the Next Generation Secure Cloud: Protocols, Proofs, and Implementation.* Lambert Academic Publishing.

[35] Naor, M. and Rothblum, G. N. (2005). The complexity of online memory checking. In *Foundations of Computer Science, 2005. FOCS 2005. 46th Annual IEEE Symposium on*, pages 573–582. IEEE.

[36] Ramchen, K. and Waters, B. (2014). Fully secure and fast signing from obfuscation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 659–673. ACM.

[37] Sahai, A. and Waters, B. (2014). How to use indistinguishability obfuscation: Deniable encryption, and more. In *Proceedings of the 46th Annual ACM Symposium on Theory of Computing, STOC '14*, pages 475–484. ACM.

[38] Shacham, H. and Waters, B. (2008). Compact proofs of retrievability. In *Advances in Cryptology-ASIACRYPT 2008*, pages 90–107. Springer.

[39] Shi, E., Stefanov, E., and Papamanthou, C. (2013). Practical dynamic proofs of retrievability. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 325–336. ACM.

[40] Stefanov, E., van Dijk, M., Juels, A., and Oprea, A. (2012). Iris: A scalable cloud file system with efficient integrity checks. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 229–238. ACM.

[41] Wang, Q., Wang, C., Li, J., Ren, K., and Lou, W. (2009). Enabling public verifiability and data dynamics for storage security in cloud computing. In *Computer Security–ESORICS 2009*, pages 355–370. Springer.

[42] Wee, H. (2005). On obfuscating point functions. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, pages 523–532. ACM.