Official Arbitration with Secure Cloud Storage Application

Alptekin Küpçü
Koç University, İstanbul, Turkey
akupcu@ku.edu.tr

February 11, 2013

Abstract

Static and dynamic proof of storage schemes have been proposed for use in secure cloud storage scenarios. In this setting, a client outsources storage of her data to a server, who may, willingly or not, corrupt the data (e.g., due to hardware or software failures), or delete infrequently accessed parts to save space. Most of the existing schemes only solve part of this problem: The client may ask for a cryptographic proof of integrity from the server. But what happens if this proof fails to verify?

We argue that in such a case, both the client and the server should be able to contact an official court, providing cryptographic proofs, so that the Judge can resolve this dispute. We show that this property is stronger than what has been known as *public verifiability* in the sense that official arbitration should handle a malicious client as well. We clearly show this formalization difference, and then present multiple schemes that work for various static and dynamic storage solutions in a generic way. We implement our schemes and show that they are very efficient, diminishing the validity of arguments against their use, where the overhead for adding the ability to resolve such disputes at a court is only **2 ms** and **80 bytes** for each update on the stored data, using standard desktop hardware.

Finally, we note that disputes may arise in many other situations, such as when two parties exchange items (e.g., e-commerce) or agree on something (e.g., contract-signing). We show that it is easy to extend our official arbitration protocols for a general case, including dynamic authenticated data structures.

Keywords: provable data possession; dynamic provable data possession; secure cloud storage; public verifiability; fair exchange; cryptographic protocols

1 Introduction

Efficient secure cloud storage protocols have been proposed since 2007, starting with provable data possession (PDP) work of Ateniese et al. [5] and proof of retrievability work of Juels and Kaliski [47], followed by many others [67, 16, 29, 34, 6, 48]. In these scenarios, the client outsources storage of her files to a server (e.g., Dropbox, Amazon S3, Google Drive, Microsoft Skydrive), but does not necessarily fully trust the server. In particular, the client would like to have some warranty over the integrity of her files. To achieve this, the client stores some metadata M, and may later challenge the server on some random subset of blocks of her data, and obtain a cryptographic proof of integrity in return. If this proof verifies against the client's metadata, with high probability, the client's whole data is intact, and thus the client may rest in peace.

The problem begins when the server's proof fails to verify. In this case, ideally, the client and the server should be able to resolve their dispute at a court. By obtaining cryptographic evidence from both parties, the Judge should be able to rule the correct decision. Previously, a related property named *public verifiability* has been proposed. Unfortunately, we show that public verifiability is not enough for the Judge to rule, by showing that a dishonest client may frame an honest server in the publicly-verifiable version of PDP (see Section 3). The main reason is that, public verifiability property was intended to enable third parties to perform verification, whereas in our model, the client is assumed to be potentially malicious, since there are potential monetary gains behind framing an honest server. For example, Amazon may offer a warranty in case of data loss, and a malicious client may want to obtain this warranty nevertheless. Note that such a new business model for cloud storage with warranty is desirable, since it may bring enterprise customers. Thus, we formalize official arbitration different from public verifiability to ensure that the resulting protocol can be used by the Judge officially.

Next, we observe that there is a generic and easy solution for this issue in the static setting. In a static setting, the client's data does not change over time (even by the client herself). The basic idea is to obtain the server's signature on the metadata M once when the file is uploaded. At this time, if the server's signature fails to verify, the client may assume that her files are not backed up. If the signature verifies, and later some dispute occurs, the client can present this metadata M, together with the server's signature on it, to the Judge. At this point, through the use of secure cloud storage proofs (i.e., the server's proof that the file is intact), the Judge can arbitrate between the client and the server.

When we move to the dynamic setting where the client herself updates her data (and thus the metadata), the straightforward generalization of the idea above leads to a very inefficient solution. We provide a scalable and efficient solution in Section 4, that works on almost all dynamic provable data possession (DPDP) scenarios defined by Erway et al. [37].[1] We provide security proofs and performance measurements. We have implemented our protocols, and will be releasing the source code publicly.

We then extend our dispute resolution idea to other domains. Many cryptographic scenarios require a dispute resolution between participants. Additional examples include the electronic commerce setting [27, 28, 10, 54] where the buyer and the seller may be at a dispute, or an outsourced computation setting [9] where the boss and the contractor(s) may be disputing the correctness of the computation. We show how our official arbitration techniques can be generalized to provide dispute resolution in other cases.

Related Work: Some previous schemes offered a property called *public verifiability*, where the client makes some metadata public so that other users can verify the integrity of the client's data on the server [5]. Yet, those schemes cannot be used for official arbitration purposes, since the client may publish invalid metadata, and hence frame an honest server (see Section 3). The PDP model simply assumes the client is trusted; an assumption that does not hold in our scenario where the client has potential monetary gains from a dispute.

Shah et al. [68] provide an audit (i.e., public verifiability) method for a static storage scenario with zero-knowledge property (i.e., the file content is hidden from the auditor). Yet, their protocol is not optimistic (the Judge needs to be involved during the upload of the file).

Wang et al. initially proposed a public verifiability protocol where the Judge (TPA) needs to keep a long-term state for each client-server-file triple [70] but later provided a mechanism to arbitrate between the client and server without a stateful Judge [69]. They furthermore provided a novel batch audit mechanism. Unfortunately, their scheme applies on top of their own construction only, and hence is not general like ours. Furthermore, their arbitration requires

[1] We will observe that our scheme works easily on any DPDP protocol that does not employ a secret key for verification.

bilinear maps and random oracle model, while no formal security definition is given. Actually, our arbitration mechanisms can also be applied on top of [70] with a very minor modification[2], and our performance will be better. (In [69] authors obtain about 470ms overhead for individual audits, and 370ms overhead for batch audits on top of the underlying DPDP system. See Section 5 for our 2ms overhead.)

Recently, Zheng and Xu presented FDPOR [72], which claims to achieve dynamic proof of retrievability and fairness in the random oracle model. Unfortunately, their fairness definition fails to completely capture the case where a malicious client may frame an honest user. For example, using their fairness definition, one cannot show the attack on PDP that we show in Section 3. A better definition would state that if the server can prove integrity of the data to an honest client, then the server should also be able to prove integrity to the Judge (actually, their protocol does not achieve public-verifiability either).

Official arbitration, to the best of our knowledge, is a new formalization. Our schemes can be applied on top of *any* DPDP protocol where the client has no secret key, as defined by Erway et al. [37]. Note that, our schemes are the only schemes that can be used for official arbitration (not public verifiability) of any DPDP scheme, to the best of our knowledge. Therefore, in the performance section, we only compare two of our schemes. Furthermore, again to the best of our knowledge, our scheme in Section 3 is the only scheme that can be applied on top of any (static) PDP protocol to provide official arbitration.

Contributions: We provide the *first constructions of general-purpose arbitration schemes* that are applicable to various static and dynamic scenarios, and capable of performing fully-automated arbitration by a Judge. Our model builds upon the DPDP model [37], and thus is applicable to a wide range of protocols. We believe, the addition of our scheme on top of DPDP protocols may **greatly enhance the use of secure cloud storage in enterprise settings**, since now the client and the server may resolve their disputes officially at a court, paying very low computation and communication overhead. Our solutions further have the following benefits:

- Our methods are *optimistic* in the sense that the trusted third party only gets involved in case of a dispute (similar to an *optimistic* fair exchange).
- Our method requires only $O(1)$ expensive operations compared to a straightforward method which requires $O(n)$ such operations when performed n times (e.g., for n updates over the stored data).
- Our method improves efficiency of roughly 25000 updates from **7 hours** to **51 seconds** and from **610 MB** to **2 MB** for real workloads.
- Our method's **per-update overhead** is roughly **2 ms** and **80 bytes**.

Our additional contributions include a generalization of official arbitration to other scenarios, with formal security definitions.

2 Definitions

Arbitration first requires a claim c to be made. We assume the claim is something against *the Blamed* (denoted $\mathcal{P}_B$). Once there is a claim, there must be (at least one) evidence e either supporting the claim, or falsifying it. We will call the party making the claim *the Claimer*

[2]In their scheme, metadata signed by the client is stored at the server, therefore during the claim it needs to be sent to Judge by the server instead of the client. This is the only necessary modification.

(denoted $\mathcal{P}_C$) and the party providing the evidence *the Responder* (denoted $\mathcal{P}_R$). Depending on the scenario, these roles may be played by different parties, or sometimes an entity may play more than one of these roles. The arbitrating party will naturally be called *the Judge* (denoted $\mathcal{J}$).[3]

Definition 2.1 (Official Arbitration Scheme)**.** *An official arbitration scheme involves four parties: the Claimer $\mathcal{P}_C$, the Blamed $\mathcal{P}_B$, the Responder $\mathcal{P}_R$, and the Judge $\mathcal{J}$. The following (interactive) algorithms, as described below, will be executed by the responsible parties, in this order:* `claim`*,* `request`*,* `prove`*,* `decide`*.*

The protocol will start by the Claimer $\mathcal{P}_C$ running the following algorithm and sending the claim to the Judge. To allow for general claims to be made, the input to all the algorithms below include the entire state of the executor (even though we do not explicitly show state as an input):

`claim`$() \to c$: This probabilistic polynomial time (PPT) algorithm is executed by the Claimer to create a claim c using the current state of the Claimer. The claim c is assumed to inherently include information about the Blamed $\mathcal{P}_B$.

Normally, once the Claimer sends the claim to the Judge, one would expect the Judge to transfer the claim directly to the Responder. Yet, to allow for the cases where the Judge may want to hide some details of the claim from the Responder, or may send a related claim which is not exactly the same as the original claim c, we allow the Judge to process the claim using the following function:

`request`$(c) \to c'$: This PPT algorithm is executed by the Judge to use the given claim c to obtain a (related) claim c' to be forwarded to the Responder $\mathcal{P}_R$.

Once the Responder receives this request from the Judge, (s)he is expected to respond with some evidence; either supporting the claim, or falsifying it. The algorithm used by the Responder is defined as follows:

`prove`$(c') \to e$: This PPT algorithm is executed by the Responder upon receipt of the claim c' from the Judge. The resulting evidence e is sent back to the Judge.

To make it more clear and precise, we shall analyze the `prove` algorithm in two cases: The case where the resulting evidence is a *supporting evidence*, and the case where the resulting evidence is a *falsifying evidence*. For example, if the Claimer and the Responder are the same party, then the evidence is expected to be a supporting evidence to the claim. Similarly, when the Responder and the Blamed are the same party, we expect the evidence to be a falsifying evidence. We now analyze both cases.

Supporting Responder: Here we consider the case where the Responder is supporting the Claimer. It may be the case that the Responder is the Claimer himself (i.e., $\mathcal{P}_R = \mathcal{P}_C$), or some other party supporting the Claimer. The algorithm used by the Supporting Responder is the following:

$\texttt{prove}_S(c') \to e$: This PPT algorithm produces a *supporting* evidence *for* the claim.

[3] In general, the arbitrating party may be also called *the Arbiter*, but we will reserve that keyword for the arbitrating party in optimistic fair exchange protocols .

Falsifying Responder: Here we consider the case where the Responder is falsifying the claim made by the Claimer. It may be the case that the Responder is the Blamed herself (i.e., $\mathcal{P}_R = \mathcal{P}_B$) therefore defending herself, or some other party supporting the Blamed by falsifying the claim. The algorithm used by the Falsifying Responder is the following:

$\texttt{prove}_F(c') \rightarrow e$: This PPT algorithm produces a *falsifying* evidence *against* the claim.

Note that the Responder may be a party completely outside the claim ($\mathcal{P}_R \neq \mathcal{P}_B \wedge \mathcal{P}_R \neq \mathcal{P}_C$), but still helping the Blamed by providing a falsifying evidence (via $\texttt{prove}_F$), or helping the Claimer by providing supporting evidence (via $\texttt{prove}_S$). In such cases, the Claimer will be the Judge's proxy for the Supporting Responder, and the Blamed will be the Judge's proxy for the Falsifying Responder, just as in the real courts. We may use $\texttt{prove}$ notation when it is clear from the context whether the Responder is supporting or falsifying the claim.

Finally, based on the claim and the evidence given, the Judge must decide whether or not the claim holds, and detect the guilty party. The decision algorithm and its possible outputs are explained below:

$\texttt{decide}(c, e) \rightarrow \{O, C, B, N\}$: This PPT algorithm is run by the Judge to decide the case based on the evidence provided for, or against the claim. The algorithm may output the following:

- O: This output means that everything is OK and no further action is necessary.
- C: This output signals that the malicious Claimer $\mathcal{P}_C$ is trying to frame the honest Blamed $\mathcal{P}_B$. In this case, the Claimer may be punished.
- B: This output signals that the claim against the Blamed $\mathcal{P}_B$ holds. In this case, the Blamed may be punished.
- N: This output signals that no single cheating party can be identified.

MODEL: Even though we have not explicitly specified, we allow the algorithms to be interactive algorithms (e.g., the processing of evidence may require multiple interactions with the Responder). Furthermore, the Judge may have a setup algorithm that may be used to generate keys to be used by other parties to communicate with the Judge, or to perform other necessary setup. We make no assumptions about the communication model except that we assume the claim and the evidence will eventually reach the Judge intact, and the adversary cannot prevent this.

2.1 Security Definitions

Intuitively, security of an official arbitration scheme should be based on the claim's validity and the Judge's decision. A claim is **valid** if and only if[4] either one of the following two conditions hold:

1. A valid supporting evidence is provided.
2. All falsifying evidence provided is invalid.

Similarly, a claim will be called **invalid** if and only if[4] either one of the following two conditions hold:

[4]This is not necessarily an *if and only if* condition in real life. But defining it this way helps automating the decision process for cryptographic applications, without affecting the results shown in this paper.

1. A valid falsifying evidence is provided.

2. All supporting evidence provided is invalid.

The security definitions of official arbitration schemes will be based on the observations and definitions above. For the sake of simplicity, we will separate the security definitions for the case where we have a Falsifying Responder from that of the case where we have a Supporting Responder. We will start with the case with the Falsifying Responder, and use $\texttt{prove}_F$ to denote the Responder's algorithm.

Definition 2.2 (Secure Official Arbitration with Falsifying Responder)**.** *An official arbitration scheme with a Falsifying Responder is secure if and only if the following two conditions hold:*

1. *$\forall c$ that is a **valid** claim, $\forall$ evidence e,*

$$Pr[\texttt{decide}(c, e) \rightarrow \{B\}] = 1 - neg(k)$$

2. *$\forall c$ that is an **invalid** claim, $\exists$ evidence e output through the process $\texttt{request}(c) \rightarrow c', \texttt{prove}_F(c') \rightarrow e$ run by the Judge and the Responder, respectively,*

$$Pr[\texttt{decide}(c, e) \rightarrow \{B\}] = neg(k)$$

where the probabilities are taken over the random choices of the Judge and the Responder, and the $neg(k)$ is a negligible function over some security parameter k of the system which depends on the claim c (and the Judge's setup).

The definition formally states that no valid claim should be falsifiable with any evidence, except with negligible probability. Similarly, for any invalid claim, there must be a falsifying evidence that convinces the Judge that the Blamed is innocent, so that the Judge wrongly punishes the Blamed only with negligible probability. Note that the definitions are based on the end result of the `decide` algorithm, not the process itself.

Next, we will define security for official arbitration where the Responder is providing supporting evidence, and thus use $\texttt{prove}_S$ to denote the Responder's algorithm.

Definition 2.3 (Secure Official Arbitration with Supporting Responder)**.** *An official arbitration scheme with a Supporting Responder is secure if and only if the following two conditions hold:*

1. *$\forall c$ that is a **valid** claim, $\exists$ evidence e output through the process $\texttt{request}(c) \rightarrow c', \texttt{prove}_S(c') \rightarrow e$ run by the Judge and the Responder, respectively,*

$$Pr[\texttt{decide}(c, e) \rightarrow \{B\}] = 1 - neg(k)$$

2. *$\forall c$ that is an **invalid** claim, $\forall$ evidence e,*

$$Pr[\texttt{decide}(c, e) \rightarrow \{B\}] = neg(k)$$

where the probabilities are taken over the random choices of the Judge and the Responder, and the $neg(k)$ is a negligible function over some security parameter k of the system which depends on the claim c (and the Judge's setup).

This definition formalizes the idea that the Responder should be able to convince the Judge using a supporting evidence to any valid claim (except with negligible probability); whereas for any invalid claim, no supporting evidence will help validating the claim, so that the Judge wrongly punishes the Blamed only with negligible probability.

Definition 2.4 (Invalid-Claim-Punishing Official Arbitration Scheme)**.** *An alternative formulation to the case of an invalid claim of both definitions may instead require* $Pr[\texttt{decide}(c, e) \rightarrow \{C\}] = 1 - neg(k)$*, suggesting that the Claimer* $\mathcal{P}_C$ *is trying to frame the honest Blamed* $\mathcal{P}_B$*, and thus the Claimer should be punished for keeping the Judge busy with such a case. We will call schemes satisfying this property* ***invalid-claim-punishing*** *official arbitration schemes.*

Note that our definitions assume that the security parameter k depends on the claim c for the sake of generality. When we see example scenarios, it will be clear from the context what k denotes. Moreover, note that we used the same negligible function notation in the definitions for simplicity, yet in reality a different negligible function may be used for each probability, or a proper inequality may be used instead (i.e., $\geq 1 - neg(k)$ and $\leq neg(k)$).

Furthermore, normally, one would expect that the Responder has a single chance to respond, and the evidence provided in that single response would let the Judge decide that the claim is either valid or invalid (this prevents unlimited rounds of trying to provide a useful evidence). Therefore, we need that any proposed official arbitration scheme must provide a `prove` algorithm that can output "the correct" evidence. Then, the basic `decide` algorithm would validate or invalidate the claim by checking only a single evidence. If the evidence supports or does not falsify the claim, then the claim would be deemed valid, and if the evidence falsifies or does not support the claim, then the claim would be deemed invalid. Therefore, **in any official arbitration scheme, it is extremely important that the `prove` and `decide` algorithms have matching semantics**.

Lastly, it would be possible to extend the above case, where the Responder has a single chance to respond, to a more interactive response sequence. If `decide` and `prove` are allowed to be ***interactive*** algorithms, then such a scenario may be realized. Moreover, it is possible to imagine **multiple responders, some of whom being Supporting Responders and some others being Falsifying Responders**. In such a setting, the Judge may choose to take the majority opinion. Yet, there are big risks associated with such distributed settings. As one example, it has been shown that having multiple trusted entities in an optimistic fair exchange protocol may completely destroy fairness under certain conditions [51]. Therefore, we keep this complicated setting outside the scope of this paper and leave it as future work.

Notation: Throughout the paper, we will denote a signature by a party P on a value x as $sign_P(x)$. If we are talking about fair exchange of items, we will name the trusted third party (TTP) as the *Arbiter*. If we mean an official arbitration, then we will call the TTP the *Judge*. The notation $VE_{Arb}(value; label)$ denotes a verifiable encryption under the public key of some TTP (e.g., Arbiter) of the *value* such that it can be verified without decrypting that the *value* satisfies some requirement (e.g., when an Endorsed E-Cash coin [23] is encrypted using Camenisch-Shoup verifiable escrow [24], then it can be verified whether or not the escrow contains a valid coin without revealing the coin), together with a public *label* stating any conditions and information about when the TTP should decrypt the verifiable escrow (e.g., the label can say that the TTP should decrypt this verifiable escrow only when a party P provides a valid signature on x.). A negligible function $neg(k)$ is smaller than $1/poly(k)$ for any polynomial function $poly(k)$ in k, $\forall k > k_0$ where k_0 is a constant. Full definitions for underlying DPDP protocols may be found in the Appendix.

3 Arbitration for Static Storage

In static PDP schemes, there is an initial protocol between the client and the server where the client uploads her (encoded) data F to the server, while keeping some small metadata M locally [5, 67]. After this initial upload, the client can challenge the server to check the integrity of her

data, and the server returns a proof. If the proof verifies, the client rests assured that her data is still intact (with some high probability). Otherwise, the client has caught the server cheating, and hence needs to resolve this issue officially at a court.

Public verifiability does not imply arbitration by a Judge: To provide a concrete example, we shall build upon the publicly-verifiable PDP scheme by Ateniese et al. [5]. Unfortunately, even though their public verifiability extension can possibly be used among collaborators reaching the same data, it cannot be used officially. This is because the client publishes a secret value v, which is the key for the pseudo-random function used, after uploading the file. At this point, a malicious client may publish an incorrect value, and hence all public verifications will fail even if the server is honest. Note that the main reason is that PDP-type schemes assume the client is honest, and thus do not consider the attack we just showed. Yet, this is a very realistic attack that must be addressed in practice. For example, a client may try to frame Amazon and obtain money in return. On the other hand, Amazon may like to offer such a provable service with possible warranty to bring enterprise customers. Once official arbitration is possible, it will be beneficial to both the service provider and the customers.

This framing issue can be addressed, in a generic manner, using a fair exchange protocol [15, 39, 3, 4, 8, 7, 52, 53, 57, 59, 10] and proper arbitration. Indeed, just a digital signature [45, 44] will be enough for the static storage case. To address this framing attack, we require that the server signs the proper metadata on top of this publicly-verifiable PDP scheme. In particular, the value v must be included as part of the metadata signed, along with other PDP metadata (i.e., the RSA group values N, g, e) [5]. Note that this metadata (call it $M = N, g, e, v$ here) now completely enables challenge verification by a third party.

The proposed modification to the publicly-verifiable PDP protocol [5] is the following: After the initial upload and the client made the v value public, the server checks for correctness of v[5] and sends the client his signature on the metadata $sign_S(M)$. If the signature does not verify with the N, g, e, v values known by the client, the client must assume that the upload failed and her data is not backed-up. Otherwise the protocol has succeeded, and the signature can be used for arbitration in case of a dispute.

We require only minimal additions to PDP-type static protocols: namely, the verification and signing of the metadata by the server, as described above. Now, if, during the challenge-response part of the PDP protocol, the proof fails to verify, then the client must contact the Judge for arbitration. The client presents the metadata M, together with the server's signature on it ($sign_S(M)$) as evidence. The Judge decides as follows:

- If the supporting evidence contains a valid signature on the metadata M, which is verified using the public key of server in a trusted public-key infrastructure (PKI), then the Judge requests an integrity proof from the server. The server needs to run the PDP response algorithm and send the resulting proof as evidence. If the proof does not verify using the metadata M, then the Judge outputs B, deciding that the server did not keep the client's data intact, and punishes the server accordingly.

- Otherwise, if the signature in the supporting evidence does not verify, or the PDP proof verifies, then the Judge outputs C and possibly punishes the client for taking her time for an invalid claim.

[5] We do not have enough space to re-present the details of the publicly-verifiable PDP protocol [5]. But, the server can check to see if $T_{i,m}^e \stackrel{?}{=} h(w_v(i))g^m \mod N$ using only the values known to him. Similarly, in publicly-verifiable Compact POR [67], the server must verify the signature on t_0 and check if $e(\sigma_i, g) \stackrel{?}{=} e(H(name||i)\Pi_{j=1}^s u_j^{m_{ij}}, v)$ using only the values known to him. Since this is a one-time operation, it is acceptable for the server to check every block in both protocols.

In general, our protocol can be applied on top of *any* PDP scheme that uses no secret keys. This is an intuitive requirement that the client need not keep secrets so that a third party can also perform challenge verification properly. For example, the publicly-verifiable PDP [5] and the publicly-verifiable Compact POR [67] use no secrets after setup. However, although no scheme known by us uses them, it may be possible to perform third-party verification even when the client keeps some secret values by using costly primitives such as commitments [65, 38, 30] and zero-knowledge proofs [43, 42, 11], as well as secure multi-party computation schemes [71, 41, 14, 25, 26]. Even though the simplicity of our scheme is astonishing, this issue has not been discussed before, and its difference from public verifiability was not known to be shown. Lastly, we assume a PKI contains the server's signature verification key. Since the server is a well-known entity (e.g., Amazon), this is a realistic argument. For cases where PKI is not available, we present a solution in Section 4.3.

Theorem 3.1. *If the digital signature scheme used is secure against existential forgery [44], and the PDP scheme used is secure [5], then the scheme above provides secure official arbitration for PDP.*

Proof. **Invalid-Claim case:** A claim is invalid under two conditions (note that the server is the Blamed):
(1) There is no storage agreement between the client and the server, hence the server did not sign M. In this case, the Judge will output B with only negligible probability since otherwise the Supporting Responder –the client– who outputs $sign_S(M)$ as evidence can be used to break the security of the signature scheme using a straightforward reduction.
(2) There is a storage agreement between the server and the client, but the server did not violate the terms (i.e., has not corrupted the client's data). In this case, the Falsifying Responder –the server– can return a valid PDP proof due to the *correctness* of the PDP scheme used. Thus, again, the Judge will not decide B.
Valid-Claim case: If the claim is valid, this means the server has signed M, and corrupted the client's data. For the sake of contradiction, assume the Judge still decides that the server is innocent with non-negligible probability. This is only possible if the server can still output a verifying proof to the Judge's challenge with non-negligible probability. Then we can use the server to break the security of the underlying PDP scheme [5].[6]

The security parameter k will be the security parameter of the signature scheme used for the invalid-claim case, and the security parameter of the underlying PDP scheme for the valid-claim case. Actually, the scheme is an **invalid-claim-punishing official arbitration scheme** since the decision may be C in case of an invalid claim. □

4 Arbitration for Dynamic Storage

In this section, we shall present two schemes. The first scheme will be a generalization of the idea for static schemes, but will be inefficient. In the second scheme, we will make use of the fact that a Judge is available to resolve the disputes between the client and the server, thereby providing a much more efficient scheme. We will compare these two approaches in Section 5. Throughout, we shall refer to the client as C and the server as S.

As in PDP-type schemes, DPDP schemes also have an initial protocol between the client and the server where the client uploads her (encoded) data F to the server, while keeping

[6] We need to note that PDP schemes provide a security level that requires the malicious server to succeed with a very small probability p, which may not necessarily be negligible in the mathematical sense. For the best explanation and result, we direct the reader to Compact POR [67] and Section 5.9.2 of [49, 50].

some metadata M [37]. Later, with each update to the client's data, there is another protocol performed by the client and the server using PrepareUpdate, PerformUpdate, VerifyUpdate algorithms. The integrity verification follows the Challenge, Prove, Verify protocols, where the Prove protocol is run by the server, and Challenge, Verify protocols are run by the client (or the Judge in our case), as defined by Erway et al. [37]. For the sake of completeness, we include these algorithms in the Appendix. To use our solutions on top of any DPDP scheme[7], we require the following outlined changes:

- An (optimistic) agreement protocol (i.e., fair signature exchange protocol) must be performed during the initial upload of the (encoded) file. If this agreement fails, the client should assume that the upload failed.
- An (optimistic) *dynamic* agreement protocol must be performed during each data update (this may or may not be done via a fair signature exchange protocol, as we will discuss). If there is a problem with the agreement, the client must contact the Arbiter. If an agreement during an update request failed, the client must assume that the file is not updated, and –in the best case– it is in the shape defined by the last successful agreement.
- If, at any point in the DPDP scheme the proof sent by the server fails to verify, the client contacts the Judge for arbitration.

We do not require any changes to the core DPDP protocol. We only provide additional message exchanges and dispute resolution strategies on top of any DPDP scheme[7]. Therefore, we envision that our official arbitration solution will be easy to deploy and use, compatible with any underlying scheme.

4.1 Inefficient Scheme

Following the outline above, the most basic idea would be to use an optimistic fair exchange protocol (e.g., [3]) whenever we require an agreement between the client and the server. Both parties will be holding counter values, initialized to some agreed-upon value (e.g., zero). Furthermore, the counters are assumed to be monotonic, and all the information about the counters (i.e., initial value, increments/decrements) are known to the Judge. For the sake of simplicity, in our presentation, we shall assume that the counters are initialized to 0, and are monotonically-increasing with uniform increments of 1. We will denote the counter value kept at the client using ctr_C, and that kept at the server using ctr_S.

In the dynamic scenario, with each update that the client performs on her data, the metadata also changes accordingly. Thus, a new agreement on this metadata must be made, by running a fair contract signing protocol (e.g., [3]). With each update, the client and the server exchange the following values fairly (after incrementing the counters): $sign_C(ctr_C)$ and $sign_S(M, ctr_S)$ where $ctr_C = ctr_S$. If the fair exchange completes successfully, then the client now has the signature of the server $sign_S(M, ctr_S)$ on the metadata associated with the latest update/counter, and the server has the signature of the client $sign_C(ctr_C)$ on the latest counter.[8]

Now, if the proof does not verify, the client contacts the Judge, and presents the latest metadata M, counter ctr_C, and the server's signature $sign_S(M, ctr_S)$. The Judge then performs the following actions:

[7] Any DPDP scheme where the updates, challenges, and verification requires no secrets.

[8] The client's signature need not include the metadata again; it is enough if the server signs it. Moreover, in practice, these signatures must include some more details about the contract between the client and the server, e.g., some identifier.

1. The Judge checks the signature of the server. Since the server is expected to be a well-known entity, we may assume his public key is in some trusted PKI. If the signature is invalid, the Judge may punish the client for taking her time for an invalid claim. If the signature is valid, the Judge proceeds.

2. The Judge requests the client's latest signature from the server. The server replies with $sign_C(ctr_C)$. For now, assume the client's public key is also in the PKI (we will relax this in our Efficient Scheme). If this signature does not verify, the Judge may punish the server; otherwise she proceeds.

3. If the counters match ($ctr_C = ctr_S$):

   (a) The Judge runs the Challenge algorithm of the underlying DPDP protocol, and sends the challenge to the server. The server runs the Prove algorithm and returns the proof.

   (b) The Judge checks the proof sent by the server against the metadata M that was signed by the server using the Verify algorithm of the underlying DPDP protocol. If the proof verifies, then the data is intact, and the Judge may punish the client for taking her time for an invalid claim.

   (c) If the verification fails, then the server is guilty, and is punished accordingly.

4. If there is a mismatch in counters, we have the following cases:

   (a) $ctr_S > ctr_C$: The server is cheating by trying to replay an old signature from the client. Punish the server.

   (b) $ctr_C > ctr_S$: The client is cheating by trying to replay an old signature from the server. Punish the client. Note that if the fair signature exchange failed, then the client should have contacted the fair exchange Arbiter for resolution first, not the Judge, since this is not necessarily a storage problem. Normally, with a failed fair exchange of signatures (even after trying to resolve with the Arbiter), the client should not assume the server holds the latest updated version.

Theorem 4.1. *If the digital signature scheme used is secure against existential forgery [44], and the signatures are exchanged using an (optimistic) fair exchange protocol [3], and the DPDP scheme used is secure [37], then the scheme above provides secure official arbitration for DPDP.*

Proof. If a fair signature exchange failed, the participants must contact the Arbiter of the fair exchange protocol to resolve their issue first, and assume the latest update did not take place.

Assuming the signature exchange completed fairly, both parties have signatures of each other, which means ideally $ctr_B = ctr_C$ in the given evidence. If instead $ctr_B \neq ctr_C$, then one of the parties is trying to use an older version. The smaller counter value will denote an older version (assuming a *monotonically-increasing counter*), and the corresponding party will be punished as described in the `decide` algorithm.

If, on the other hand, a malicious party tries to trick the Judge's decision algorithm to punish an honest party, then (s)he needs to generate a signature of the honest party on a counter value that is larger than any one of the signatures the malicious party has received. If (s)he succeeds with non-negligible probability, then (s)he can be used to break the security of the signature scheme in a reduction. Thus, we can conclude that the Judge will decide wrongly only with negligible probability for the counter mismatch case, as required. For the case where the counters match, we have two options:

Invalid-Claim case: If the claim is invalid (the server is still keeping the data intact), then,

through the correctness of the DPDP protocol, he can answer challenges and update requests using proofs that verify with the latest metadata. Therefore, for the client to be able to frame an honest server with an invalid claim, she must produce a fake signature with an incorrect metadata. But such a client may be used to break the underlying signature scheme in a reduction. Thus the Judge will not decide B (that the server is guilty), except with negligible probability of the client forging a signature.

Valid-Claim case: If the server has corrupted the data, then the server must break the security of the underlying DPDP scheme to be able to still produce a verifying evidence.[9] Thus, with high probability, the Judge will decide that the server is guilty.

We hope the reader is convinced that reductions to the security of the DPDP scheme or the signature scheme can easily be shown, but we do not provide complete reductions for the sake of space. The security parameter k depends on the security parameters of the underlying DPDP scheme and the signature scheme used. Moreover, the scheme can be considered an **invalid-claim-punishing official arbitration scheme** since the decision algorithm outputs C in case of an invalid claim. □

4.2 Efficient Scheme

Unfortunately, fair exchange protocols are costly since all known state-of-the-art optimistic fair exchange protocols [4, 3, 52, 53, 59, 10, 7] employ a costly primitive called *verifiable escrow* (or verifiable encryption) [24, 20]. We will observe this performance issue in Section 5.

Remember that optimistic fair exchange protocols employ a trusted third party (TTP), called the *Arbiter*. Realizing that we already have TTP, the Judge, we integrate this fair exchange together with the arbitration protocol, achieving much better efficiency. Figure 1 depicts the Efficient Scheme applied on top of the DPDP-type protocols.

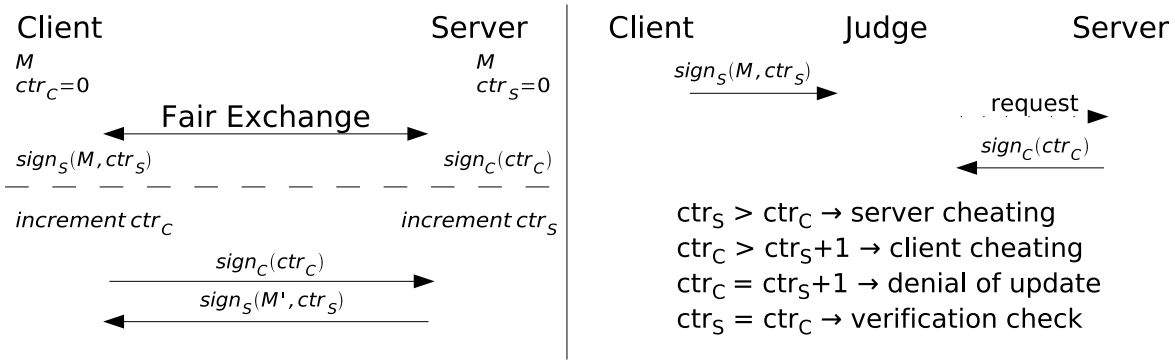

Figure 1: Left side shows our novel optimistic dynamic agreement protocol, where the part above the dashed line is the one time setup and the part below it is performed with every update. Right side summarizes the arbitration procedure in case of a dispute.

The idea is that a fair signature exchange is performed only once, when the client first uploads her data to the server. Then, for each update, after incrementing the counters, the signatures are indeed exchanged unfairly. Below we present the details of the dispute resolution for the Efficient Scheme as it applies on top of any DPDP protocol.[7] In this case, the client may be claiming two things:

[9] DPDP schemes have a probability $1 - p$ of detecting a cheating server. When p is negligible, the server cannot produce a verifying evidence with non-negligible probability.

1: Claim 1: The client claims that the server has corrupted the data (i.e., proof did not verify).

2: Claim 2: The client claims that the server denies updating the data. In this case, let F' denote the update the client is trying to perform, and M be the metadata of the last successful update.

The first two steps of the Judge's duties (requesting and verifying signatures) is the same as in the Inefficient Scheme. Furthermore, in case of Claim 1, if the counters match, the Judge's duties remain the same again. We will now define what the Judge does in case of Claim 2, if the counters match (assume ctr_C and ctr_S denote the counter values for the last successful update):

1. The Judge runs the PrepareUpdate algorithm of the underlying DPDP protocol using the data F' provided by the client.[10] Then, she sends this update request to the server. The server runs the PerformUpdate algorithm for the new data F' and returns the resulting new metadata M' as well as a proof-of-update (see Appendix).

2. The Judge checks the proof sent by the server against the old metadata M that was signed by the server, together with the new metadata $tagT'$ and the proof-of-update using the VerifyUpdate algorithm. If the proof fails, the Judge punishes the server.

3. The Judge requests new signatures from both the client and the server. The client responds with her signature on the incremented counter $sign_C(ctr'_C)$ $(ctr'_C = ctr_C + 1)$, and the server responds with his signature on the new metadata and incremented counter $sign_S(M', ctr'_S)$ $(ctr'_S = ctr_S + 1)$.

4. If both signatures verify and the counters match, then the Judge forwards the client's signature to the server, and the server's signature to the client. Otherwise, if there is a problem with the signatures, the Judge may punish that party for sending a faulty signature.

Now that the update has been successfully performed, and the new signatures are fairly exchanged, Claim 2 is resolved, and the client and the server may continue normally. Until now, we haven't considered the counter mismatch situation. This is explained next (for both claims):

1. $ctr_S > ctr_C$: The server is cheating by trying to replay an old signature from the client. Punish the server.

2. $ctr_C > ctr_S + 1$: The client is cheating by trying to replay an old signature from the server. Punish the client.

3. $ctr_C = ctr_S + 1$: There are two possibilities in this case. (1) The server did not send her signature response for the latest update (willingly or due to failures). This only holds for Claim 2, and can be resolved in the same way as explained above for the counter match situation. (2) Everything went well but the client is trying to overload the Judge. This can be limited by making the client pay for such requests after some number of free resolutions. Normally, the client should stop working with the server if he faces problems several times. This issue will be further explored in detail.

[10] We require that the PrepareUpdate algorithm would not use any secret keys, since otherwise the client needs to reveal those to the Judge. The PrepareUpdate algorithm requires additional inputs, which are hidden for the sake of presentation, but will be provided by the client in practice.

Note that *for the denial-of-update case* (Claim 2), *the server is given a second chance.* This is due to two reasons: (1) it is not possible for the client to prove that her update was denied, and (2) our first goal is functionality rather than punishment. Therefore, in a denial-of-update situation (possibly due to a service failure on the server side), if the server properly updates with the Judge's request, then the client and the server may proceed with further rounds normally.

Moreover, it is possible that *the Judge performs this update incognito*: The Judge need not identify himself as the Judge; instead he can impersonate the client (with the client's help, possibly using anonymous identification/credential/authentication/communication techniques [18, 66, 21, 22, 55, 32] that can even hide the IP address). Note that the messages that would have been sent by the client and the messages the Judge sends are completely identical, and are exactly as defined by the underlying DPDP protocol. This way, we prevent the server from acting maliciously against the client and behaving nicely against the Judge. We further refer the reader to the *zero-knowledge auditing* property defined by Shah et al. [68].

Furthermore, naturally, we assumed that it is in the contract between the client and the server that the server should perform updates requested by the client. This may not always hold; *there may be limits on the updates as a contract deadline time, or based on the number of updates* (e.g., "updates can be performed until 01.01.2015"). In such a case, the Judge shall check the expiry date of the contract between the server and the client, as well as the counters (against the limit on the number of updates).

Besides, it is also possible that the claim is a *denial-of-retrieval*, in which case the client claims that the server denies sending the client's data. The resolution will be very similar to the challenge-response case: the Judge retrieves the data on behalf of the client (as well as proof of correctness) using the underlying DPDP scheme, and forwards this data to the client after verifying the proof. If the server does not cooperate, the Judge punishes him.

Lastly, consider the case where no cheater can be identified and the output is N. If the server acts maliciously several times (or even keeps failing frequently), an honest client is expected to stop working with that server. Hence, the Judge may start charging for the arbitration process between a particular client and server after several free arbitrations. Combined with the discussion above that the server cannot distinguish the Judge and the Client, the server cannot force the client to pay for unnecessary arbitrations. In Section 4.3, we present a scheme where all the payments in the system can be fully automated.

Theorem 4.2. *If the digital signature scheme used is secure against existential forgery [44], and the DPDP scheme used is secure [37], then the scheme above provides secure official arbitration for DPDP.*

Proof. Assuming the initial fair exchange protocol has completed successfully, both parties have the initial signatures of each other: the client has $sign_S(M, 0)$, and the server has $sign_C(0)$. The proof of dispute resolution immediately after this initial exchange is exactly the same as the proof for the Inefficient Scheme; therefore we proceed by including at least one round of the new optimistic dynamic agreement protocol we described ("unfair exchange of signatures").

At the beginning of each round, the counters of the client and the server have the same value (remember that the previous exchange must have completed correctly, since otherwise the parties must perform dispute resolution instead of going through another exchange). After the increment of the counters, both parties have each other's signature with the previous counter value, obtained after the preceding successful exchange: the client has $sign_S(M, ctr_S - 1)$, and the server has $sign_C(ctr_C - 1)$, where $ctr_S = ctr_C$ denote the current counter values. After the client sends her signature to the server, she still has $sign_S(M, ctr_S - 1)$, whereas the server now has $sign_C(ctr_C)$.

If the server does not respond with his signature now (or sends an invalid signature), then the

client sends a denial-of-update claim to the Judge. Since we have $ctr_C = ctr_S + 1$ in this case, the Judge performs the update on behalf of the client. If the server behaves maliciously during this update, the Judge punishes the server. Due to the security of the underlying DPDP scheme, the Judge can verify that the new metadata sent by the server corresponds to the updated data (otherwise, if the server manages to send a valid proof without correctly updating the metadata, he can be used to break the DPDP scheme using a reduction). Since the Judge expects that the client would stop working with this malicious server, the client shall be punished if this claim is repeated frequently.

Instead, if the server sends his correct signature in return to the client's, this means both parties have each other's latest signature: the client has $sign_S(M', ctr_S)$, and the client has $sign_C(ctr_C)$ with $ctr_S = ctr_C$. At this point, the protocol becomes the same as the Inefficient Scheme for the purposes of dispute-resolution, and therefore the same proof applies here. This means the Efficient Scheme is also an **invalid-claim-punishing official arbitration scheme**. □

4.3 Arbitration with Automated Payments

We now extend our arbitration protocol for DPDP-type systems to include several possible payment types in an automated manner (through use of electronic checks [28] or electronic cash [27]). **The payments we consider are of the following types**:

1. *The client pays the server for service* (e.g., storage). This payment is done during the initial upload of the data, but it can be repeated after a pre-defined time or number of updates.

2. *The server pays the client in case of failure to provide service* (e.g., data corruption). This is essentially a *warranty* provided by the server.

3. *The Judge gets paid for her work by the cheating party.* We are assuming the Judge has (official) authoritative power over both parties.

We will refer to our new automated-payment official arbitration protocol for DPDP as the *Payment Scheme* (which is depicted in Figure 2).

At first, as in many real scenarios, the server sends a contract to the client. The contract specifies the details of the agreement between the client and the server, and information about the server such as his IP/DNS address, including his public key pk_S (thus **we no longer require a trusted PKI**). For example, the contract can specify that the server will keep the client's files intact, will perform updates as requested by the client, and will not perform denial of service. If the client is not happy with the contract, she aborts the protocol.

If the client is happy with the contract, she can go ahead and send her file for storage at the server (through the underlying DPDP system). The server then computes the DPDP metadata M, and the proof P that the metadata corresponds to the file, together with her signature on the proof and the initial counter value $sign_S(P, ctr_S)$. As in our Efficient Scheme, all signatures in our payment protocol will also include a counter kept at both parties, and initialized to 0. In case the metadata is randomized and cannot be computed by both parties in the same manner, a pseudorandom seed used by the randomized algorithm can be included within the signature.

Next, the client picks a random public key pk_C for her signature. This ensures client **privacy**, since her public keys with every server (or more precisely, with every contract) may be different, providing **unlinkability** through public keys, and again **removes the need for a trusted PKI**. She then prepares a verifiable escrowof her payment v =

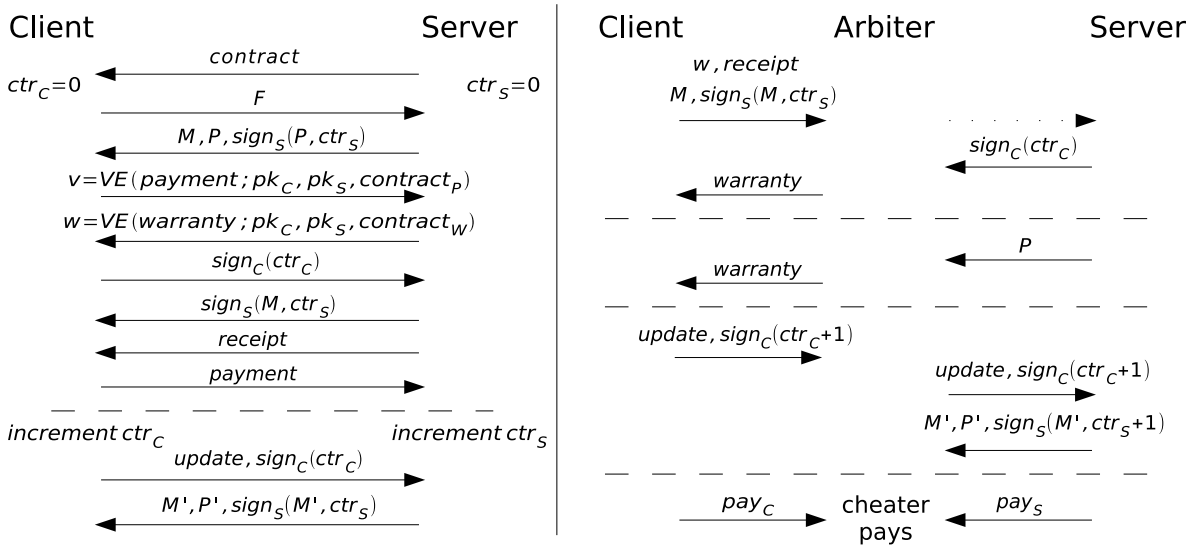

Figure 2: Left side above the dashed line is the setup phase of the automated-payment official arbitration protocol for DPDP, whereas below the dashed line shows a regular update phase. Right side shows dispute resolution by the Judge for various claims and scenarios.

$VE_{Arb}(payment; pk_C, pk_S, contract_P)$, labeled using the public keys of the client pk_C and the server pk_S, and the payment contract. The payment contract specifies the original contract, the metadata M, the signatures to be exchanged, the *receipt* to be given by the server in return for the payment, a timeout value for the fair exchange, an optional warranty clause specifying the verifiable escrow of a warranty check we will show below, and possibly some additional details. Note that if the payment is made using e-cash [27], then again **anonymity** of the client is preserved. If there is anything wrong with the verifiable escrow (i.e., it does not verify or values in the label are incorrect) then the server aborts.

Optionally, the server can provide the client with a warranty check at this point. This is a business decision, where the server may provide this extra service to bring in more customers, especially at the enterprise level. If this is desired, the server sends a verifiable escrow of the warranty $w = VE_{Arb}(warranty; pk_C, pk_S, contract_W)$ again with the public keys specified in the label, together with a warranty contract. The warranty contract describes that the warranty check should be decrypted if the server fails to observe the obligations in the contract (e.g., corrupts the client's data). If there is anything wrong with the warranty, the client aborts.

At this point, if the protocol is not aborted by any party yet, the parties start exchanging the first signatures (as in the Efficient Scheme). Namely, the client sends her signature $sign_C(ctr_C)$ on the counter (initialized to 0), and the server responds with his signature $sign_S(M, ctr_S)$ on the metadata and the counter (initialized to 0).

Once the signature exchange is done, the server sends the *receipt*[11] which includes the original contract, with terms describing any limits on the number or time of updates (e.g., "updates can be performed until 01.01.2015" or "a total of 1000 updates can be performed"), and the public keys of the server and the client. For a possible dispute resolution, the receipt is sent to the Arbiter by the client; the dispute is baseless without any receipt. If the receipt is correctly formed (i.e., the terms were the ones described in the payment contract, the public keys are the same as the ones agreed beforehand, and the time is current –with loose synchronization [52, 53]–), then the client sends the payment to the server, ending the setup phase.

[11]The *receipt* is like a real receipt showing the details of the purchase, signed by the server.

If the server did not receive the payment after giving the receipt, then he contacts the Arbiter, providing the payment verifiable escrow v, the metadata M, the client's signature $sign_C(ctr_C)$ on the initial counter, his signature $sign_S(M, ctr_S)$ on the metadata and the initial counter, the DPDP proof P, and the *receipt* (i.e., all the messages that an honest server would have provided to a client). The Arbiter checks the signatures (using the public keys in the verifiable escrow's label), verifies the proof using the metadata[12], and compares the payment contract with the receipt. If everything is fine, the Arbiter decrypts the verifiable escrow and hands over the payment to the server. The Arbiter stores all the values received from the server, at least until after the timeout. Just as in a timeout-based fair exchange protocol [3, 7, 53], the Arbiter will not honor requests from the server after the timeout.

If the client has not received the server's signature, or the receipt, she waits until the timeout of the fair exchange (in the worst case) and then contacts the Arbiter. If, before the timeout, the server has contacted the Arbiter, and received the payment by providing the metadata, his signature, the proof, and the receipt, then the Arbiter forwards these values to the client.[13] Note that, any dispute up to now is resolved by the Arbiter (who may or may not be the same authority as the Judge), since it is not directly related to the integrity of the stored data, but rather an initial agreement between the client and the server on the terms of storage.

At the end, the client only needs to keep the signature $sign_S(M, ctr_S)$, the *receipt*, and possibly the warranty w, on top of the metadata for the underlying DPDP protocol M. The server only needs the store the signature $sign_C(ctr_C)$ on top of the file F and other necessary information for DPDP. Both parties need to store their counters. Starting at the end of the payment phase, any third party can verify the server's (dis)honesty (but cannot punish unless she is the Judge).

Now that the initial agreement between the client and the server has been completed, we continue by using our Efficient Scheme for each update. The Judge's duty is exactly the same as in the Efficient Scheme (except that the client must provide the receipt, showing that she deserves the service), with the additional payment-related actions as follows:

- If the server is found guilty, then the Judge decrypts the warranty escrow and provides the payment inside to the client.
- If claims between the same two participants (identified by their public keys or the receipt) have been repeated many times, the Judge may choose to charge the participants (see more details in the Appendix).

Theorem 4.3. *If the digital signature scheme used is secure against existential forgery [44], the verifiable escrow scheme used is secure against chosen-plaintext attacks, the payment (e.g., e-cash) scheme used is unforgeable, and the DPDP scheme used is secure [37], then the scheme above provides secure official arbitration for DPDP.*

Proof. We will prove that at the end of the setup phase, the client obtains the server' signature $sign_S(M, ctr_S)$, the *receipt*, and possibly the warranty w, and the server obtains the client's signature $sign_C(ctr_C)$ (or the protocol is aborted and no storage agreement took place).

First of all, note that if one party does not send a message (or sends an invalid message), the protocol is either aborted, or a resolution with the Arbiter is required. The server cannot resolve before obtaining a valid signature $sign_C(ctr_C)$ of the client, since during the dispute resolution process this signature needs to be presented to the Arbiter. Therefore, before this

[12] In a real deployment, the Arbiter challenges the server for the proof using the underlying DPDP protocol.

[13] Obviously, the Arbiter must make sure that he is contacted by the client herself on the correct exchange. This association is possible via various methods [3, 53]. Besides, remember that the verifiable escrow contains the client's signature verification key, therefore this verification can easily be performed.

point in the protocol, the parties must abort rather than resolve (any other case will mean that the server has forged the client's signature, and thus the security of the signature scheme is broken). Yet, until this point, no party received any useful information: the server did not receive the payment (only the encrypted version that will not be decrypted without the client's signature); similarly the client only received an escrowed warranty, but not the receipt or the server's signature (likewise, if the client manages to forge the server's signature at this point, the security of the signature scheme is broken), which are required for the decryption of the warranty escrow. If the server or the client manages to obtain the payment or the warranty at this point, a reduction to the security of the verifiable escrow scheme may be done; using an adversarial server to break the security of the verifiable escrow scheme used for the payment, or an adversarial client to break the security of the verifiable escrow scheme used for the warranty.

If anything goes wrong after the server obtains the signature $sign_C(ctr_C)$ of the client (i.e., the server did not send his signature, or sent a non-verifying signature, or did not send the correct receipt, or the client did not send the payment), then Arbiter must be involved for resolution. The server needs to contact the Arbiter before the timeout specified in v to be able to get his payment. After the timeout, the Arbiter will not honor his request. During the resolution, the server must prove to the Arbiter that he is acting honestly by providing all messages that should have been sent to the client according to the protocol specification. These messages include his signature on metadata and initial counter, together with its verifying proof, and the receipt, according to the specifications in the contract in the label of v. If the server sends a fake signature, an adversarial reduction to the security of the signature scheme is possible. If the server manages to send a fake proof, then he can be used as an adversary to break the underlying DPDP scheme. Therefore, the server must send correct values to the Arbiter to obtain the client's payment.

The client can contact the Arbiter after the timeout and obtain any missing messages (in particular the signature of the server on the metadata, and the receipt), if the server resolved with the Arbiter and obtained the payment previously. Remember that the Arbiter has verified these values, and thus they must be the correct values. If the client realizes that the server has not contacted the Arbiter before the timeout, than she may assume that the contract failed, and the exchange will never be finished. Ideally, an honest client should stop working with a malicious server.

Now that we have showed either the protocol is completely aborted, or everything went well (or is resolved). Specifically, the client obtained the server' signature $sign_S(M, ctr_S)$, the *receipt*, and possibly the warranty w, and the server obtained the client's signature $sign_C(ctr_C)$. At this point, the rest of the proof is the same as the Efficient Scheme, since the Judge's duty does not change. □

5 Performance Evaluation

In this section, we compare the Inefficient Scheme and the Efficient Scheme, since these are the only available general-purpose official arbitration schemes for dynamic cloud storage. The main novelty and advantage comes from the fact that there is already a trusted third party, and the Judge may also be employed for resolution of "unfair" exchanges in some sense. Thus, instead of performing a full fair exchange, it is possible to employ just a regular signature exchange, and resolve optimistically (only in case of disputes).

As discussed before, fair exchange of signatures is a costly operation. Known optimistic fair exchange schemes employ a verifiable escrow [3, 52, 7, 57]; therefore we can conclude that the cost of a fair exchange is definitely more than the cost of a verifiable escrow. We employ Cashlib

cryptographic library [19, 58] implementation of the verifiable escrow [24]. Each verifiable escrow takes about 1 second and 25 KB, and each DSS signature [60] takes about 1 ms and 40 bytes on a machine with 3 GHz CPU and 4 GB RAM running Ubuntu 11.10. For the numbers below, instead of assuming a particular fair signature exchange protocol, we will use the verifiable escrow numbers above.

To provide a real workload, we consider application of our solutions on top of DPDP using the CVS repositories depicted in Table 1, taken from Erway et al. [37]. Using the method based on the Inefficient Scheme, every commit would necessitate a fair exchange. For roughly 25000 commits in Table 1, even only the verifiable escrow part of the cost of the fair exchange will correspond to **7 hours** and **610 MB** overhead. The cost of a full fair exchange is much more than that for just verifiable escrow, but even this cost is extreme. Yet, using our Efficient Scheme, including the initial fair exchange, providing public verifiability for all 25000 commits will require only **51 seconds** and **2 MB**. **Per update** (commit), our overhead is less than **0.1 KB** and **2 ms** (without counting the time spent by the underlying DPDP system).[14] Besides, **no growing history** need to be kept at either side. The **storage overhead** for the client and the server is **only tens to hundreds of bytes** each. This makes our approach the first ever practical, efficient, and scalable official arbitration protocol for any dynamic provable data possession scheme.[7] Some further analysis regarding security and privacy of our schemes can be found in the Appendix.

When we consider the *Payment Scheme with warranty*, there is no noticeable difference in performance numbers. The only real overhead is the warranty escrow, and since it is performed only once during the first contract phase, *per commit* network and computation overhead values do not change, as when distributed among thousands of updates, its overhead is close to zero.

	Rsync	**Samba**	**Tcl**
Total KBytes	8331 KB	18525 KB	44585 KB
# of commits	11413	27534	24054
Network overhead per commit	**80 bytes**	**80 bytes**	**80 bytes**
Computation overhead per commit	**2 ms**	**2 ms**	**2 ms**

Table 1: Overhead *per commit*, in total for both the client and the server.

6 General Official Arbitration

We now consider official arbitration for various general scenarios and not just cloud storage. The general idea remains the same as the arbitration for secure cloud storage. The main issue is that, secure cloud storage schemes had a very useful property: there is an underlying proof mechanism that achieves two very important goals:

- The underlying secure cloud storage proof constitutes evidence for or against the claim.
- For dynamic schemes, the underlying proof-of-update enables parties to update their agreement accordingly, while agreeing on the next value of the proper metadata.

Overall, these let completely automated arbitration to take place. Furthermore, remember that the counters let the Judge realize an order among resolution of claims in dynamic settings.

[14] Compare to 370 ms *overhead* of Wang et al. protocol [69] specific to their scheme.

6.1 Arbitration for Static Agreements

6.1.1 Basic Scheme

Consider a general *contract-resolution* scheme, where the underlying scenario is that the Claimer $\mathcal{P}_C$ and the Blamed $\mathcal{P}_B$ has successfully made an agreement beforehand, through an (optimistic) agreement protocol as defined in Appendix A.2 (essentially a fair contract signing protocol). The dispute stems from the fact that the Claimer claims that the Blamed has violated the terms of the contract. Below, we will describe each algorithm of the official arbitration protocol:

`claim`$() \to c$: The output c contains the agreed-upon contract m, as well as information about $\mathcal{P}_B$, and what part of the contract $\mathcal{P}_B$ violated.

`request`$(c) \to c'$: The output c' includes the original claim c, and requests the signature of the Blamed on the claimed contract m, as well as any evidence that $\mathcal{P}_B$ violated the contract.

`prove`$(c') \to e$: The output evidence e should include the signature $sign_{\mathcal{P}_B}(m)$ of the Blamed on the contract m, assuming that signature really exists. Furthermore, it must be proven that the $\mathcal{P}_B$ has violated the terms of the contract m.

`decide`$(c, e) \to \{O, C, B, N\}$: If the evidence contains a valid signature, which is verified using the public key of $\mathcal{P}_B$ in a trusted PKI, and there is evidence that the Blamed has violated the message terms, then the Judge outputs B and punishes the Blamed. Otherwise, the Judge may output C and punish the Claimer for taking her time for an invalid claim.

Note that the evidence needs to include some proof that the Blamed violated the contract.[15] This might be outside the scope of this digital arbitration process, and may involve real persons (e.g., witnesses) or items (e.g., broken windows, hair of the Blamed) or recordings (e.g., security camera footage, photos), etc. But for many realistic scenarios, a completely automated digital evidence-checking process may exist (such as what we had for arbitration for secure cloud storage). We shall look into another such case now, keeping in mind the general outline above.

6.1.2 E-commerce Scheme

Consider a scenario where, according to the agreement m, the Blamed should have paid x amount of money to the Claimer, and in return the Claimer should have given a signed receipt $sign_{\mathcal{P}_C}(x)$ back to the Blamed.[16] In real life, this is a very common scenario that we face everyday; happens all the time in e-commerce, and there are protocols to achieve this exchange *fairly* [3, 15, 10, 4, 52, 53, 59].
`claim`: The output contains the agreed-upon contract m, which contains information about x and $\mathcal{P}_B$, claiming that the $\mathcal{P}_B$ did not pay x.

`request`: The Judge requests from the Supporting Responder the signature of the Blamed on the claimed contract m, and requests from the Falsifying Responder the receipt showing that $\mathcal{P}_B$ paid x.

[15] This represents the real life case where we assume a person is innocent until proven guilty. A simple change may require that the Blamed must prove that he is innocent, through a falsifying evidence.

[16] Note that this signature is simplified for presentation purposes. It will actually contain more information about the exchange (possibly some unique identifier), since signing just an integer would be susceptible to replay attacks.

$\texttt{prove}_S$: The supporting evidence should include the signature $sign_{\mathcal{P}_B}(m)$ of the Blamed on the contract m, assuming that signature really exists.

$\texttt{prove}_F$: The falsifying evidence should include the signature $sign_{\mathcal{P}_C}(x)$ of the Claimer on the receipt of x, assuming that signature really exists.

`decide`: If $sign_{\mathcal{P}_B}(m)$ does verify but $sign_{\mathcal{P}_C}(x)$ does not verify, then B is output. Otherwise, the Judge may output C and possibly punish the Claimer for taking her time for an invalid claim.

Note that the above scenario actually contains **two related official arbitration executions**. In the first one, the claim is that the Blamed has agreed on a contract m, and the ***supporting evidence*** is his signature $sign_{\mathcal{P}_B}(m)$. In the second one, the claim is that the Blamed has violated the terms in the contract m (i.e., not paid x), and the ***falsifying evidence*** is the Claimer's signature (receipt) $sign_{\mathcal{P}_C}(x)$.

Theorem 6.1. *If the digital signature scheme used is secure against existential forgery [44][17], and an agreement protocol is used for exchanging signatures, then the E-commerce Scheme is a secure official arbitration scheme.*

Proof. **Invalid-Claim case:** Assume that the Judge decides that the Blamed is guilty with non-negligible probability, even though the claim is invalid. Then, either of the following two cases hold:
(1) The Blamed did not sign the contract m, but the Supporting Responder could output $sign_{\mathcal{P}_B}(m)$ with non-negligible probability. In this case, the Supporting Responder can be used as an adversary that breaks the underlying signature scheme using a straightforward reduction.
(2) The Blamed paid x but the Falsifying Responder could not output an evidence containing $sign_{\mathcal{P}_C}(x)$. By our assumption that agreement has been completed successfully, this could not have happened.[18]
Therefore, we can conclude that the Judge outputs B with only negligible probability when the claim is invalid.
Valid-Claim case: Assume that the Judge decides with non-negligible probability that the Blamed is innocent, even though the claim is valid. The claim being valid means that the Blamed signed m, but did not pay x. This means, the Supporting Responder can output an evidence containing $sign_{\mathcal{P}_B}(m)$. Thus, for the Judge to decide that the Blamed is innocent, the Falsifying Responder must output a valid signature $sign_{\mathcal{P}_C}(x)$. But if that is the case, the Falsifying Responder can be used to break the security of the signature scheme using a straightforward reduction. Hence, the Judge will decide B with high probability (i.e., $1 - neg(k)$).

The security parameter k is the security parameter of the signature scheme used. Furthermore, the scheme is an **invalid-claim-punishing official arbitration scheme** since the decision may be C in case of an invalid claim. □

6.2 Arbitration for Dynamic Agreements

The E-commerce Scheme is useful for static scenarios where the contract is useful for once. For example, in an e-commerce scenario, each time a customer buys an item, there is a new contract (just as the case in real world commerce). Furthermore, in terms of official arbitration, there

[17] Depending on the application, it may be necessary for the signature scheme to be secure against chosen-message attacks.

[18] Remember that we are assuming the exchange of x and $sign_{\mathcal{P}_C}(x)$ is performed fairly; its arbitration is the Arbiter's job, not the Judge's.

is no relation between multiple contracts, even those between the same customer and the same merchant. Now consider a general scenario, where the agreed-upon contract m is dynamic: It changes over time, and the contracts are related (e.g., via counters).

6.2.1 Inefficient General Scheme

The official arbitration phase of our dynamic protocol will be the same as the Basic Scheme (or the E-commerce Scheme), as long as the last valid contract m is used. This should be guaranteed through an (optimistic) dynamic agreement scheme.

Similar to our protocols for secure cloud storage, we require that both parties are holding counter values, initialized to some agreed-upon value (e.g., zero). Furthermore, the counters are assumed to be monotonic, and all the information about the counters (i.e., initial value, increments/decrements) are known to the Judge. For the sake of simplicity, in our presentation, we shall assume that the counters are initialized to 0, and are monotonically-increasing with uniform increments of 1. We will denote the counter value kept at the Claimer using ctr_C, and that kept at the Blamed using ctr_B.

In this scenario, with each update to the agreement, a new (optimistic) fair contract signing protocol must be executed by the Claimer and the Blamed, since this is the most straightforward (though costly) way of achieving dynamic agreement. In cases that the fair exchange fails (i.e., counters or contracts in the signatures do not match, or signatures do not verify), the participants must contact the Arbiter and resolve, before contacting the Judge. Up to this point, this follows the Inefficient Scheme for secure cloud storage.

The following modifications to the Basic Scheme are necessary to make it applicable to dynamic scenarios:

$\texttt{claim}() \rightarrow c$: The output c shall contain the latest version of the agreed-upon contract m, as well as information about $\mathcal{P}_B$, and what part of the contract the Blamed violated.

We present two responders again: the Supporting Responder and the Falsifying Responder.

$\texttt{prove}_S(c') \rightarrow e$: The supporting evidence e should include the signature $sign_{\mathcal{P}_B}(m, ctr_B)$ of the Blamed on the contract m and the latest counter ctr_B, assuming that signature exists. Furthermore, if possible, it must be proven that the the Blamed has violated the terms on the message m (see Section 6.1).

$\texttt{prove}_F(c') \rightarrow e$: The falsifying evidence e should include the signature $sign_{\mathcal{P}_C}(m', ctr_C)$ of the Claimer on the contract m' and the latest counter ctr_C, assuming that signature exists (ideally $m = m'$). Furthermore, if possible, it must be proven that the the Blamed has *not* violated the terms on the message m' (e.g., by providing $sign_{\mathcal{P}_C}(x)$ as in the E-commerce Scheme).

$\texttt{decide}(c, e) \rightarrow \{O, C, B, N\}$: The Jugde performs the following steps:

1. Verify the signature $sign_{\mathcal{P}_B}(m, ctr_B)$. If the verification fails, then the claim is invalid: output C and stop.
2. Verify the signature $sign_{\mathcal{P}_C}(m, ctr_C)$. If the verification fails, then fake evidence is provided: output B and stop.
3. $ctr_B > ctr_C \Rightarrow \mathcal{P}_B$ is cheating by trying to replay an old signature from $\mathcal{P}_C$. Output B and stop.

4. $ctr_C > ctr_B \Rightarrow \mathcal{P}_C$ is cheating by trying to replay an old signature from $\mathcal{P}_B$. Output C and stop.

5. $ctr_B = ctr_C \Rightarrow$ If $m \neq m'$ then direct both parties to the Arbiter (this is a fair exchange problem). Otherwise, this means $\mathcal{P}_C$ and $\mathcal{P}_B$ agree on the latest contract $m = m'$, and the Judge processes further evidence (digital or real) as in Section 6.1 and decides accordingly.

Theorem 6.2. *If the digital signature scheme used is secure against existential forgery [44][17], and the signatures are exchanged using an (optimistic) dynamic agreement protocol, then the Inefficient General Scheme is a secure official arbitration scheme.*

Proof. The proof for all the steps until counters match is identical to the first three paragraphs of the proof of the Inefficient Scheme. When both the signed counters and contract m match, the resolution proceeds as in Section 6.1, and thus that proof applies here directly. □

6.2.2 Efficient General Scheme

We are still considering the dynamic scenario, and will present our efficient scheme for general agreements. We employ our novel idea for an efficient optimistic dynamic agreement protocol, as in our Efficient Scheme for secure cloud storage. Thus, we still require that the initial contract-signing is performed through a fair signature exchange protocol, and therefore at the end both parties obtain each other's signatures. To be precise, the Claimer obtains the signature $sign_{\mathcal{P}_B}(m, 0)$ of the Blamed on the contract m, together with the initial counter value 0, and the Blamed obtains $sign_{\mathcal{P}_C}(m, 0)$ of the Claimer on the same contract m, with the counter value 0. *Once this fair exchange is completed successfully, fair exchange protocols or costly cryptographic primitives are* ***not*** *employed* in our scheme.

After the initial agreement of signatures, the following optimistic dynamic agreement protocol is executed whenever a new version of the contract m' is going to replace the old contract m:

1. Both parties increment their counters.

2. The Claimer sends his signature $sign_{\mathcal{P}_C}(m', ctr_C)$ on the new contract and the updated counter value to the Blamed.

3. If this signature is valid and the counter value and the contract is correct, then the Blamed responds with his signature $sign_{\mathcal{P}_B}(m', ctr_B)$ on the same new contract and the same counter value.

4. Upon receipt, the Claimer checks this signature, as well as the message and the counter.

During this process, if the Blamed did not receive a valid signature, then the protocol is simply aborted. Furthermore, if the Claimer sent a valid signature but did not receive a valid response, again, she should assume that the protocol is aborted and the new contract m' is not in place. Remember that the Claimer is expected to stop working with the Blamed if several such malicious failures occur.

On top of this efficient optimistic dynamic agreement protocol, we employ the following decision algorithm and call the resulting scheme the **Efficient General Scheme**.
`decide`$(c, e) \rightarrow \{O, C, B, N\}$: The Jugde performs the following steps:

1. Verify the signature $sign_{\mathcal{P}_B}(m, ctr_B)$. If the verification fails, then the claim is invalid: output C and stop.

2. Verify the signature $sign_{\mathcal{P}_C}(m, ctr_C)$. If the verification fails, then fake evidence is provided: output B and stop.

3. $ctr_B > ctr_C \Rightarrow \mathcal{P}_B$ is cheating by trying to replay an old signature from $\mathcal{P}_C$. Output B and stop.

4. $ctr_C > ctr_B + 1 \Rightarrow \mathcal{P}_C$ is cheating by trying to replay an old signature from $\mathcal{P}_B$. Output C and stop.

5. $ctr_C = ctr_B + 1 \Rightarrow$ There are two possibilities in this case. (1) $\mathcal{P}_B$ did not complete the dynamic agreement protocol (willingly or due to failures). (2) Everything went well but $\mathcal{P}_C$ is trying to overload the Judge. In both cases, the Claimer is expected to stop working with the Blamed if he faces problems several times. Thus, the Claimer may need to pay for such requests after some number of free resolutions. Output N and stop.

6. $ctr_B = ctr_C \Rightarrow$ If $m \neq m'$ then again rule 5 applies. Otherwise, this means the Claimer and the Blamed agreed on the latest contract $m = m'$, and the Judge processes further evidence (digital or real) as in Section 6.1 and decides accordingly.

Theorem 6.3. *If the digital signature scheme used is secure against existential forgery [44][17], the initial signatures are successfully exchanged using an (optimistic) agreement protocol (i.e., a fair contract-signing protocol [3]), and each time the contract changes the optimistic dynamic agreement protocol above is executed, then the Efficient General Scheme is a secure official arbitration scheme.*

Proof. Assuming the initial fair exchange protocol has completed successfully (otherwise there is no contract anyway), both parties have the initial signatures of each other: the Claimer has $sign_{\mathcal{P}_B}(m, 0)$, and the Blamed has $sign_{\mathcal{P}_C}(m, 0)$. The proof of dispute resolution immediately after this initial exchange is exactly the same as the proof for the Inefficient General Scheme; therefore we proceed by including at least one round of the new optimistic dynamic agreement protocol we described.

At the beginning of each round, the counters of the Claimer and the Blamed have the same value (remember that the previous exchange must have completed correctly, since otherwise they must assume the agreement is aborted, and should stop working together if this occurs repeatedly). After the increment of the counters, both parties have each other's signature with the previous counter value, obtained after the preceding successful exchange: the Claimer has $sign_{\mathcal{P}_B}(m, ctr_B - 1)$, and the Blamed has $sign_{\mathcal{P}_C}(m, ctr_C - 1)$, where $ctr_B = ctr_C$ denote the current counter values. Once the Claimer sends her signature to the Blamed on the updated contract m', she still has $sign_{\mathcal{P}_B}(m, ctr_B - 1)$, whereas the Blamed now has $sign_{\mathcal{P}_C}(m', ctr_C)$.

If the Blamed does not respond with his signature now (or sends an invalid signature), then the expectation is that the Claimer considers the agreement as aborted, and stops working with the Blamed. Since we have $ctr_C = ctr_B + 1$ in the signatures for this case, the Judge has the same expectation and does not perform arbitration. The main reason that no cheater can be identified in this situation is because on top of the possibility that the Blamed did not respond to the agreement, there is always the possibility that the Claimer is replaying an old signature. As usual, the participants shall be punished if this claim is repeated frequently.

Instead, if the Blamed sends his correct signature in return to the Claimer's, this means both parties have each other's latest signature: the Claimer has $sign_{\mathcal{P}_B}(m', ctr_B)$, and the Blamed has $sign_{\mathcal{P}_C}(m', ctr_C)$ with $ctr_S = ctr_C$. At this point, the protocol becomes the same as the Inefficient Scheme for the purposes of dispute-resolution, and therefore the same proof applies here. □

Note the important difference from the Efficient Scheme for secure cloud storage: The case $ctr_C = ctr_B + 1$ for secure cloud storage can be handled automatically by running the PrepareUpdate, PerformUpdate, VerifyUpdate mechanism of the underlying DPDP protocol. In our Efficient General Protocol, if the underlying agreement has the property that the next value m' of the contract follows a structure from the previous value m of it, and there is a proof that can show that m' is really the value of the contract that should follow m based on the claim c, then this method can also be employed to automate the general official arbitration process. For secure cloud storage, the server proves that the next metadata is M' if one performs the update F' in the client's claim on top of the data that corresponds to the metadata M.

Authenticated data structures present such an opportunity to use our Efficient General Scheme; they provide a summary of the data structure (essentially the metadata) that can be used for verification purposes. Then, when an update on the authenticated structure is requested, a new metadata is produced such that it can be proven that the new metadata does indeed correspond to the new data structure summary after the update is performed on top of the data structure whose summary was the old metadata. This means our Efficient General Scheme can be applied, in an automated manner, on top of any *dynamic authenticated data structure*, including authenticated skip lists [63], authenticated hash tables [64], and many other authenticated data structures [36, 62], where updates can be proven.

ACKNOWLEDGEMENTS

This work is supported by TÜBİTAK, the Scientific and Technological Research Council of Turkey. We also thank Anna Lysyanskaya, C. Chris Erway, and Charalampos Papamanthou for their useful discussions during the initial phases of this paper.

References

[1] N. Asokan, M. Schunter, and M. Waidner. Optimistic protocols for fair exchange. In *ACM CCS*, 1997.

[2] N. Asokan, V. Shoup, and M. Waidner. Optimistic fair exchange of digital signatures. In *EUROCRYPT*, 1998.

[3] N. Asokan, V. Shoup, and M. Waidner. Optimistic fair exchange of digital signatures. *IEEE Selected Areas in Communications*, 18:591–610, 2000.

[4] G. Ateniese. Efficient verifiable encryption (and fair exchange) of digital signatures. In *ACM CCS*, 1999.

[5] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song. Provable data possession at untrusted stores. In *ACM CCS*, 2007.

[6] G. Ateniese, S. Kamara, and J. Katz. Proofs of storage from homomorphic identification protocols. In *ASIACRYPT*, 2009.

[7] G. Avoine and S. Vaudenay. Optimistic fair exchange based on publicly verifiable secret sharing. In *ACISP*, 2004.

[8] F. Bao, R. Deng, and W. Mao. Efficient and practical fair exchange protocols with off-line ttp. In *IEEE Security and Privacy*, 1998.

[9] M. Belenkiy, M. Chase, C. Erway, J. Jannotti, A. Küpçü, and A. Lysyanskaya. Incentivizing outsourced computation. In *NetEcon*, 2008.

[10] M. Belenkiy, M. Chase, C. Erway, J. Jannotti, A. Küpçü, A. Lysyanskaya, and E. Rachlin. Making p2p accountable without losing privacy. In *ACM WPES*, 2007.

[11] M. Bellare and O. Goldreich. On defining proofs of knowledge. In *CRYPTO*, 1992.

[12] S. M. Bellovin and M. Merritt. Encrypted key exchange: Password-based protocols secure against dictionary attacks. In *IEEE Security and Privacy*, 1992.

[13] S. M. Bellovin and M. Merritt. Augmented encrypted key exchange: A password-based protocol secure against dictionary attacks and password file compromise. In *ACM CCS*, 1993.

[14] M. Ben-Or, R. Canetti, and O. Goldreich. Asynchronous secure computation. In *STOC*, 1993.

[15] M. Ben-Or, O. Goldreich, S. Micali, and R. L. Rivest. A fair protocol for signing contracts. *IEEE Transactions on Information Theory*, 36:40–46, 1990.

[16] K. D. Bowers, A. Juels, and A. Oprea. Hail: A high-availability and integrity layer for cloud storage. In *ACM CCS*, 2009.

[17] X. Boyen. Hpake: Password authentication secure against cross-site user impersonation. In *CANS*, 2009.

[18] S. Brands. *Rethinking Public Key Infrastructures and Digital Certificates: Building in Privacy.* MIT Press, 2000.

[19] Brownie cashlib cryptographic library. http://github.com/brownie/cashlib.

[20] J. Camenisch and I. Damgard. Verifiable encryption, group encryption, and their applications to group signatures and signature sharing schemes. In *ASIACRYPT*, 2000.

[21] J. Camenisch and A. Lysyanskaya. An efficient system for non-transferable anonymous credentials with optional anonymity revocation. In *EUROCRYPT*, 2001.

[22] J. Camenisch and A. Lysyanskaya. A signature scheme with efficient protocols. In *SCN*, 2002.

[23] J. Camenisch, A. Lysyanskaya, and M. Meyerovich. Endorsed e-cash. In *IEEE Security and Privacy*, 2007.

[24] J. Camenisch and V. Shoup. Practical verifiable encryption and decryption of discrete logarithms. In *CRYPTO*, 2003.

[25] R. Canetti, U. Feige, O. Goldreich, and M. Naor. Adaptively secure multi-party computation. In *STOC*, 1996.

[26] R. Canetti, Y. Lindell, R. Ostrovsky, and A. Sahai. Universally composable two-party and multi-party secure computation. In *STOC*, pages 494–503, 2002.

[27] D. Chaum. Blind signatures for untraceable payments. In *CRYPTO*, 1982.

[28] D. Chaum, B. den Boer, E. van Heyst, S. Mjlsnes, and A. Steenbeek. Efficient offline electronic checks (extended abstract). In *EUROCRYPT*, 1990.

[29] R. Curtmola, O. Khan, R. Burns, and G. Ateniese. Mr-pdp: Multiple-replica provable data possession. In *ICDCS*, 2008.

[30] I. Damgard and E. Fujisaki. A statistically-hiding integer commitment scheme based on groups with hidden order. In *ASIACRYPT*, 2002.

[31] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22:644–654, 1976.

[32] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *USENIX Security*, 2004.

[33] Y. Dodis, P. J. Lee, and D. H. Yum. Optimistic fair exchange in a multi-user setting. In *PKC*, 2007.

[34] Y. Dodis, S. Vadhan, and D. Wichs. Proofs of retrievability via hardness amplification. In *TCC*, 2009.

[35] J. R. Douceur. The sybil attack. In *IPTPS*, 2002.

[36] C. Dwork, M. Naor, G. N. Rothblum, and V. Vaikuntanathan. How efficient can memory checking be? In *TCC*, 2009.

[37] C. Erway, A. Küpçü, C. Papamanthou, and R. Tamassia. Dynamic provable data possession. In *ACM CCS*, 2009.

[38] E. Fujisaki and T. Okamoto. Statistical zero knowledge protocols to prove modular polynomial relations. In *CRYPTO*, 1997.

[39] J. A. Garay, M. Jakobsson, and P. MacKenzie. Abuse-free optimistic contract signing. In *CRYPTO*, 1999.

[40] C. Gentry, P. MacKenzie, and Z. Ramzan. A method for making password-based key exchange resilient to server compromise. In *CRYPTO*, 2006.

[41] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *STOC*, 1987.

[42] O. Goldreich, S. Micali, and A. Wigderson. Proofs that yield nothing but their validity or all languages in np have zero-knowledge proof systems. *Journal of ACM*, 38:728, 1991.

[43] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18:208, 1989.

[44] S. Goldwasser, S. Micali, and R. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17:281–308, Apr. 1988.

[45] S. Goldwasser, S. Micali, and A. Yao. Strong signature schemes. In *STOC*, 1983.

[46] D. P. Jablon and W. Ma. Strong password-only authenticated key exchange. *ACM Computer Communications Review*, 26:5–26, 1996.

[47] A. Juels and B. S. Kaliski. PORs: Proofs of retrievability for large files. In *ACM CCS*, 2007.

[48] S. Kamara and K. Lauter. Cryptographic cloud storage. In *FC*, 2010.

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

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

[51] A. Küpçü and A. Lysyanskaya. Optimistic fair exchange with multiple arbiters. In *ESORICS*, 2010.

[52] A. Küpçü and A. Lysyanskaya. Usable optimistic fair exchange. In *CT-RSA*, 2010.

[53] A. Küpçü and A. Lysyanskaya. Usable optimistic fair exchange. *Computer Networks*, 56:50–63, 2012.

[54] A. Y. Lindell. Legally-enforceable fairness in secure two-party computation. In *CT-RSA*, 2008.

[55] Y. Lindell. Anonymous authentication. *Journal of Privacy and Confidentiality*, 2:35–63, 2010.

[56] P. D. MacKenzie, T. Shrimpton, and M. Jakobsson. Threshold password-authenticated key exchange. In *CRYPTO*, 2002.

[57] O. Markowitch and S. Saeednia. Optimistic fair exchange with transparent signature recovery. In *FC*, 2001.

[58] S. Meiklejohn, C. Erway, A. Küpçü, T. Hinkle, and A. Lysyanskaya. Zkpdl: Enabling efficient implementation of zero-knowledge proofs and electronic cash. In *USENIX Security*, 2010.

[59] S. Micali. Simple and fast optimistic protocols for fair electronic exchange. In *PODC*, 2003.

[60] NIST. Digital signature standard (dss). *FIPS PUB 186-3*, 2009.

[61] H. Pagnia and F. C. Gartner. On the impossibility of fair exchange without a trusted third party. Technical report, Darmstadt University of Technology TUD-BS-1999-02, 1999.

[62] C. Papamanthou. *Cryptography for Efficiency: New Directions in Authenticated Data Structures*. PhD thesis, Brown University, 2011.

[63] C. Papamanthou and R. Tamassia. Time and space efficient algorithms for two-party authenticated data. In *ICICS*, 2007.

[64] C. Papamanthou, R. Tamassia, and N. Triandopoulos. Authenticated hash tables. In *ACM CCS*, 2008.

[65] T. P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *CRYPTO*, 1991.

[66] C. Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4:161–174, 1991.

[67] H. Shacham and B. Waters. Compact proofs of retrievability. In *ASIACRYPT*, 2008.

[68] M. A. Shah, R. Swaminathan, and M. Baker. Privacy-preserving audit and extraction of digital contents. Technical report, HP Labs Technical Report No. HPL-2008-32, 2008.

[69] C. Wang, Q. Wang, K. Ren, and W. Lou. Privacy-preserving public auditing for data storage security in cloud computing. In *INFOCOM*, 2010.

[70] Q. Wang, C. Wang, J. Li, K. Ren, and W. Lou. Enabling public veriability and data dynamics for storage security in cloud computing. In *ESORICS*, 2009.

[71] A. C.-C. Yao. How to generate and exchange secrets. In *FOCS*, 1986.

[72] Q. Zheng and S. Xu. Fair and dynamic proofs of retrievability. In *CODASPY*, 2011.

A Additional Definitions

A.1 DPDP Algorithms [37]

In this section, we repeat the DPDP algorithm definitions by Erway et al. [37]. In general, the algorithms presented below are all considered to be probabilistic polynomial-time algorithms.

- $\mathsf{KeyGen}(1^k) \rightarrow \{\mathsf{sk}, \mathsf{pk}\}$ is run by the client. Its input is a security parameter, and it outputs a secret key and a public key. The client stores both keys, and sends the public key to the server.
- $\mathsf{PrepareUpdate}(\mathsf{sk}, \mathsf{pk}, F, \mathsf{info}, M_c) \rightarrow \{\mathsf{e}(F), \mathsf{e}(\mathsf{info}), \mathsf{e}(M)\}$ is run by the client. The keys are input to the algorithm, as well as information about the update and the updated data. The output is an encoded version of the updated data, as well as corresponding information (e.g., updated block number), and a new metadata. The client sends this update information and encoded data to the server.
- $\mathsf{PerformUpdate}(\mathsf{pk}, F_{i-1}, M_{i-1}, \mathsf{e}(F), \mathsf{e}(\mathsf{info}), \mathsf{e}(M)) \rightarrow \{F_i, M_i, M'_c, P_{M'_c}\}$ is run by the server after receiving an update request. The server knows the previous versions of the file and the metadata. Using the update information sent by the client, the server updates the data and the metadata. Then, the server sends the updated metadata and associated proof to the client.
- $\mathsf{VerifyUpdate}(\mathsf{sk}, \mathsf{pk}, F, \mathsf{info}, M_c, M'_c, P_{M'_c}) \rightarrow \{\mathsf{accept}, \mathsf{reject}\}$ is run by the client to verify the proof sent by the server. It either accepts or rejects.
- $\mathsf{Challenge}(\mathsf{sk}, \mathsf{pk}, M_c) \rightarrow \{c\}$ is run by the client. The goal is to output some random challenge to send to the server to check for integrity.
- $\mathsf{Prove}(\mathsf{pk}, F_i, M_i, c) \rightarrow \{P\}$ is run by the server after receiving a challenge. The server must output the corresponding proof and send to the client.
- $\mathsf{Verify}(\mathsf{sk}, \mathsf{pk}, M_c, c, P) \rightarrow \{\mathsf{accept}, \mathsf{reject}\}$ is run by the client to verify the proof sent by the server in response to the client's challenge. If the algorithms accepts, the client can rest assured that her file is kept intact.

Following Erway et al. [37], static PDP schemes can be thought as a special case of this definition, where the only possible update type is *full re-write*. Thus, throughout our paper, we do not find it necessary to define static PDP schemes separately.

A.2 Agreement Protocols

For completeness, we define agreement protocols, which are *specialized fair exchange protocols*. An agreement protocol involves two parties, Alice and Bob, and a trusted authority, called the Arbiter. Note that, we will use *the Judge* to denote the authority for official arbitration, and *the Arbiter* to denote the authority for agreement. We present their functionalities separately for clarity and separation of duty; but in a real implementation they can as well be a single entity.

For an agreement protocol to work, the message that is agreed on must be **verifiable** as defined by Küpçü and Lysyanskaya [53]. Informally, *agreement requires that the agreed-upon message can be verified by some algorithm for correctness of the actions of agreeing parties* (some sort of either supporting or falsifying proof is necessary; we direct the reader to [53] for a formal definition). It is enough if one party can generate a proof that is verified using the verification function associated with the fair exchange protocol. Henceforth, to be consistent with the definitions above, we shall assume the party who can generate such a proof is the Responder. Furthermore, for the simplicity of the presentation, we will present an agreement protocol between Alice and Bob, and later on assume they are the Claimer and the Blamed. An agreement protocol for a static message is just a fair exchange protocol. In an optimistic agreement protocol, just as in an optimistic fair exchange protocol [3], the Arbiter gets involved only in case of a dispute.

Definition A.1 ((Optimistic) Agreement Protocol)**.** *An (optimistic) agreement protocol for a message m is an (optimistic) fair exchange protocol at the end of which Alice obtains the signature of Bob on the message ($sign_B(m)$) and Bob obtains the signature of Alice on the message ($sign_A(m)$), or both parties obtain no such signatures.*

We will not repeat the definition of an (optimistic) fair exchange protocol here, but instead redirect the reader to previous work [3, 2, 1, 52, 53, 59, 4, 33, 51]. Since fair exchange cannot be performed without a trusted third party [61], agreement protocols will also employ an arbiter.

An agreement protocol for a dynamic message needs to make sure both parties ***agree on the latest version of the message***. Again, at least one of the parties must be able to prove that the message is correctly formed based on the rules of an associated contract (and hence passes the verification check done by the Arbiter). This evidence must also prove that this message is the latest version of the message. To prove that something is the latest version, we shall use monotonic counters. As long as the monotonicity of the counter is well-defined and known by the Arbiter, it does not matter how the counter is implemented. For example, the counter may be monotonically incremented starting at 0, or it may be represented by a monotonically decreasing series of primes starting at 96079.

Definition A.2 ((Optimistic) Dynamic Agreement Protocol)**.** *An (optimistic) dynamic agreement protocol for a list of messages m_i is an (optimistic) fair exchange protocol where Alice obtains the signature of Bob on the message and the counter ($sign_B(m_i, i)$) and Bob obtains the signature of Alice on the message and the counter ($sign_A(m_i, i)$) at the end of the round represented by the counter i, or both parties obtain no such signatures at that round.*

The definition implies that an (optimistic) dynamic agreement protocol can be constructed by performing an (optimistic) agreement protocol at each round, sequentially. However, this is not necessarily the only way of achieving dynamic agreement.

Finally, note that the definitions can be relaxed to have the message signed by Alice and Bob be different, in a straightforward way.

B Analysis and Future Work

The security of all our schemes have been proven. In this section, we will have a semi-formal analysis of privacy and other properties of our protocols, mainly concentrating on the automated-payment official arbitration scheme, and mention possible future work.

Our protocols are designed to protect the **privacy of the client**. There are only two pieces of information that can identify the client during resolution: the public key of the client and the payment (assuming IP address does not identify the client; anonymous-routing techniques may be employed [32] if necessary). If the client chooses a new public key pk_C for each contract, and if the payments are made using e-cash [27], none of these can be used to identify the client. Moreover, the Judge only needs to know that the contacting party is a client, not who she is. This can be achieved using standard techniques (see [3, 53]) employing one-way functions, or simply via signatures, since the public key of the same client may be different in each contract. Furthermore, the client can always store (independently) encrypted blocks at the untrusted server, hence keeping even the file itself private (even from the Judge). We assume that the metadata itself is not enough for identifying the client (or the file). This is the case in the Erway et al. construction [37] where the metadata is the hashed root of a skip list data structure, and the Ateniese et al. construction [5] where the metadata is just an RSA group information and a pseudorandom function key.

Since in a real scenario we expect servers to be well-known entities, our focus is on client anonymity. If the server anonymity is not desired (while keeping the client anonymous), the server warranty can be made using electronic checks [28, 3] instead of e-cash, thereby improving performance of our Payment Scheme. Just as in the DPDP protocol, the client needs a way to reach the server (e.g., IP address or domain name); the Judge also needs this information to request the signature from the server (and this information should be present in the contract). But the server can still pick a different signature public key for each contract. Even though our scheme also provides nice privacy-related options for the server, at this point, we do not claim that our protocol is fully peer-to-peer (p2p) friendly. It is left as future work to analyze exactly what parts of the system may be unwanted in such a privacy-preserving p2p storage system, and to fix those pieces.

Note that, throughout all our protocols, we assume an authenticated secure channel between our parties. This can be achieved using standard techniques including (password-authenticated) key exchange [31, 12, 13, 40, 46, 56, 17] or SSL/TLS. If the following mechanisms are employed, then this requirement may be relaxed during parts of our schemes:

- Each contract may be associated with a random k-bit value picked by the client. Then, the client may sign this random value together with the counter. This prevents use of the client's signature for another contract (just in case the client does not pick a new signature key pair for each contract). The same applies to the server's signatures.

- The server may also sign the proof and the counter: $sign_S(P, ctr_S)$. This shows that the proof is provided by the server, eliminating the need to send it over an authenticated channel. The random contract identifier can also be used here to link this signature to the contract. In this case, the verifiable escrow for the payment in our Payment Scheme should be sent *after* receiving the metadata, but *before* receiving the proof and sending the signature on the counter. Still, this will not constitute a problem since the Arbiter will not provide the payment without the client's signature $sign_C(ctr_C)$, and the client will not sign the counter without the proof (thus the proof of security of our scheme remains unaffected).

- If the server signs the proof (together with the challenge sent by the client), it will be

possible for the client to provide supporting evidence that the server corrupted her data. But, if a server knows that the data is corrupted (either the server corrupted the data willingly, or became aware of a corruption due to some failure), then one should not expect the server to sign an incorrect proof. Furthermore, the server may accept spending some more time on verifying each proof himself before signing and sending. Hence, overall, there are countermeasures that the server may take against this process, and the decision on whether or not this extra signature is worth putting is left as the system designers' decision.

- The measures above take away the need to have signatures to be exchanged over a secure and authenticated channel between the client and the server.

- If confidentiality is not an issue, the data can be sent over an insecure channel. The same applies to the metadata and the proof.

Remember that if denial requests between the same two participants (identified by their public keys, or the receipt, or the contract identifier) have been repeated many times, the Judge may choose to charge the participants. Here we are at a disadvantage by removing the trusted PKI, since the client and server can jointly mount a denial-of-service attack to the Judge by creating new keys and receipts and asking for dispute resolution each time. They will not be paying resolution fees if the Judge does not charge for the first few attempts. With a trusted PKI, it would be easier to link multiple resolution requests and charge the responsible parties if such requests occur often. This presents a trade-off between anonymity and resilience to Sybil attacks [35]. Therefore, a client should stop doing business with a server that fails to comply with the update protocol several times. In a symmetric sense, the server should stop doing business with a client who repeatedly tries to frame him. But since it is hard for a server to change the contract he signed, the Judge comes to rescue: By charging multiple invalid claims, the Judge helps an honest server stop doing business with a malicious client by discouraging the client from sending invalid claims. The server can simply deny signing the next contract with that client, and if the client still wants to bug the server with invalid claims over the existing contract, she needs to pay the Judge.

Payments to the Judge may be done through various methods. If the Judge is ultimately trusted, the Claimer shall send a payment when making the claim, and if the invalid claim punishment is not necessary, then the Judge shall return the payment back. Another alternative is to have another independent TTP who processes payments to the Judge. When making a claim, the client provides a verifiable escrow of the payment to the Judge. This verifiable escrow is under the independent TTP's public key, and therefore the Judge needs to take all the messages related to the arbitration process to the independent TTP, showing mischief, to obtain the payment.[19]

Finally, following the paradigm in [3, 53] it should possible to remove the use of time-outs from our one-time setup phase completely by using one additional verifiable escrow in the system. More importantly, our efficient optimistic dynamic agreement scheme can be used in fair multi-exchange scenarios where the next exchange is defined within the current one (e.g., in our case, the update counter *ctr* and the DPDP provable-update mechanism – PrepareUpdate, PerformUpdate, VerifyUpdate– serve that purpose).

[19]The server's signature on the proof is necessary here, since authenticated channels cannot be used as proof to a third party (due to the use of symmetric cryptography in practice). This applies to any message exchanged during the arbitration process, as well as the server also signing the Judge's messages. If the server does not sign the Judge's requests, the Judge can penalize the server. Note that charging the server is easier since the server is assumed to be a well-known entity.