Official Arbitration and its Application to Secure Cloud Storage

Alptekin Küpçü
Koç University Department of Computer Engineering, İstanbul, Turkey
akupcu@ku.edu.tr

May 15, 2012

Abstract

Many cryptographic protocols exist that enable two parties to exchange items (e.g., e-commerce) or agree on something (e.g., contract-signing). In such settings, disputes may arise. Official arbitration refers to the process of resolving disputes between two (or more) parties by a trusted and authorized Judge, based on evidence provided. As an example, consider the secure cloud storage scenario where there needs to be an official arbitration process between the client and the server in case of data loss or corruption. Without such a mechanism that can be officially used by the Judge in the court, the barrier on the enterprise adoption of such systems is high.

In this paper we first formally define official arbitration, and then provide several general-purpose official arbitration protocols. Later, we focus on secure cloud storage, and provide efficient official arbitration schemes that can be used on top of any secure cloud storage scheme. We furthermore present a completely automated system where the Judge can just be a computer instead of a human being. All our constructions have security proofs, and we conclude with performance measurements showing that our overhead for official arbitration is roughly **2 ms** and **80 bytes** for each update on the stored data.

Keywords: Cryptographic protocols, fair exchange, arbitration, cloud storage, public verifiability.

1 Introduction

Many cryptographic protocols are being developed every day, where possible disputes between the participants may arise. Such scenarios the include electronic commerce setting where the buyer and the seller may be at a dispute, an outsourced computation setting [10] where the boss and the contractor(s) may be disputing the correctness of the computation, or a secure cloud storage setting where the server may have corrupted the client's data, hence leading to a dispute. In this paper, our aim is to provide official arbitration (i.e., legal enforcement) for various cryptographic applications, again through the use of cryptography, and provide automated resolution mechanisms so that the Judge can actually be a computer rather than a human, greatly reducing costs and increasing efficiency of the courts. To provide a concrete example, in the second part of this paper, we will focus on providing official arbitration for secure cloud storage schemes.

We shall also consider agreement (i.e., contract signing) protocols, through which two parties agree on a contract. Then, using an official arbitration protocol, the parties may use this agreement as evidence to a trusted Judge in case of a dispute. Based on other available supporting or falsifying evidence, the Judge can arbitrate between the parties and punish the guilty ones using her authority. To exemplify this scenario, consider a storage outsourcing system such as Amazon S3, Google Documents, or Microsoft Azure. In this scenario, the client outsources storage of her files to the server, but does not necessarily trust the server. Many systems have

been developed to provide means for the client to verify if the server is keeping her data intact based on some metadata M of the file the client has [5, 45, 64, 7, 36, 17, 29, 67, 68]. **Yet, to be adopted at the enterprise level, such systems must provide a way for the client and the server to resolve problems with a Judge in case of a dispute about the data being intact**.

The basic idea is that the client and the server shall agree on the metadata M that can later be presented to the Judge. At this point, through the use of other proofs (i.e., the server's proof that the file is intact), the Judge can provide **official arbitration** between the client and the server. Note that this is a stronger property than *public verifiability*, which only states that third parties can also check for the integrity of the client's data; *but the result of this check would not necessarily be trusted to be used for legal enforcement.* In Section 4.1 we see an example where official arbitration is clearly different from public verifiability.

Furthermore, we shall differentiate between the case where the agreement is static (one-time) versus the case where the agreed-upon message keeps changing over time dynamically. An example of the static case would be a secure cloud backup scheme [5, 45, 64, 34, 6, 46] where the agreement contains the metadata on the stored file. This metadata is prepared once during the upload, and can be used many times for verification purposes. We will see that providing agreement and official arbitration in this case is simpler, as expected. The second case can be exemplified in a dynamic cloud storage scenario [36, 7, 67, 68] where the agreement again contains the metadata on the stored file, but it keeps changing with each update to the file. Therefore, such scenarios need a dynamic agreement scheme to be used in official arbitration. We will see that known straightforward solutions are not enough in such a case, and provide a novel solution that is efficient and scalable.

We start by defining secure official arbitration, as well as agreement protocols, in Section 2. Then, in Section 3 we provide several general-purpose official agreement schemes, and propose alternative methods to fully automate the arbitration process. We present a basic scheme for static cases, and a dynamic scheme as well as an optimized efficient scheme for dynamic cases. Afterward, in Section 4, we take secure cloud storage as our sample application area, and provide various protocols that are targeted and scenario-tailored versions of our general-purpose protocols. In particular, we develop on top of the definitions provided by Erway et al. [36], since they provide a general definition for secure cloud storage systems. For each one of our protocols, we also prove their security. At the end, we analyze performance, and provide comparisons.

Contributions: Our contributions are as follows:

- We provide a formal definition of secure official arbitration.
- We provide the *first constructions of general-purpose official arbitration schemes* that are applicable to various static and dynamic scenarios, and capable of performing fully-automated official arbitration.
- We provide a very efficient optimistic dynamic agreement protocol where two parties would like to agree on the latest version of a message that keeps getting updated and at least one party can prove that the message is formed correctly (according to some well-defined rules in a contract).
  - Our method is *optimistic* in 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.
- We combine our dynamic agreement protocol with our dynamic official arbitration protocol to obtain **the first general official arbitration for any secure cloud storage scheme** (defined as dynamic provable data possession schemes [36])

– 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**.

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* (denoted $\mathcal{P}_C$) and the party providing the evidence *the Responder* (denoted $\mathcal{P}_R$). Depending on the scenario, these three 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*, and be denoted by $\mathcal{J}$.[1]

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 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 arbitrating party. Note that at this point we allow for general claims to be made, therefore the input to the algorithm is the entire state of the Claimer:

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

Normally, one would expect the Judge to transfer the claim directly from the Claimer 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`$(c) \rightarrow c'$: 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 claim, (s)he is expected to respond to the Judge with some evidence; either supporting the claim, or falsifying it. The algorithm used by the Responder is defined as follows:

`prove`$(c') \rightarrow e$: 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. Even though we do not explicitly show, it is assumed that the state of the Responder is also input to this algorithm.

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*. In our definitions, we denoted the Claimer and the Responder as different parties. Yet, one can perfectly imagine the Claimer and the Responder being the same party (in such a case, the evidence is expected to be a supporting evidence to the claim).

[1] 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 .

Furthermore, when the Responder and the Blamed are the same party, we expect the evidence to be a falsifying evidence. We will 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.

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:

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

Note that the Responder may be a party completely outside the claim ($\mathcal{P}_B \neq \mathcal{P}_R \neq \mathcal{P}_C$), but still helping the Blamed by providing a falsifying evidence (through $\texttt{prove}_F$), or helping the Claimer by providing supporting evidence (through $\texttt{prove}_S$). 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 `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:

$\texttt{decide}(c, e) \to \{O, C, B, N\}$: 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.

Before we define security of an official arbitration scheme, we would like to give some example scenarios. One example is that the Claimer claims that he was supposed to obtain something from the Blamed, as signed in a contract. The `claim` algorithm then outputs a claim containing the message m that represents the contract, allegedly signed by the Blamed $\mathcal{P}_B$. The `request` algorithm requests the signature on the message m from the Responder. In this scenario, it is natural to consider the Claimer and the Responder being the same party (i.e., $\mathcal{P}_C = \mathcal{P}_R$). The `prove` algorithm (equivalently, $\texttt{prove}_S$) returns the signature $sign_{\mathcal{P}_B}(m)$ of $\mathcal{P}_B$ on the message m. The Judge's `decide` algorithm's job is to verify that signature on the message m given in the claim c. If the signature verifies, the Judge may continue her arbitration process using the signed contract m (that part may be outside the scope of the digital resolution process).

Now consider the case where the Claimer $\mathcal{P}_C$ and the Responder $\mathcal{P}_R$ are different parties. Such a setting may arise in secure cloud storage schemes where some sort of provable data

storage scheme is used [5, 36, 7, 45, 64, 34, 6, 67, 17, 29, 65, 68]. The Claimer may be the client who is claiming that the server $\mathcal{P}_B$ corrupted her file. The Responder will probably be the server (i.e., $\mathcal{P}_R = \mathcal{P}_B$) who needs to provide falsifying evidence that the file is kept intact using cryptographic proofs. This is a very interesting and useful scenario, and hence we are going to analyze it in more detail in Section 4.

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[2] either one of the following two conditions hold:

1. There exists a supporting evidence.
2. All falsifying evidence provided is invalid.

Similarly, a claim will be called **invalid** if and only if[2] either one of the following two conditions hold:

1. There exists a falsifying evidence.
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.

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.

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,*

[2]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.

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

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*.*

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 second case (the case of an invalid claim) of both definitions may 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 $\leq neg(k)$). Remember that a negligible function $neg(k)$ is smaller than any polynomial function in k for all values of k greater than some constant k_0.

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 [49]. Therefore, we keep this complicated setting outside the scope of this paper and leave it as future work.

2.2 Agreement Protocols

For completeness, we define agreement protocols, which are *specialized fair exchange protocols.* An agreement protocol involves two parties 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 [51]. 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 [51] 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 2.5 ((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, 50, 56, 4, 33, 49]. Since fair exchange cannot be performed without a trusted third party [58], 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 2.6 ((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.

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

3 Constructions

In this section, we provide several constructions of secure official arbitration schemes for different scenarios. We will describe the schemes from simple to complex, and from general to specific.

3.1 Basic Scheme

In a general *contract-resolution* scheme, 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 Section 2.2. The Claimer claims that the Blamed has violated the terms of the contract. Below, we will describe each algorithm of the protocol:

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

`decide`$(c, e) \rightarrow \{O, C, B, N\}$: 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 possibly 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.[3] 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.

3.1.1 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.[4] 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* [51, 11]. Furthermore, there are many protocols where the arbitration can be completely automated [3, 2, 1, 50, 11, 16, 8, 4, 56, 9].

`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.

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

[3] 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.

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

`prove`$_F$: The output evidence should include the signature $sign_{\mathcal{P}_C}(x)$ of $\mathcal{P}_C$ 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 $sign_{\mathcal{P}_B}(m)$. In the second one, the claim is that the Blamed has violated the terms in m, and the ***falsifying evidence*** is $sign_{\mathcal{P}_C}(x)$.

The proof of the Basic Scheme follows the same lines, but since real-life evidence (e.g., witnesses) are outside our scope, we limit out attention to the E-commerce Scheme in the following theorem.

Theorem 3.1. *If the digital signature scheme used is secure against existential forgery [43][5], 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.[6]
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 may be used to break the security of the signature scheme again 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. □

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

[6]Remember that we are assuming the exchange of x and $sign_{\mathcal{P}_C}(x)$ is performed fairly; it's arbitration is the Arbiter's job, not the Judge's. The Blamed may come up with a counter-claim stating that the exchange of the payment with the receipt failed. In this case, the Arbiter needs to be involved.

3.2 Dynamic Scheme

Now consider a scenario, where the agreed-upon message m is dynamic: it changes over time. The official arbitration 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.

Both parties will be holding counter values, initialized to some agreed-upon value (e.g., 0). 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. The following modifications to the Basic Scheme are necessary:

$\texttt{claim}(state) \rightarrow c$: 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 (it is possible that he violated the dynamic agreement protocol, or the contract itself).

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

$\texttt{prove}_S(c') \rightarrow e$: The output evidence e should include the signature $sign_{\mathcal{P}_B}(m, ctr_B)$ of $\mathcal{P}_B$ 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 3.1).

$\texttt{prove}_F(c') \rightarrow e$: The output evidence e should include the signature $sign_{\mathcal{P}_C}(m', ctr_C)$ of $\mathcal{P}_C$ 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., $sign_{\mathcal{P}_C}(x)$).

$\texttt{decide}(c, e) \rightarrow \{O, C, B, N\}$: The Jugde performs the following steps[7][8]:

1. If the claim is about the agreement protocol, the Judge directs the parties to the Arbiter, whose role is well-defined for a fair contract-signing protocol [51].[9]
2. Otherwise, verify the signature $sign_{\mathcal{P}_B}(m, ctr_B)$. If the verification fails, then output C and stop.
3. Verify the signature $sign_{\mathcal{P}_C}(m, ctr_C)$. If the verification fails, then output B and stop.
4. $ctr_B > ctr_C \Rightarrow \mathcal{P}_B$ is cheating by trying to replay an old signature from $\mathcal{P}_C$. Output B and stop.
5. $ctr_C > ctr_B + 1 \Rightarrow \mathcal{P}_C$ is cheating by trying to replay an old signature from $\mathcal{P}_B$. Output C and stop.
6. $ctr_C = ctr_B + 1 \Rightarrow$ There are two possibilities in this case. (1) $\mathcal{P}_B$ did not complete the dynamic agreement protocol (willingly or due to failures). Again direct to the Arbiter. (2) Everything went well but $\mathcal{P}_C$ is trying to overload the Judge. This can be limited by

[7] We are assuming in the agreement protocol the Claimer sends his signature first.

[8] If the Blamed sends her signature first in the fair exchange protocol, then the checks in steps 4, 5, and 6 should be replaced by $ctr_B > ctr_C + 1$, $ctr_C > ctr_B$, $ctr_B = ctr_C + 1$ respectively.

[9] This part may or may not be outside the scope of electronic resolution process.

making $\mathcal{P}_C$ pay for such requests after some number of free resolutions. Normally, $\mathcal{P}_C$ should stop working with $\mathcal{P}_B$ if he faces problems several times.

7. $ctr_B = ctr_C \Rightarrow$ If $m \neq m'$ then again direct to the Arbiter. Otherwise, this means $\mathcal{P}_C$ and $\mathcal{P}_B$ agree on the latest message $m = m'$, and the Judge processes further evidence (digital or real) as in the Basic/E-commerce Scheme and decides accordingly.

Theorem 3.2. *If the digital signature scheme used is secure against existential forgery [43][5], and the signatures are exchanged using an (optimistic) dynamic agreement protocol, then the Dynamic Scheme is a secure official arbitration scheme.*

Proof. If a fair signature exchange failed, in the worst case we have[7] $ctr_C = ctr_B + 1$ or $m \neq m'$. In those cases, the participants must contact the Arbiter of the agreement protocol to resolve their issue first, and assume the latest updated agreement 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 of the agreement (except the $ctr_C = ctr_B + 1$ case above). 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 one, 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, as required.

The rest of the Judge's decision is the same as the Basic/E-commerce Scheme, and therefore the security proof can be applied directly here, which means the Dynamic Scheme is also an **invalid-claim-punishing official arbitration scheme**. Again, the security parameter k represents the security parameter of the signature scheme used. □

Note that in this scenario, with each update to the agreement, a new fair contract signing protocol must be executed by the Claimer and the Blamed, since this is the most straightforward way of achieving dynamic agreement. Unfortunately, this is a very costly operation, and therefore should be avoided. **In the next section, we will consider the problem as a whole: the agreement phase itself together with its official arbitration**. We will present a novel optimistic dynamic agreement scheme that is secure when used within the official arbitration context that we present in the next section. This approach will help us get rid of the costly fair exchange at each step, and will result in great savings in time and bandwidth, as discussed in Section 4.4.

3.2.1 Efficient Optimistic Dynamic Agreement

We are still considering the dynamic scenario, but we would like to provide an efficient scheme for the Claimer and the Blamed, by avoiding the costly fair contract-signing at each renewal of the contract. To do this, we need to consider the process as a whole, rather than completely separating the agreement and official arbitration protocols.

In our **novel efficient optimistic dynamic agreement protocol**, we will 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 $sign_{\mathcal{P}_C}(m, 0)$. *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 rest proceeds as follows when 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 $sign_{\mathcal{P}_C}(m', ctr_C)$ to the Blamed.
3. If this signature is valid and the counter value and the contract is correct, then the Blamed responds with $sign_{\mathcal{P}_B}(m', ctr_B)$.
4. Upon receipt, the Claimer checks this signature, as well as the message and the counter.

During this process, if anything goes wrong, they contact the Arbiter. The Arbiter's resolution process is defined as follows (assuming initial fair exchange completed successfully):

- If the Claimer sent her signature but did not receive a response (or received a non-verifying response), then she contacts the Arbiter. The Arbiter obtains $sign_{\mathcal{P}_C}(m', ctr_C)$ from the Claimer, and $sign_{\mathcal{P}_B}(m'', ctr_B)$ from the Blamed. If $ctr_C = ctr_B$ and $m' = m''$ then the Arbiter sends the Claimer's signature to the Blamed, and vice versa.
- If the Claimer received a signature but with $m'' \neq m'$, then she contacts the Arbiter. The Arbiter obtains both signatures from the Claimer, and if $ctr_C = ctr_B$ but $m'' \neq m'$, he contacts the Blamed (the resolution after this point may be outside the scope of the digital process, but it is also possible that the Blamed just made an error and corrects it when the Arbiter contacts him).
- For any case that is not resolved as above, the parties must assume that the exchange failed, and therefore the contract is not updated.

Note that this resolution process may help smooth out problems due to honest failures, and only suggests failure notification in case of malicious acts. As an example, consider the secure cloud storage scenario, where the updated contract would correspond to the updated metadata that enables file verification. In case that the above exchange fails (possibly even after a timeout to tolerate failures), then the client should assume that the update failed, and therefore should not trust the server to store the latest version of her file. Furthermore, the client is expected to stop working with the server if several such malicious failures occur.

On top of this efficient optimistic dynamic agreement protocol, we employ our Dynamic Scheme as the official arbitration protocol, and call the resulting scheme the **Efficient Scheme**.

Theorem 3.3. *If the digital signature scheme used is secure against existential forgery [43][5], 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 Scheme is a secure official arbitration scheme.*

Proof. Assuming the initial fair exchange protocol has completed successfully, 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 Dynamic 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 of our efficient optimistic dynamic agreement protocol, 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 contact the Arbiter instead of

going through another exchange). After the increment of the counters (step 1), 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 $sign_{\mathcal{P}_C}(m, ctr_C - 1)$. After step 2, the Claimer still has $sign_{\mathcal{P}_B}(m, ctr_B - 1)$, whereas the Blamed now has $sign_{\mathcal{P}_C}(m', ctr_C)$. If step 2 has not been executed, then both parties assume the round has never started. Similarly, if anything goes wrong in steps 3 and 4, the Claimer contacts the Arbiter, as explained in the Efficient Scheme, and must assume the contract update failed if the Arbiter cannot resolve the issue.

If the Blamed sends his signature in step 3, and verifications in step 4 succeeded, this means both parties have each other's latest signature: the Claimer has $sign_{\mathcal{P}_B}(m', ctr_B)$, and the Blamed has $sign_{\mathcal{P}_C}(m', ctr_C)$. At this point, the protocol becomes the same as the Dynamic 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**. □

For this protocol to work, it is necessary that *the relationship between the preceding contract and the succeeding contract is well-defined, provable, and known* by the Arbiter. In DPDP systems, this is achieved by the server proving that an update is performed correctly; a property that can be verified by the Arbiter using both the new metadata resulting after the update and the preceding metadata defining the file before the update.

4 Applications

One of the most common cases where official arbitration is required for a dynamic scenario is the secure cloud storage case, as exemplified by Dynamic Provable Data Possession (DPDP) schemes [36, 7, 67, 68]. Consider a storage outsourcing system such as Amazon S3, Google Documents, or Microsoft Azure. In this scenario, the client outsources storage of her files to the server, but does not necessarily trust the server in keeping her data intact. Furthermore, the system is dynamic: the client would like to be able to modify/delete her files. Ateniese et al. [5] and Juels and Kaliski [45] were the first to provide means for the client to verify whether or not the server is keeping her data intact based on some metadata M of the file the client has. Later on, Erway et al. [36] introduced the fully-dynamic scenario, where the client can keep updating her data, still being able to verify its integrity. The definitions we will follow will mostly be based on PDP [5] and DPDP [36]. Since then, the outsourced storage scenario attracted much interest from researchers [5, 45, 6, 34, 36, 64, 7, 29, 17, 67, 68]. Yet, to be successfully adopted in an enterprise setting, such systems must provide a way for the client and the server to resolve problems with the help of a Judge in case of a dispute about the data being intact [47, 48].

In the following sections, we will provide various ways of applying the official arbitration schemes of Section 3. For completeness, we describe briefly how the Basic Scheme can be applied on top of PDP [5], in Section 4.1. Next, we present a barebone protocol (based either on the Dynamic Scheme or the Efficient Scheme), which provides official arbitration for any DPDP scheme [36] with no secret keys. We then describe an automated protocol, where the Judge can even automate punishment through payments using electronic cash [27] or electronic checks [28].

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 client's data's integrity 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 4.1).

Shah et al. [65] provide an audit (i.e., public verifiability) method for a static storage scenario, and has more features including zero-knowledge property (i.e., the file contents are 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 [67] but then provided a mechanism to officially arbitrate between the client and server without a stateful Judge [66]. They furthermore provided a novel batch audit mechanism. Unfortunately, their scheme applies on top of their own construction, and hence is not general as ours. Furthermore, their official arbitration requires bilinear maps and random oracle model, while no formal security definition is given. Actually, our official arbitration mechanisms can also be applied on top of [67] with a very minor modification (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), and the performance will be better (In [66] authors obtain about 470 ms overhead for individual audits, and 370 ms overhead for batch audits on top of the underlying DPDP system. See Section 4.4 for our overhead).

Simultaneously with our work, Zheng and Xu presented FDPOR [68], 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 4.1. 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). Moreover, from their construction and proof, it is not clear if they achieve dynamic POR vs. dynamic PDP. The reason is that the proof sent by the server depends only on the challenged blocks, and nothing else. Besides, they use error-correcting codes for each block independently. Therefore, the server may freely delete an encoded block completely, and would not get caught unless that block is challenged (either during a proof or update).

Official arbitration, to the best of our knowledge, is a new formalization. The Dynamic Scheme and the Efficient Scheme can be applied on top of *any* DPDP protocol where the client has no secret key, as defined by Erway et al. [36]. Note that, our schemes are the only schemes that can be used for official arbitration of any DPDP scheme, to the best of our knowledge. Therefore, in the performance section, the only two schemes we can and will compare are these two. Furthermore, again to the best of our knowledge, our scheme in Section 4.1 is the only scheme that can be applied on top of any (static) PDP protocol to provide official arbitration.

Agreement protocols can be done using various contract-signing protocols that have existed in the literature [16, 38, 2, 1, 3, 4, 9, 8, 50, 54, 56]. But, for the first time, we considered dynamic agreement and used an efficient optimistic dynamic agreement protocol together with an official arbitration protocol, optimizing them both for dynamically-changing scenarios.

4.1 Official Arbitration for PDP

For completeness, we include a protocol, based on the Basic/E-commerce Scheme, to provide official arbitration for PDP schemes. 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 metadata M [5, 64]. 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.

Public verifiability does not imply official arbitration: To provide a concrete example, we shall build upon the publicly-verifiable PDP scheme by Ateniese et al. [5]. The main difference between their public-verifiability extension and our official arbitration solution is that

their extension cannot be used for official arbitration purposes. 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.

To address this framing attack, our Basic scheme must be employed on top of this publicly-verifiable PDP scheme, and 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 is the following: After the initial upload and the client made the v value public, the server checks for correctness of v[10] 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 therefore should consider her data not backed-up. Otherwise the protocol has succeeded, and the signature can be used for official arbitration in case of a dispute.

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 [64] 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 [62, 37, 30] and zero-knowledge proofs [42, 41, 12], as well as secure multi-party computation schemes [40, 15, 25, 26].

We require only minimal additions to PDP-type protocols: 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 official arbitration. *The client will be the Claimer, and the server will be the Blamed.* As a supporting evidence, the client (who is also the Supporting Responder) provides the metadata M, together with the server's signature on it ($sign_S(M)$). The Judge decides as follows:

`decide`: If the supporting evidence contains a valid signature on the metadata M, which is verified using the public key of server in a trusted PKI, then the Judge requests falsifying evidence from the server (who is the Falsifying Responder) by sending a PDP challenge [5]. 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. 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.

Theorem 4.1. *If the digital signature scheme used is secure against existential forgery [43], 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:
(1) There is no storage agreement between the client and the server, hence the server did not sign M. Following the proof of the Basic Scheme, the Judge will output B with only negligible

[10] 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 [64], 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.

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 reduction).
(2) There is 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 according to the proof of the Basic Scheme, 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].[11]

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. □

4.2 Official Arbitration for DPDP

The Dynamic Scheme and the Efficient Scheme as defined in Sections 3.2 and 3.2.1 directly apply to this scenario. The only difference is that we shall refer to the parties as the client (denoted by C) and the server (denote by S). Since both schemes offer secure official arbitration for dynamic scenarios, they are perfectly-suited for a dynamic application like DPDP. As in PDP, 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 some metadata M [36]. 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. For the sake of completeness, we include these algorithms as defined by Erway et al. [36] in the appendix. To use our Dynamic Scheme or Efficient Scheme on top of any DPDP scheme[12], we require the following 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. 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 initiates the official arbitration scheme with the Judge.

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[12]. Therefore, we envision that our official arbitration solution will be extremely easy to deploy and use, compatible with any underlying scheme. Figure 1 depicts the Efficient Scheme applied to the DPDP protocol.

[11] 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 a mathematical sense. For the best explanation and result, we direct the reader to Compact POR [64] and Section 5.9.2 of [47, 48].

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

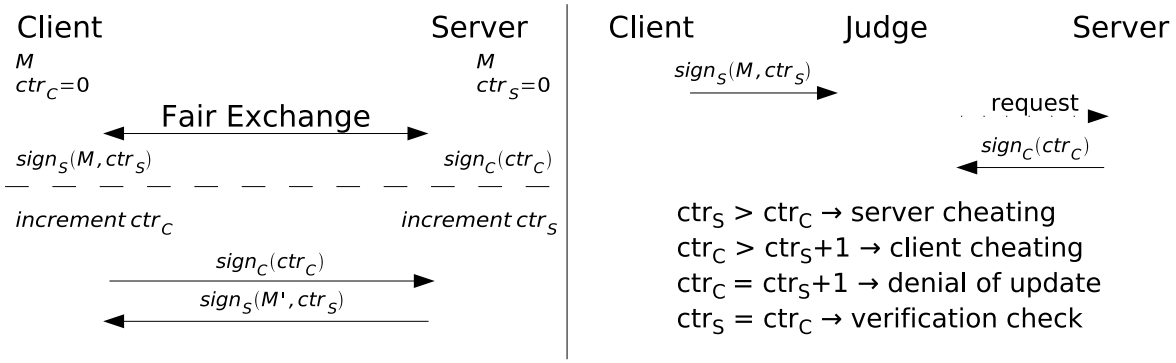

Figure 1: Left side shows our novel optimistic dynamic agreement protocol, where the part above the dashed line is the one time setup and the part below it is performed with every update. Right side summarizes the arbitration procedure in case of a dispute.

Below we present the details of the Efficient Scheme as it applies on top of any DPDP protocol (remember that the Claimer is the client, and the Blamed is the server):

`claim`: We will consider two types of claims in this scenario:

C1: The client claims that the server has corrupted the data.

C2: The client claims that the server denies updating the data. In this case, the client also sends the new data F', and the metadata M of the last successful update.

`request`: For both claims, the Judge request a supporting evidence from the client containing the metadata M and the latest signature of the server, and a falsifying evidence from the server containing the latest signature of the client (just as in the Dynamic Scheme and the Efficient Scheme). Then, depending on the claim number, the Judge makes the following extra requests:

C1: The Judge runs the Challenge algorithm of the DPDP protocol, and sends the challenge to the server.

C2: The Judge runs the PrepareUpdate algorithm of the DPDP protocol using the data F' provided together with the claim.[13] Then, he sends this update request to the server. He furthermore requests new signatures from both the client and the server.

$\texttt{prove}_S$: For both claims, the supporting evidence must include the metadata M and the latest signature of the server $sign_S(M, ctr_S)$ (just as in the Dynamic Scheme and the Efficient Scheme).

C1: Note that the counters of the client and the server should match in this case.

C2: Note that if an update failed, the client needs to use the information about the last successful exchange. Therefore, in this case we assume ctr_S and ctr_C denote the counter values used in the latest successful exchange. Next, the client increments her counter to prepare for the update, and sends a new signature $sign_C(ctr_C + 1)$ to the Judge. (ctr_C still denotes the old value of the counter for preventing ambiguity.)

$\texttt{prove}_F$: For both claims, the server returns the client's latest signature $sign_C(ctr_C)$ (just as in the Dynamic Scheme and the Efficient Scheme). Then, depending on the claim number, the server also sends extra evidence:

[13] 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.

C1: The server runs the Prove algorithm using the challenge provided by the Judge and returns the proof.

C2: 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 [36]). The server also increments his counter and sends a new signature on this new metadata $sign_S(M', ctr_S + 1)$ to the Judge. (ctr_S still denotes the old value of the counter for preventing ambiguity.)

`decide`: This part is as defined in the Efficient Scheme (or the Dynamic Scheme). All received signatures are verified first. If the signature verification fails, blame the party that sent the signature (not the one who is claimed to be the signer, since responders are supposed to provide valid evidence). The only differences in the decision process are the following:

C1: The main clarification is for the case where $ctr_B = ctr_C$ (case 7)(remember ctr_B is ctr_S). In this case, verify the proof sent by the server using the Verify algorithm of the DPDP protocol using the metadata M. If the proof fails, output B (the server is guilty). Otherwise, possibly output C.

C2: The main clarification is for the case in which $ctr_C = ctr_B + 1$ (case 6)(remember ctr_B is ctr_S). In this case, verify the proof-of-update using the VerifyUpdate algorithm of the DPDP protocol. For this verification, use the metadata M provided by the client and signed by the server during the previous successful update with counter value ctr_S. If the proof fails, output B (the server did not perform the update). Otherwise, forward the new signature $sign_C(ctr_C + 1)$ provided by the client to the server, and forward the new signature $sign_S(M', ctr_S + 1)$ as well as the new metadata M' provided by the server to the client[14], and output N.

Theorem 4.2. *If the digital signature scheme used is secure against existential forgery [43][15], and the DPDP scheme used is secure [36], then the scheme above provides secure official arbitration for DPDP.*

Proof. This proof relies on the proofs of the Dynamic Scheme and the Efficient Scheme. We will not repeat those proofs here. Instead, we will only mention DPDP-specific parts.
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 and/or counter value. But such a client may be used to break the underlying signature scheme in a reduction.
Thus the Judge will not decide B, except with negligible probability of 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.[16] Thus, with high probability, the Judge will decide that the server is guilty.
If the server denied the update, he is given a second chance: The Judge himself tries to update on behalf of the client. If the server is still cheating, again due to the security of the underlying DPDP scheme, the proof-of-update will fail to verify with high probability[16], and the Judge will output B.[17]

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

[14] This also fulfills the duty of the Arbiter.

[15] No chosen-message attack is possible in this scenario.

[16] 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.

[17] Remember that the client owns the latest correct metadata signed by the server.

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 (except one case where the output is N). □

Note that *for the denial-of-update case* (claim C2), *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, (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 may be 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/credentials/authentication techniques [19, 63, 21, 22, 52]). This way, we prevent the server from acting maliciously against the client and behaving nicely against the Judge. We leave the details outside our scope, and refer the reader to the *zero-knowledge auditing* property defined by Shah et al. [65].

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.* 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 *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 DPDP scheme, and forwards this data to the client.

Lastly, consider the case where no cheater can be identified and the output is N. If the server acts maliciously several times, the 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. In Section 4.3 we present a dynamic official arbitration scheme where the process, including such payments, can be fully automated.

4.3 Automated-Payment Official Arbitration for DPDP

We now extend our official arbitration protocol for DPDP 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/Arbiter gets paid for her work by the cheating party.* As before, we are assuming the Judge has (official) authoritative power over both parties. In this section we will refer to our original official arbitration protocol for DPDP as the ***barebone protocol*** and our new automated-payment official arbitration protocol for DPDP as the ***payment protocol*** (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.

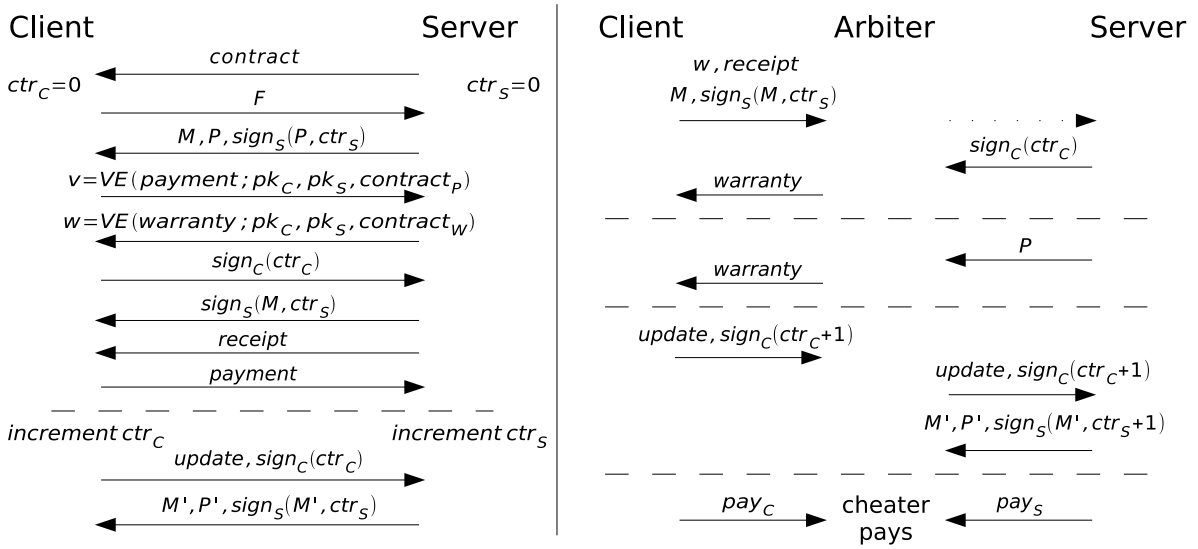

Figure 2: Left side above the dashed line is the setup phase of the automated-payment official arbitration protocol for DPDP, whereas below the dashed line shows a regular update phase. Right side shows dispute resolution by the Judge for various claims and scenarios.

If the client is happy with the contract, she can go ahead and send her file for storage at the server (through the DPDP system, possibly after encoding and tagging). 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 barebone protocol, 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) will be different, providing ***unlinkability*** through public keys. She then prepares a verifiable escrow[18] of her payment $v = VE_{Arb}(payment; pk_C, pk_S, contract_P)$, labelled 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 (it does not verify or the label is incorrect) then the server aborts.

Optionally, the server can provide the client with a warranty check at this point. 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

[18]The notation $VE_{Arb}(value; label)$ denotes an encryption under the public key of the Arbiter of the *value* such that it can be verified 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 Arbiter should decrypt the verifiable escrow.

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 of the barebone protocol (as introduced in the optimistic dynamic agreement protocol of 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*[19] 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), 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 [51]–), then the client sends the payment to the server, ending the setup phase.

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 proof P, and the receipt *receipt*. The Arbiter checks the signatures (using the public keys in the verifiable escrow's label), verifies the proof using the metadata, 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, 8, 50], 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.[20]

Theorem 4.3. *The protocol above is an agreement protocol that guarantees that the server obtains the client's signature and payment, and the client obtains the server's signature and receipt (except with negligible probability, assuming a chosen plaintext secure verifiable escrow scheme, an unforgeable signature scheme, and a payment (e.g., e-cash) scheme which is unforgeable).*

Proof Sketch. 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 point in the protocol, the parties must abort rather than resolve. Yet, until this point, no party received any useful information: the server did not receive the payment (only the encrypted version); similarly the client only received an escrowed warranty, but not the receipt or the server's signature. 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.,

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

[20]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, 50]. Besides, remember that the verifiable escrow contains the client's public key, therefore this verification can easily be performed.

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.

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. 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. □

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 DPDP protocol M. The server only needs the store the signature $sign_C(ctr_C)$ (and of course the file F 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 he is the trusted Judge).

Now that the initial agreement between the client and the server has been completed, we continue by using our optimistic dynamic agreement protocol in the barebone protocol for each update. Thus, the proof that this scheme provides secure official arbitration immediately follows.

The Judge's duty is exactly the same as in the barebone protocol (except that the client must provide the receipt, showing that she deserves the service), with the addition of payment-related actions as follows:

- If the server is found guilty, then the Judge decrypts the warranty escrow and provides it to the client.
- Remember that the Judge may get paid if denial requests 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.[21] Therefore, a client should stop doing business with a server that fails to comply with the update protocol several times.[22][23]

4.4 Performance Evaluation

As we discussed before, fair exchange of signatures is a costly operation. Known optimistic fair exchange schemes employ a verifiable escrow [3, 2, 1, 50, 8, 54]; therefore we can conclude that

[21] 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, thus without 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. The trade-off is left to the user to judge.

[22] Payments to the Judge may be done through various methods: e.g., the client provides a verifiable escrow of the Judge's payment while making the claim. This verifiable escrow may as well be under another independent Arbiter's public key, and therefore the Judge needs to take all the messages related to the arbitration process, showing mischief, to obtain the payment.

[23] The server's signature on the proof is necessary here, since authenticated channels cannot be used as proof to a third party. 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. Making our protocols completely usable in peer-to-peer storage systems using DPDP is left as future work.

the cost of a fair exchange is more than the cost of a verifiable escrow. In our implementation [20, 55] of the most efficient verifiable escrow known to us [24], each verifiable escrow takes about 1 second and 25 KB on a machine with 3 GHz CPU and 4 GB RAM. On the other hand, each DSS signature [57] takes about 1 ms and 40 bytes.

As discussed in the related work section, the only secure official arbitration methods we compare are the Dynamic Scheme and the Efficient Scheme. To provide a real usage scenario and numbers, we consider their application on DPDP using the CVS repositories depicted in Table 1, taken from Erway et al. [36]. Using the method based on the Dynamic 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 escrows, 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**. This makes our approach the first ever practical, efficient, and official arbitration protocol for any dynamic provable data possession scheme[12]. **Per update** (commit), our overhead is less than **0.1 KB** and **2 ms** (without counting the time spent by the underlying DPDP system).[24]

	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: Authenticated CVS service overhead, in total for both the client and the server.

Note that our protocol requires **no growing history**. Only a counter and the latest signature after the last successful exchange needs to be kept at each party (possibly also a warranty and a receipt, when the payment protocol in Section 4.3 is used). In any case, the **storage overhead** for the client and the server is **only tens to hundreds of bytes** each.

4.5 Analysis and Future Work

We have already proven security of all our protocols. 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 protocol is 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 she is a client, not who she is. This can be achieved using standard techniques employing one-way functions [3, 51]. Furthermore, the client can always store (independently) encrypted blocks at the untrusted server, hence keeping even the file itself private. We assume that the metadata itself is not enough for identifying the client (or the file), which is the case in DPDP [36] where the metadata is the hashed root of a skip list data structure.

Since in a real scenario we expect servers to be well-known entities, our focus is on client anonymity. Note that the server warranty can be made using electronic checks [28, 3], thereby making it more efficient, since we are assuming the server is not anonymous. Just as in the DPDP protocol the client needs a way to reach the server (e.g., IP address), the Judge also

[24]Compare to 370 ms overhead of Wang et al. protocol [66] specific to their scheme.

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. Yet, 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 part 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, 13, 14, 39, 44, 53, 18] or SSL. If the following mechanisms are employed, then this requirement may be relaxed during parts of our payment protocol:

- 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 is 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.
- If the server signs the proof (together with the challenge sent by the client), it will be possible for the client to provide a 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 cost 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.

Lastly, our dynamic official arbitration and dynamic agreement protocols can be of independent interest. Following the paradigm in [3, 51] it should possible to remove the use of timeouts from our system completely by using one additional verifiable escrow in the system. More importantly, our 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). In general, our protocols are applicable on top of any *dynamic provable data structure*, including authenticated skip lists [60], authenticated hash tables [61], and many other authenticated data structures [35, 59], where updates can be proven.

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. Ateniese, R. D. Pietro, L. V. Mancini, and G. Tsudik. Scalable and efficient provable data possession. In *SecureComm*, 2008.

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

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

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

[11] 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.

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

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

[14] 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.

[15] M. Ben-Or, R. Canetti, and O. Goldreich. Asynchronous secure computation. In *STOC*, pages 52–61, 1993.

[16] M. Ben-Or, O. Goldreich, S. Micali, and R. L. Rivest. A fair protocol for signing contracts. *IEEE Transactions on Information Theory*, 36(1):40–46, 1990.

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

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

[19] S. Brands. *Rethinking public key infrastructures and digital certificates: building in privacy.* MIT Press, 2000.

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

[21] J. Camenisch and A. Lysyanskaya. An efficient system for non-transferable anonymous credentials with optional anonymity revocation. *EUROCRYPT*, pages 93–118, 2001.

[22] J. Camenisch and A. Lysyanskaya. A signature scheme with efficient protocols. *SCN*, 2576:268–289, 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 *ACM TOCS*, pages 639–648, 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. Mjolsnes, 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*, IT-22(6):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. Lee, and D. 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] C. Dwork, M. Naor, G. N. Rothblum, and V. Vaikuntanathan. How efficient can memory checking be? In *TCC*, 2009.

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

[37] E. Fujisaki and T. Okamoto. Statistical zero knowledge protocols to prove modular polynomial relations. In *CRYPTO*, pages 16–30, London, UK, 1997. Springer-Verlag.

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

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

[40] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *STOC*, pages 218–229, 1987.

[41] 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(3):728, 1991.

[42] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18(1):208, 1989.

[43] S. Goldwasser, S. Micali, and R. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281–308, Apr. 1988.

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

[45] A. Juels and B. S. Kaliski. PORs: Proofs of retrievability for large files. In *ACM CCS*, pages 584–597, 2007.

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

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

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

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

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

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

[52] Y. Lindell. Anonymous authentication. *Journal of Privacy and Confidentiality*, 2, 2010.

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

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

[55] 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.

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

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

[58] H. Pagnia and F. Gärtner. On the impossibility of fair exchange without a trusted third party. *Darmstadt University of Technology*, TUD-BS-1999-02, 1999.

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

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

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

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

[63] C. Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3):161–174, 1991.

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

[65] 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.

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

[67] 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.

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

A DPDP Algorithms [36]

In this section, we repeat the DPDP algorithm definitions by Erway et al. [36]. 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 the information about the update and the updated data. The output is an encoded version of the updated data, as well as corresponding information, and a new metadata. The client sends this update information and new 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 server.
- $\mathsf{Verify}(\mathsf{sk}, \mathsf{pk}, M_c, c, P) \rightarrow \{\mathsf{accept}, \mathsf{reject}\}$ 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.