Abstract of "Efficient Cryptography for the Next Generation Secure Cloud" by Alptekin Küpçü, Ph.D., Brown University, May 2010.

Peer-to-peer (P2P) systems, and client-server type storage and computation outsourcing constitute some of the major applications that the next generation cloud schemes will address. Since these applications are just emerging, it is the perfect time to design them with security and privacy in mind. Furthermore, considering the high-churn characteristics of such systems, the cryptographic protocols employed must be efficient and scalable. This thesis shows that cryptography can be used to efficiently and scalably provide security and privacy for the next generation cloud systems.

We start by describing an efficient and scalable fair exchange protocol that can be used for exchanging files between participants of a P2P file sharing system. In this system, there are two central authorities that we introduce: the *arbiter* and the *bank*. We then try distributing these entities to reduce trust assumptions and to improve performance. Our work on distributing the arbiter leads to impossibility results, whereas our work on distributing the bank leads to a more general cloud computation result showing how a boss can employ untrusted contractors, and fine or reward them. We then consider cloud storage scenario, where the client outsources storage of her files to an untrusted server. We show how the client can challenge the server to prove that her file is kept intact, even when the files are dynamic. Next, we provide an agreement protocol for a dynamic message, where two parties agree on the latest version of a message that changes over time. We then apply this agreement protocol to the cloud storage setting and show how a judge can arbitrate between the client and the server officially based on the agreed-upon message and the proof sent by the server. Lastly, we show that all our solutions are efficient and scalable by presenting results from the cryptographic library we implemented.

# Efficient Cryptography for the Next Generation Secure Cloud

by

Alptekin Küpçü

B.S., Bilkent University, 2004

M.Sc., Brown University, 2007

A dissertation submitted in partial fulfillment of the
requirements for the Degree of Doctor of Philosophy
in the Department of Computer Science at Brown University

Providence, Rhode Island

May 2010

This dissertation by Alptekin Küpçü is accepted in its present form by the Department of Computer Science as satisfying the dissertation requirement for the degree of Doctor of Philosophy.

Date _____ _____

Prof. Anna Lysyanskaya, Advisor

Recommended to the Graduate Council

Date _____ _____

Prof. Yevgeniy Dodis, Reader
NYU

Date _____ _____

Prof. John Jannotti, Reader

Date _____ _____

Prof. Roberto Tamassia, Reader

Approved by the Graduate Council

Date _____ _____

Sheila Bonde
Dean of the Graduate School

# Vita

I, Alptekin Küpçü, was born in Ankara, Turkey, in 1983. I graduated with a B.S. degree from Bilkent University Department of Computer Engineering in 2004, with $3^{rd}$ rank. Then, I continued my studies at the Department of Computer Science at Brown University. I received my M.Sc. degree in 2007, under Uğur Çetintemel's supervision. During my Ph.D., I worked under the supervision of Anna Lysyanskaya, and worked together with John Jannotti, Roberto Tamassia, and Yevgeniy Dodis.

Throughout my life, I received multiple scholarships, fellowships, and awards. I am going to mention only some of those here. Due to my success at the university entrance examination in Turkey (SAT equivalent), I was awarded the "$76^{th}$ year anniversary prize" by Türkiye İş Bankası, an award given to the first 76 out of roughly 1.5 million contestants. Later, at the graduate education examination in Turkey (GRE equivalent), I was ranked $2^{nd}$ out of roughly 100 thousand attendees. Apart from awards for my academic success, I was also awarded a $3^{rd}$ rank prize at a poem competition in high school.

I am an active person, involved in many organizations ranging from computer science related activities to gaming clubs, involving groups of various sizes, ranging from small groups of friends to the whole graduate student community at the university, organizing events whose participants were from all over the nation, and sometimes even internationals coming from abroad.

# Acknowledgements

I want to thank my advisor Anna Lysyanskaya, and my committee John Jannotti, Roberto Tamassia, and Yevgeniy Dodis, for all their help throughout my Ph.D. I also would like to acknowledge and thank my academic siblings Mira Belenkiy and Melissa Chase, and all my other collaborators: C. Chris Erway, Charalampos Papamanthou, Theodora Hinkle, Sarah Meiklejohn, and Eric Rachlin. I further thank all other Brownie Points project members: Gabriel Bender, Saurya Simha Velagapudi, Jason Rassi, Paul O'Leary McCann, Diana Kathleen Huang, Joshua Kossoy Fuhrmann, and Alex Hutter. And I must mention friends and colleagues who helped me with at least one of my papers: Nevzat Onur Domaniç, İbrahim Eden, Semiha Ece Kamar, Markulf Kohlweiss, Hatice Şahinoğlu, and Meinolf Sellmann. Finally, I thank my master's advisor Uğur Çetintemel for his continued support even after my M.Sc. studies.

# Dedication

To my true friends. They know who they are...

# Contents

⋆ Chapters 2 and 3 of this thesis is joint work with Anna Lysyanskaya in papers [96, 95]. Chapter 4 is joint work with Mira Belenkiy, Melissa Chase, C. Chris Erway, John Jannotti, and Anna Lysyanskaya in paper [15]. Chapter 5 is joint work with C. Chris Erway, Charalampos Papamanthou, and Roberto Tamassia in paper [70]. Chapter 7 is joint work with C. Chris Erway, Theodora Hinkle, Anna Lysyanskaya, and Sarah Meiklejohn in paper [68]. Lastly, Appendix A includes contributions by Theodora Hinkle, Anna Lysyanskaya, Sarah Meiklejohn, and Charalampos Papamanthou.

# List of Tables

# List of Figures

# Chapter 1

# The Next Generation Secure Cloud

Systems researchers are designing cloud systems addressing various needs including cloud storage, cloud computation, collaborative systems, and many other peer-to-peer systems. We have seen many problems with integrating security after the design (as an example, consider the relative failure of adoption of IPSec on the Internet). Therefore, security must be included in the initial design of such next generation systems.

This thesis addresses security of some of the emerging cloud schemes like outsourced computation and storage, as well as state-of-the-art peer-to-peer (P2P) systems like BitTorrent file sharing. It mainly focuses on two different types of cloud systems. One type is focused on outsourcing some work to a more powerful entity (e.g., outsourcing storage to Amazon S3), whereas the other type's focus is outsourcing to multiple small entities (e.g., outsourcing distribution of files to users of a peer-to-peer system). In both settings though, the current generation systems provide no guarantees at the service level. This thesis shows that security guarantees in various cloud systems can be provided using cryptographic methods without losing efficiency and scalability. Furthermore, privacy is an important feature for the users, and so our protocols are designed to respect privacy of the participants.

Research presented in this thesis is mainly inspired by the Brownie Points project at Brown University [34]. As a member of this group, I first worked on employing electronic cash in a P2P file sharing system called BitTorrent [52] to provide accountability, fairness, and fault tolerance while preserving privacy [16]. In a BitTorrent system, peers would like to exchange files among each other. Current BitTorrent systems fail to provide strong accountability and incentives to share files, and hence many peers stop uploading as soon as they finish their own downloads.

Our group proposed using electronic cash (e-cash) as a method of solving these problems. E-cash, introduced by Chaum [50], is an efficient [40, 44] and anonymous payment system. A user can withdraw from the bank a wallet containing multiple e-coins. Then, users can exchange those e-coins (e.g., buy items), and they go deposit to the bank the coins they earned. The basic idea of using e-cash in BitTorrent involves paying an e-coin to download a block (BitTorrent treats files as composed of many blocks), and earning an e-coin for uploading a block. Since e-cash is used as the payment mechanism, peers will preserve their privacy in those transactions.

The transactions between the peers need to be fair, efficient, and scalable, considering the needs of the high-churn P2P system. There are two basic types of transactions between the peers: Exchanging a file (more precisely, a BitTorrent block) with an e-coin (buy transaction), or exchanging a block with another block (barter transaction). Our group presented fair exchange protocols for both transaction types [16]. Yet, initial implementation results made it clear that those protocols will not be efficient enough for a high-churn (high-interaction) P2P file sharing application. The problem is that all previously known fair exchange protocols (including the ones in [16]) required costly cryptographic primitives for every exchange. Considering that BitTorrent peers exchange thousands of blocks per file, the overhead becomes very high.

## 1.1 Fairness in the Cloud

Fair exchange is an everyday problem in which two parties, let us call them Alice and Bob, would like to exchange some items, and they want to do so fairly: either both parties obtain each other's item, or neither does. In addition to fairness, we are concerned with performance (efficiency and scalability) of the resulting cryptosystem, since the poor performance of the previous fair exchange protocols is one of the main reasons they are not widely used today. It has been shown that no fair exchange can be performed without a trusted third party (the arbiter) [122]: Without loss of generality, Alice will send the last message. In case she chooses not to, Bob will be at a disadvantage, unless a third party can provide fairness somehow, based on messages Alice and Bob have exchanged previously. Yet, with a trusted arbiter, fairness is trivial: both parties can send their items to the arbiter, and the arbiter sends Alice's item to Bob and Bob's item to Alice. Of course, this puts too much load on the arbiter, and therefore is impractical. Hence, recent previous work focused on *optimistic* fair exchange protocols in which the arbiter gets involved only in case of a dispute between the two parties [7].

Chapter 2 presents the most efficient barter protocol known [96]. Our first contribution is definitional. Previous work defined fair exchange of *signatures*. We extended this definition and formally defined general fair exchange of *arbitrary data*. As our second contribution, we came up with protocols that takes optimistic fair exchange protocols from being only theoretical to being completely practical. The main difference of our protocol is that the costly step is necessary only once per peer, and then the two peers can exchange as many blocks as they want using only efficient cryptography. Considering thousands of blocks being exchanged with only tens of peers, the improvement is pretty significant. When two parties exchange 2.8GB of data, previous work requires an additional 250MB of communication and 84 minutes

of computation, whereas our protocol's overhead is about 2MB of communication and 80 seconds of computation, using state-of-the-art building blocks. Our protocols have more efficiency gains in particular when the same parties keep exchanging multiple items, even when they do not know how many or which items they will end up exchanging ahead of time. This improvement is possible since the protocol assumes that buying is considered fair in case bartering fails, which perfectly fits our BitTorrent scenario.

## 1.2 Trust in the Cloud

As we mentioned above, every fair exchange protocol requires a trusted third party (called the *arbiter*) [122]. Every trusted entity is a potential weakness of a system, and thus in Chapter 3 we considered techniques to relax the trust assumptions by distributing the job of the trusted arbiter among multiple parties [95]. This can be done using previous work on Byzantine fault tolerance and secure multi-party computation [81, 22, 24, 47], but these lead to inefficient solutions whose communication and computation complexity are quadratic in the number of parties used as the distributed arbiter. Thus, we focused on employing multiple autonomous arbiters who do not talk to each other. We proved that for a general class of optimistic fair exchange protocols, having multiple autonomous arbiters requires trusting all arbiters, and hence reducing the trust this way is impossible.

## 1.3 Computation in the Cloud

In a system that involves e-cash, there is a need for a bank to keep account balances, let users withdraw and deposit coins. With each deposit, the bank needs to verify the coin to see if it is valid. In a high-churn P2P system as above, the bank may get

overloaded, so one needs to lighten the burden of the bank. One way of doing this is the barter protocol we mentioned above, since no money changes hands if everything goes well. If peers can barter instead of buying, this is great for the bank. Moreover, it is also beneficial for the peers since when multiple blocks are exchanged, bartering is much more efficient than buying. An additional way of lightening the burden of the bank is to outsource the bank's job to untrusted contractors. Instead of forcing a central authority to handle all the work, we can outsource the bank's coin verification job to P2P system users to mitigate this bottleneck.

Motivated by outsourcing the bank's verification job, in Chapter 4 we present a novel outsourcing mechanism for a boss employing untrusted contractors for computation [15]. The idea is that the boss rewards the contractors who do their job correctly and punishes the dishonest ones. He can employ multiple contractors for the same computation and use their answers to check each other. We split the contractors into three categories (honest, rational, malicious), and analyzed their incentives in a game-theoretic setting. Malicious contractors are not rational, but they still need to keep a non-negative balance between rewards and punishments to remain employed. Their goal is to maximize the damage to the system by making the boss accept wrong answers or waste resources on re-doing the computation. Our setting still provides meaningful correctness and reliability guarantees even when, for example, more than half of the users are malicious (as opposed to other lines of work such as Byzantine fault tolerance or secure multi-party computation [81, 22, 24, 47]). We provide the first and only scheme that has parameterized graceful tolerance against malicious users.

## 1.4 Storage in the Cloud

Another outsourcing scenario in the cloud involves storage instead of computation. In such a scenario, a client would like to store her data at an untrusted storage server. Current systems like Amazon S3 are highly popular, and yet there is no way in the S3 deployment to prove that Amazon keeps your file intact. Ideally, the server should be able to present a cryptographic proof to the client.

Previous work in this area showed how the server can prove that the data is kept intact, but only for archival files (files that do not change) [9, 90, 139]. If a user wants to modify her file on the server, she needs to download the whole file, modify it, and put it back on the server as a new file. This obviously puts an impractical load on both the client and the server.

In Chapter 5, we address this problem by providing the first completely dynamic solution in which the client can efficiently modify her file in any way she wishes by downloading and uploading only the portions of the file that are affected [70]. We also give the first security definition for such a dynamic scenario. We furthermore extend our basic construction that works on a single file and showed how to construct provable file systems and versioning systems (e.g., CVS). Our scheme is very efficient in practice with only about 430KB of communication and about 30ms of computation overhead for a 1GB file. Previous work on online memory checking suggests that our scheme is almost asymptotically optimal (with a $\log \log n$ factor, where $n$ is the number of blocks in a file), although no full proof is given yet.

## 1.5 Judging in the Cloud

Unfortunately, such a system is practically useless unless the client or the server can prove anything to a trusted judge in case of a dispute about the integrity of the

file. Previous work on archival storage does not provide public verifiability that can be used for official purposes; an honest server cannot prove its innocence against a malicious client's accusation [9]. Our solution to this problem in the static storage scenario is to employ fair exchange of signatures on a contract specifying the file and any public keys involved. The judge then rules based on the contract signed by both parties and the proof given by the server.

For the dynamic case, a naïve idea would be performing a fair exchange with each update. Since the file is modified, there needs to be a new contract each time. Chapter 6 presents a very efficient public verifiability protocol that does not require a full-fledged fair exchange with each update; it uses only efficient cryptography (signatures) [71]. Compared to the naïve version (which is the only existing two-party solution), this protocol requires milliseconds instead of seconds, and bytes instead of tens of kilobytes, per update. Besides, this public verifiability protocol can be used by a judge for official arbitration, and can incorporate automatic payments in such cases, through use of electronic payments like e-cash.

The protocol in Chapter 6 is actually a more general agreement protocol for dynamic data. The goal is to make sure two parties agree on the latest version of a message that keeps changing over time. For our protocol to work, there must be a means to prove and verify that the message is correctly formed according to some rules in a contract. In the case of outsourced storage, the message is some metadata kept at both the client and the server. The server can prove that the metadata is correct as presented in Chapter 5, and this can be verified by a trusted authority.

## 1.6 Implementing the Cloud

Implementing cryptographic protocols is a hard task. In general, cryptographic protocols are fairly complicated in design, and as such their implementation requires

great care. Unfortunately, one cannot expect all good cryptographers to be good programmers, or vice versa. As a result, many useful cryptographic protocols go unimplemented, despite the fact that they may be efficient enough to be used in practice.

Throughout this thesis, we provide protocols that are efficient and scalable enough to be used in real systems. As part of the Brownie Points project, our group is building a real BitTorrent client, incorporating the cryptographic protocols in this thesis, to analyze the effects when larger numbers of users are involved [69]. Chapter 7 discusses our implementation of the protocols described in this thesis.

We have developed a cryptographic language that lets users to easily specify discrete-logarithm based zero-knowledge proofs that are at the heart of many state-of-the-art constructions such as blind signatures, e-cash and verifiable encryption (see Appendix A). Our goal is to make coding cryptographic protocols as easy as describing a cryptographic protocol on paper. Therefore, our language closely resembles an academic paper format. We have further developed a cryptographic library that implements e-cash and verifiable encryption on top of these zero-knowledge proofs, and fair exchange on top of e-cash and verifiable encryption. This library is already being used in a P2P file sharing system deployment [34]. As a consequence of the efficiency of our fair exchange protocols and our usable implementation, our colleagues at Brown University were able to deploy, for the first time, a peer-to-peer file sharing system that enjoys fairness, while preserving the privacy of the participants [69].

## 1.7 Organization

The organization of this thesis follows the organization of the introduction. We touch many aspects of security in the cloud, including three main resources: network, computation, and storage. We further analyze trust issues arising in the cloud, provide

means to settle disputes, and discuss distributing trusted parties. We also provide implementation results. Finally, the reader may find the Appendix very handy, since it is a comprehensive compilation of pseudocodes of state-of-the-art primitives.

# Chapter 2

# Networking in the Cloud

## 2.1 Introduction

Fairly exchanging digital content is an everyday problem. A fair exchange scenario commonly involves Alice and Bob. Alice has something that Bob wants, and Bob has something that Alice wants. A fair exchange protocol guarantees that at the end either each of them obtains what (s)he wants, or neither of them does (see [111] for more details and examples).

In this chapter, we consider a general file exchange (bartering) scenario, inspired by the BitTorrent [52] peer-to-peer file sharing protocol. Alice has several files (BitTorrent blocks) of interest to Bob, and Bob has several files (blocks) of interest to Alice. They do not know ahead of time how many or which blocks they will end up exchanging. They want to perform a fair exchange: Alice should get Bob's file (block) if and only if Bob gets Alice's file (block). In a signature fair exchange [7, 6, 5], there is a verification mechanism (i.e., the public key) that enables the sender to verifiably encrypt the signature so that the receiver can check that the encrypted signature verifies. No such efficient verifiable encryption method is currently known for exchanging files. Therefore, a compensation is required after the fact if one of the parties cheat.

In our scenario, we are assuming that Alice/Bob will be equally happy to get a payment in return to her/his file. Thus, exchanging a file with a payment (buying) is also considered fair, as in some previous works [7, 16, 44, 107, 102].

One of the hardest points in creating a usable optimistic fair exchange protocol suitable for P2P file sharing applications is that the peers to contact and the content to exchange are not pre-defined. BitTorrent clients keep connecting to different peers to obtain different blocks. Fault-tolerance issues, connectivity problems, and availability of data blocks are all factors affecting from whom which block should be obtained. Our protocol uniquely addresses these issues by removing the need to know what content to exchange with whom beforehand.

In a nutshell, in our protocol, Alice sends a verifiable escrow of a payment (e.g., e-coin) to Bob first. Then, they exchange encrypted files. Afterward, Alice sends Bob an escrow of her key with her signature on the escrow. Then, Bob sends Alice the key to his file. Finally, Alice sends Bob the key to her file. Since Bob has a verifiable escrow of an e-coin and an escrow of a key before he sends his key to Alice, he is protected. In the worst case, if Alice does not provide the correct key and the key escrow contains garbage, Bob can go to the Arbiter and obtain Alice's payment. The escrow of the payment cannot contain garbage, because it was formed using a *verifiable* escrow. After the exchange of the verifiable escrow, the rest of our protocol can be repeated as many times as necessary to exchange multiple files (even if the number and content of the files were not known in advance), unless there is a dispute.

We provide two versions of the protocol: In the first one (the one described briefly above) only one party provides a verifiable escrow. This version requires the use of timeouts for dispute resolution purposes. We provide another version that needs both parties to provide verifiable escrows but requires no timeouts. Both versions are very efficient since they use only *one* (resp. *two*) expensive primitives (verifiable escrow

and payment) regardless of the number of files exchanged. We stress the fact that our timeouts can be very large (e.g., one day or week) to allow for unexpected situations in which the participants act honestly (e.g., network failure), and thus require very loose synchronization (e.g., one hour difference), and users can freely participate in other exchanges without waiting for the timeout.

### 2.1.1 Previous Work

It is well-known that a fair exchange protocol is impossible without a trusted third party (TTP) [122] (called the *Arbiter*) that ensures that Alice cannot take advantage of Bob, and vice versa. Without loss of generality, Alice will have to send the last message of the protocol, and we want to protect Bob in case she chooses not to do so. Without an arbiter, gradual release type of protocols where parties send pieces to each other in rounds can provide only weaker forms of fairness, and are much less efficient [23, 30].

Luckily, the impossibility result [122] does not require that the Arbiter be involved in each transaction, but simply that the Arbiter exists. If Alice and Bob are both well-behaved, there is no need for the Arbiter to do anything (or even know an exchange took place). Micali [110], Asokan, Schunter and Waidner [5], and Asokan, Shoup and Waidner [7, 6] investigated this *optimistic* fair exchange scenario in which the Arbiter gets involved only in case of a dispute. Two such protocols [7, 78] were analyzed in [141] (see also [12]).

Asokan, Shoup and Waidner (ASW) [7] gave the first provably secure and completely fair optimistic exchange protocol for exchanging digital signatures. Later on, Belenkiy et al. [16] gave a protocol for buying digital content in exchange for e-cash, building on top of the ASW protocol. They provided an optimization for the Arbiter so that, unlike in the ASW protocol, the amount of work that the Arbiter is required

to do depends only logarithmically on the size of the file. They also assume there is an additional TTP (which we call the *Tracker*) that provides a means of verification that the file actually contains the right content (e.g., using hashes). Such entities certifying hashes already exist in current BitTorrent systems [52].

Belenkiy et al. [16] used e-cash (introduced by Chaum [50]), in particular, endorsed e-cash [44] in their constructions. The reason is that other forms of payments (signatures or electronic checks used in [7, 107]) do not provide any privacy. In our protocols, any form of payment can be employed, but we will also use endorsed e-cash in our sample instantiation since it is efficient and anonymous. See Section 2.7 for more discussion on employing different payment systems.

### 2.1.2 Contributions

We present the most efficient fair exchange known to us, where *the efficiency is comparable to a simple unfair exchange if performed multiple times between the same pair of users, even when peers do not know beforehand which blocks they will end up exchanging.* Using the best previous work (Belenkiy et al. barter protocol [16]), $n$ pairs of blocks can be exchanged using $n$ transactions, each of which requires a costly step involving expensive cryptographic primitives (a verifiable escrow and an e-coin). Our contribution is a very efficient fair exchange protocol using which this can be done with only *one* (or *two* if we do not want to employ timeouts) step in total that involves the same expensive primitives (verifiable escrow and payment). This is a property that is unique to our protocol: Instead of employing the costly primitives for every file or block that is exchanged, we employ them once per peer, even when peers do not know beforehand which blocks they will end up exchanging. Then, exchanging multiple files/blocks between peers involves only very efficient cryptography (i.e., symmetric- and public-key encryption, and digital signatures). In a real

setting where BitTorrent peers exchange thousands of blocks with only tens of peers, there is *one or two orders of magnitude improvement in terms of both computation and communication* (40 seconds vs. 42 minutes computational overhead and 1.6MB vs. 200MB communication overhead for a 2.8GB file —for detailed numbers, see Section 2.8). This means that, with no (i.e., neglectable) efficiency loss, our fair exchange protocol can be used to exchange files instead of the unfair protocol currently used by BitTorrent or similar file sharing protocols.

We stress the fact that the timeouts used for dispute resolution purposes in one of our protocols can be very large (e.g., one day or week) to allow for unexpected situations in which the participants act honestly (e.g., network failure), and thus require very loose synchronization (e.g., one hour difference), and *users can freely participate in other exchanges without waiting for the timeout*.

We take the idea of using verifiable escrow from ASW [7], and the subprotocols of Belenkiy et al. [16] that increase the efficiency of the Arbiter (see Section 2.4.3). The Arbiter does absolutely no work in our protocols, as long as no dispute occurs. *Our protocols can make use of any type of payments*, but we will show an instantiation using e-cash since it also provides privacy. Our performance evaluation numbers will use endorsed e-cash [44] as the payment mechanism. Note that other (non-anonymous) forms of payments (e.g., electronic checks [49]) will be more efficient.

Our additional contribution is definitional. We give a general definition of fair exchange of digital content (not just digital signatures) provided that it can be verified using some verification algorithm (defined in Section 2.3). Furthermore, our fairness definition covers polynomially many exchanges between an honest party and an adversary controlling polynomially-many other participants (see [63] for an example fair exchange protocol that is fair for a single exchange but stops being fair in a multi-user setting). We then prove our protocol's security based on this definition. We sum up

the most important properties of our protocols below.

**Security of our protocol**: Our protocols provably satisfy the following condition (waiting for at most one timeout period if timeouts are used, or without waiting at all if no timeouts are used), as long as at least one of the trading parties (Alice and Bob) is honest:

- Either Alice and Bob both get their corresponding files,
- Or Alice gets Bob's file and Bob gets Alice's payment (turns into a buy protocol in effect),
- Or neither of them gets anything.

**Efficiency of our protocol**: We have the following properties regarding efficiency:

- An honest user can reuse her e-coin for other exchanges without waiting for the completion of the protocol.
- The overhead of our costly step – verifiable escrow and e-cash – is constant $O(1)$, instead of linear $O(n)$ as in previous best results, when $n$ files or blocks are exchanged.

Already, the Brownie Project [34] is using our protocols in their BitTorrent deployment. We discuss the efficiency of our protocols and our initial implementation results in Section 2.8. Discussion of limitations and future work can be found in Section 2.9.

## 2.2 Notation

Barter is an exchange of two items, which are digital files in our case. We assume that the reader is familiar with encryption and signature schemes, and hash functions.

Further required definitions and notation are given below, although partially, omitting the details not necessary for understanding the following fair exchange protocols.

An escrow is a ciphertext under the public key of some trusted third party (TTP). A *verifiable* escrow [7, 45, 37] means that the recipient can verify that the contents of the ciphertext satisfy some relation (therefore stating that the ciphertext contains the expected content). A contract (a.k.a. label, condition, or tag) attached to such a ciphertext defines the conditions under which the TTP should decrypt and give away the encrypted secret [143]. The label is public and it is integrated with the ciphertext in a such way that it cannot be modified. We will use $E_{Arb}(a; b)$ to denote an escrow of the secret $a$ under the Arbiter's public key, with the contract $b$. Similarly, $VE_{Arb}(a; b)$ will denote a verifiable escrow.

Any payment protocol that can efficiently be verifiably escrowed and is secure can be used in our protocols. Furthermore, if privacy is desired, the payments should be anonymous as in e-cash [50]. We provide an instantiation using endorsed e-cash [44] (which is an extension of compact e-cash [40]), since it satisfies all these requirements. Endorsed e-cash splits a coin into an unendorsed coin (denoted $coin'$) and endorsement (denoted $end$). One can think of $coin'$ as an encrypted coin and $end$ as the key. One can check if the endorsement $end$ in a given verifiable escrow [45] matches the given unendorsed coin $coin'$ (without learning the endorsement $end$). Furthermore, given only the unendorsed part $coin'$, no other party (except the owner) can come up with a valid endorsement $end$. Endorsed e-cash moreover has the ability to catch double-spenders. Hence, if one uses two different $coin', end$ pairs trying to spend the same coin twice, (s)he will be caught (and, since her identity is revealed, can be punished). Note that if a party tries to deposit the same coin twice (using the same $coin', end$ pair), the operation can easily be denied by checking against a list of past transactions. Lastly, only matching $coin', end$ pairs can be linked, unendorsed coins

and endorsements prepared for different exchanges remain unlinkable.

Wherever used, $K_P$ will denote a symmetric key of a party $P$, generated through an encryption scheme's key generation algorithm. We let $c = Enc_K(f)$ denote that the ciphertext $c$ is an encryption of the plaintext $f$ under the symmetric key $K$. Similarly, $f = Dec_K(c)$ will denote that the plaintext $f$ is the decryption of the ciphertext $c$ under the symmetric key $K$. Our protocol can make use of any secure symmetric encryption scheme (see the book by Katz and Lindell [93] for definitions and constructions).

Let $pk_P$ and $sk_P$ denote public and secret keys for a party $P$. Then $sign_{sk}(x)$ will denote a signature on $x$ under the secret key $sk$ which can be verified using the corresponding public key $pk$. Our protocol can make use of any secure public-key encryption scheme [56, 66] and any secure signature scheme [84].

Furthermore, let $H_k$ be a family of (universal one-way) hash functions [118], where $k$ is the security parameter, and let *hash* be a hash function uniformly choosen from the family $H_k$ of hash functions. Then, $h_x = hash(x)$ will denote that $h_x$ is the hash of $x$ under the hash function *hash*. We now introduce a definition we frequently use in this chapter.

**Definition 2.2.1.** *We say that a key $K$* ***decrypts correctly****, or is the* ***correct key*** *with respect to a plaintext hash $h_f$ and a ciphertext $c$, if the plaintext $f' = Dec_K(c)$ has the property $hash(f') = h_f$.*

Finally, a negligible probability denotes a probability that is a negligible function of the security parameter (e.g., the key-length of an encryption scheme). A negligible function of $n$ is a function which is smaller than any inverse polynomial over $n$ with $n > N$ for sufficiently large $N$ (e.g., $neg(n) = 2^{-n}$). A non-negligible probability is a probability that is not negligible.

## 2.3 (Optimistic) Fair Exchange

In this section we will give a general definition of fair exchange. Unlike in ASW, our definitions will not be specific to signature exchange, and we will consider polynomially-many exchanges between an honest user and an adversary controlling polynomially-many other users. Furthermore, we separate and clearly define the roles of all trusted parties. While providing models and definitions for a general framework of (optimistic) fair exchange applicable to a broad range of protocols, we will also show its extensions to our case.

MODEL: The model is adapted from the ASW definition [7], with clarifications and generalizations. There are three players; *Alice* and *Bob* exchanging two digital items, and the *Arbiter*[1] for conflict resolution. All players are assumed to be polynomial time interactive Turing machines. We make no assumption about the underlying network capability.[2] Any message that does not confirm with the protocol specification will be discarded by the honest parties. Any input which does not verify according to the protocol will be resolved as stated by the protocol or the protocol will be aborted if no resolution is applicable. It is important that the Arbiter resolves conflicts on the same exchange *atomically*.[3] Thus, it will only interact with either Alice or Bob at any given time instance, until that interaction ends as specified by the protocol.[4] Sensitive communication (e.g., exchange of decryption keys for files or endorsement of an e-coin) will be carried out over a secure (and possibly authenticated) channel (e.g., SSL can be used to connect to the Arbiter, a secure key exchange with no public

[1] One of the TTPs in ASW.

[2] Clients will have a local *message timeout* mechanism like the TCP timeout, which is small (e.g., one minute). The receiver deals with a *message timeout* exactly as it would deal with a non-verifying input.

[3] We present a trade-off between non-atomicity and performance of the Arbiter later on.

[4] For ease of the Arbiter to find the correct exchange, a random exchange ID can be incorporated into the messages. Since this is only a minor implementation efficiency issue, we do not want to complicate our definitions with that.

key infrastructure can be used for the communication between Alice and Bob).

For protocols using a *timeout*[5], we assume that the adversary cannot prevent the honest party from reaching the Arbiter before the timeout. If no timeouts are defined, we assume the adversary cannot prevent the honest party from reaching the Arbiter eventually. Hence, the honest party is assumed to be able to reach the Arbiter as defined by the protocol. Even with timeouts, this is not an unrealistic assumption since our timeouts can be large (e.g., one day or week).

In our model, we have two additional players, namely the *Tracker* (also in [7, 16, 52])[6] providing verification algorithms, and the *Bank* dealing with monetary parts of the system.

Setup Phase: Before the fair exchange protocol is run, we assume there is a setup phase. In this one-time pre-exchange phase, the Arbiter generates his public-private key pair (for the (verifiable) escrow schemes) and publishes his public key(s) so that both Alice and Bob obtain it. Optionally, the Arbiter may learn public keys of Alice and Bob in the setup phase, but our focus is on the case where the Arbiter does not need to know anything (and learns almost nothing, see Section 2.5.2) about Alice or Bob. *The adversary cannot interfere with the setup phase.*[7] In the setup phase, the Bank and the Tracker also generate their public-private key pairs and publish their public keys.

**Definition 2.3.1.** *Let* $\mathsf{SP}$ *denote the security parameters of the system (e.g., key lengths of the primitives used). Let PP denote all the public values in the system, including* $\mathsf{SP}$*, public keys of the trusted parties, and possibly some public parameters.*

[5]This is not the *message timeout*, it is the *timeout* specified by the protocol, which is generally much longer (e.g., one day or week).

[6]ASW has the corresponding TTP in their file exchange scheme. In their signature exchange protocol, the public key infrastructure providing the public keys can be seen as the Tracker.

[7]This is the standard trusted setup assumption that says Alice and Bob have the correct public key of the Arbiter.

*Let* $\mathsf{PPGen}(\mathsf{SP})$ *be the randomized procedure which generates the public values given the security parameters. Then, define our* $PP = (pk_{arb}, pk_{bank}, pk_{tracker}, timeout, \mathsf{SP}$*, and additional parameters for primitives used).*

From now on, we need to talk about multiple exchanges taking place. Alice has files $f_A^{(1)}, .., f_A^{(n)}$ to be exchanged with Bob, and Bob has $f_B^{(1)}, .., f_B^{(n)}$ to be exchanged with Alice ($n$ is a polynomial in $\mathsf{SP}$).[8] In general, we can consider these files as some strings in $\{0,1\}^*$, therefore consider fair exchange of anything that is verifiable. Without loss of generality, the Tracker gives Alice a *verification algorithm* $V_{f_B^{(i)}}$ for each file $f_B^{(i)}$, and Bob a verification algorithm $V_{f_A^{(i)}}$ for each file $f_A^{(i)}$ before the exchange takes place.

Assume that the content to be exchanged and associated verification algorithms are output by a generation algorithm $\mathsf{Gen}(\mathsf{SP})$ that takes the security parameters as input and outputs some content to be exchanged, with associated verification algorithms, and possibly some public information about the content. This procedure involves a trusted party $H$ and the Tracker. The parties trust the Tracker in that any input accepted by that verification algorithm will be the content they want. In other words, they are going to be happy with any content that verifies under that verification algorithm. In particular, the content generation process is trusted. The adversary cannot generate "junk" files and ask the Tracker to create verification algorithms for them. BitTorrent forum sites and ratings provide a level of defense against this in practice.

**Definition 2.3.2.** *Content and verification algorithms are secure if* $\forall$ *PPT adversaries* $\mathcal{A}$ *and* $\forall$ *auxiliary inputs* $z \in \{0,1\}^{poly(\mathsf{SP})}$ *we have (over the randomness of the generation algorithms, the adversary, and possibly the verification algorithms)*

[8] Note that Alice or Bob can represent multiple entities controlled by the adversary.

$$Pr[PP \leftarrow \mathsf{PPGen}(\mathsf{SP}); (f_H^{(1)}, V_{f_H^{(1)}}, pub_{f_H^{(1)}}, .., f_H^{(n)}, V_{f_H^{(n)}}, pub_{f_H^{(n)}}) \leftarrow \mathsf{Gen}(\mathsf{SP});$$
$$(f_{\mathcal{A}}^{(1)}, .., f_{\mathcal{A}}^{(n)}) \leftarrow \mathcal{A}(V_{f_H^{(1)}}, pub_{f_H^{(1)}}, .., V_{f_H^{(n)}}, pub_{f_H^{(n)}}, PP, z) :$$
$$\exists i \in [1..n] \quad | \quad (V_{f_H^{(i)}}(f_H^{(i)}) \neq \mathsf{accept} \vee V_{f_H^{(i)}}(f_{\mathcal{A}}^{(i)}) = \mathsf{accept})] = \ neg(\mathsf{SP})$$

The definition above models the case in which the files to be exchanged cannot be found by the adversary by some other means[9] (and hence exchanging files makes sense for the adversary), even with the help of associated verification algorithms and public information[10].

To provide evidence on the generality and applicability of our definition, we present several example verification algorithms for various tasks. For example, a file verification can be performed using hashes. So, each verification algorithm $V_{f_A^{(i)}}$ for Alice's file $f_A^{(i)}$ contains the definition of hash function used –*hash*–[11], and the hash value $h_{f_A^{(i)}} = hash(f_A^{(i)})$. The $i^{th}$ verification algorithm computes the hash of the given input according to the description of the hash function, and accepts it if and only if the computed hash matches $h_{f_A^{(i)}}$ (see Section 2.5.1 for a security analysis). As another example, consider the ASW signature exchange protocol, in which each verification algorithm contains the signature scheme's description[11], the signature public key of Alice $pk_A$[11], and the message $m_i$ to be signed. When it receives a signature as input, the $i^{th}$ verification accepts the signature if and only if it is a valid signature on message $m_i$ under the public key $pk_A$ using the signature scheme. As yet another example, an e-coin verification algorithm can take a coin to verify, and use the Bank's public key while verifying the non-interactive proofs given. Such an algorithm is a part of the specification of every e-cash scheme (e.g., see [44, 40]). Verifiable encryption schemes

[9] We assume that the adversary cannot just "guess" an honest participant's file, in which case the exchange is trivially unfair.

[10] For example, if movies are being exchanged, a lot of information is publicly available about such a movie file, such as actors, length, and release date. But these do not enable people to come up those movie files.

[11] possibly different for each verification algorithm

(e.g., [45]) and, in general, proof systems also specify a verification algorithm in their definitions. Such algorithms can be used directly in a fair exchange protocol, satisfying our definition as long as they are secure according to Definition 2.3.2.

To summarize, in the setup phase, public values are generated using $\mathsf{PPGen}(\mathsf{SP})$. The files and the verification algorithms are generated jointly by the Tracker and some trusted content generator (e.g., movie distributor) using the $\mathsf{Gen}(\mathsf{SP})$ procedure. In the context of BitTorrent, this means that we trust the content generator about the content, and the Tracker about the verification algorithms. In practice, BitTorrent forum sites and ratings on files provide this trust. A "highly rated" BitTorrent user will be trusted about the content, or alternatively, comments on the forum sites will warn against bogus content. Besides, even the public information leaked from the generation procedure does not help the adversary. From now on, we assume the content and the verification algorithms used are secure and trusted.

**Definition 2.3.3.** ***Fair Exchange Protocol****: A fair exchange protocol is composed of three interactive algorithms: Alice running algorithm A, Bob running algorithm B, and the Arbiter running the trusted algorithm T. The content and verification algorithms used need to be secure according to Definition 2.3.2. The security of the exchange is then defined in terms of completeness (when Alice and Bob are both honest) and fairness (when either Alice or Bob is malicious).*

Completeness for a (non-optimistic) fair exchange states that the interactive run of $A$, $B$ and $T$ by *honest parties* results in $A$ getting $B$'s files and $B$ getting $A$'s files (assuming an ideal network):

$$Pr[(f_B^{(1)},..,f_B^{(n)}) \leftarrow A(f_A^{(1)},..,f_A^{(n)}, V_{f_B^{(1)}},..,V_{f_B^{(n)}}, PP) \overset{T(sk_{arb})}{\longleftrightarrow} B(f_B^{(1)},..,f_B^{(n)}, V_{f_A^{(1)}},..,V_{f_A^{(n)}}, PP) \rightarrow (f_A^{(1)},..,f_A^{(n)})] = 1$$

where the notation describes that $A$, $B$ and $T$ can all communicate (in a three-way interaction) following the protocol, and at the end $A$ outputs $f_B^{(i)}$ and $B$ outputs $f_A^{(i)}$ for all $i : 1..n$.

Optimistic Completeness for an optimistic fair exchange states that the interactive run of $A$ and $B$ by *honest parties* results in $A$ getting $f_B^{(i)}$ and $B$ getting $f_A^{(i)}$ for all $i : 1..n$ (the Arbiter's algorithm $T$ is not involved, assuming an ideal network). A protocol satisfying optimistic completeness also satisfies completeness. Our *optimistic completeness* definition is:

$$Pr[(f_B^{(1)}, .., f_B^{(n)}) \leftarrow A(f_A^{(1)}, .., f_A^{(n)}, V_{f_B^{(1)}}, .., V_{f_B^{(n)}}, PP) \leftrightarrow B(f_B^{(1)}, .., f_B^{(n)}, V_{f_A^{(1)}}, .., V_{f_A^{(n)}}, PP) \rightarrow (f_A^{(1)}, .., f_A^{(n)})] = 1$$

Fairness states that at the end of the protocol, either Alice and Bob both get content that passes the verification algorithms given to them, or neither Alice nor Bob gets anything that passes the verification, in each of the $n$ exchanges, even when one of them is malicious.[12] This definition is easy to satisfy using a (non-optimistic) fair exchange protocol since Alice and Bob can both hand their files to the Arbiter, and then the Arbiter can send Bob's files to Alice and Alice's files to Bob, if they pass respective verifications. Thus, below, we will define the more interesting case; fairness for an *optimistic* fair exchange.

It is important to note that the ASW definition of fairness applies only to a single exchange, whereas our definition covers polynomially-many exchanges between an honest party and other players all controlled by the adversary. Even though we

[12] On the contrary, completeness definition only deals with honest participants.

define fairness in a symmetric way, during the security analysis one may need to consider two cases independently since the protocol can be asymmetric: the case where Alice is honest but Bob is malicious, and the case where Bob is honest but Alice is malicious.

FAIRNESS: We have an honest player $H$, and an adversarial player $\mathcal{A}$. The honest player runs algorithm $A$ in exchanges where he plays the role of Alice, algorithm $B$ in exchanges where he plays the role of Bob, and the Arbiter runs the algorithm $T$, all as defined by the protocol. $H$ has files $f_H^{(1)}, .., f_H^{(n)}$ to be exchanged with the adversary, and $\mathcal{A}$ has $f_{\mathcal{A}}^{(1)}, .., f_{\mathcal{A}}^{(n)}$ to be exchanged with $H$. The adversary is assumed to control all other players, and hence all interactions of the honest player are with parties controlled by the adversary, which is the worst possible scenario covering multiple exchanges.

First there is the trusted setup phase as explained above, getting the security parameters as input, generating secure content and verification algorithms, along with some associated public information, and giving the appropriate values to each party. Since the setup phase is trusted, $\forall i : 1..n\, V_{f_H^{(i)}}, V_{f_{\mathcal{A}}^{(i)}}, PP$ are trusted. Then parties proceed with the fairness game explained below, the honest party outputting $X$ and the adversary outputting $Y$. At the end of the game, we require the fairness condition holds on $X, Y$, the verification algorithms $V_{f_H^{(1)}}, V_{f_{\mathcal{A}}^{(1)}}, .., V_{f_H^{(n)}}, V_{f_{\mathcal{A}}^{(n)}}$, and the public values $PP$ with high probability against all PPT adversaries $\mathcal{A}$, and all polynomially-long auxiliary inputs.

$$Pr\ [\ \text{Setup; FairnessGame: FairnessCondition}\ ] = 1 - neg(\mathsf{SP})$$

FAIRNESS GAME: There are three types of interaction in our fairness game. Type 1 interactions are between $H$ and $\mathcal{A}$. Type 2 interactions are between $H$ and $T$.

Type 3 interactions are between $\mathcal{A}$ and $T$.[13] The adversary can arbitrarily interleave type $1, 2, 3$ interactions, but cannot prevent type 2 interactions from happening until the timeout if timeouts are used, or eventually otherwise. The game ends when the honest party $H$ produces its final output (including aborts and resolutions) in all the started protocols. Without loss of generality, in the fairness game we assume both parties want to exchange different content in different exchanges ($\forall i \neq j \quad f_H^{(i)} \neq f_H^{(j)}$ and $f_{\mathcal{A}}^{(i)} \neq f_{\mathcal{A}}^{(j)}$ and $\forall i, j \quad f_H^{(i)} \neq f_{\mathcal{A}}^{(j)}$).[14]

Fairness Condition: Recall that the honest party's output was $X$ and the adversary's output was $Y$ at the end of the fairness game. A general fairness condition would be $\forall i : 1..n \quad [\exists x \in X : V_{f_{\mathcal{A}}^{(i)}}(x) = \mathsf{accept} \Leftrightarrow \exists y \in Y : V_{f_H^{(i)}}(y) = \mathsf{accept}]$ meaning that either $H$ and $\mathcal{A}$ both get what they want or both don't, in each exchange.

Our protocol with payments has a very straightforward generalization of the fairness property. Our fairness condition states that either they both parties get each other's file, or one of them gets the other's file whereas the other gets his payment, or they both get nothing at each exchange. We believe that a broad range of optimistic fair exchange protocols can adapt the definition above using straightforward extensions whenever necessary.

Timely Resolution: Lastly, as pointed out by ASW [7], an optimistic fair exchange protocol must provide timely resolution: Alice and Bob must be able to have disputes resolved within a finite and limited time. In our protocol without timeouts, resolution is immediate. In our protocol with timeouts, we guarantee resolution at the timeout (which is finite and fixed). We furthermore show that timeouts do not

[13] In the implementation, $T$ may need to have a way to differentiate which one of Alice and Bob he is talking to, which can easily be done in our protocols without learning who Alice and Bob are. When necessary, using one-way function values whose pre-image is known by only one of the parties will suffice.

[14] If the honest party already has the adversary's file, the exchange will be trivially fair due to the completeness property. If the adversary already has the honest party's file, then there is no hope for fairness since the adversary can just abort the protocol but he already has the file. Similar arguments hold for exchanging the same file multiple times.

render our system less usable (Alice and Bob can freely participate in other exchanges without waiting for the timeout), and so in general we can use our more efficient protocol with timeouts.

We now present two different barter protocols, one that employs timeouts (Section 2.4), and one that does not (Section 2.6). Both of our protocols are $O(n)$ times more efficient than previous protocols [7, 6, 5, 8, 111, 16, 44], when $n$ files or blocks are exchanged, and almost as efficient as an unfair exchange, while still being provably fair.

## 2.4 Barter with Timeouts

We will show a particular instantiation of our protocol, using endorsed e-cash [44] as the payment and hashes as the file verification algorithms, and then point out how to generalize it easily, in Section 2.7. Before the protocol begins, we assume Alice has withdrawn an e-coin from the Bank. Every time Alice and Bob wants to exchange two files (every time before step 2 of the protocol below), Alice generates her fresh key $K_A$ and Bob generates his fresh key $K_B$ for a symmetric encryption scheme. Alice and Bob both have their files $(f_A, f_B)$, have the encrypted versions of their files $(c_A = Enc_{K_A}(f_A), c_B = Enc_{K_B}(f_B))$, have the hashes of their files and encryptions (Alice has $h_{f_A} = hash(f_A), h_{c_A} = hash(c_A)$, and Bob has $h_{f_B} = hash(f_B), h_{c_B} = hash(c_B)$). Besides, the Tracker provides them with the respective verification algorithms: Alice gets $h_{f_B}$, Bob gets $h_{f_A}$.[15] Everyone uses the same time zone (e.g., GMT), and the *timeout* is a globally known parameter[16]. If anything goes wrong prior to step 5 (no resolution protocol is applicable), the protocol will be aborted. The protocol proceeds

[15] We are abusing the notation by using hash values as verification algorithms provided by the Tracker hoping that the actual verification procedure of hashing the files and comparing the result with values given by the Tracker is obvious.

[16] It can easily be a per-exchange parameter known to (or agreed by) both parties.

as follows (summarized in Figure 2.1):

1. Alice creates a fresh public-secret key pair $pk_A, sk_A$ for a signature scheme. Alice sends a fresh unendorsed e-coin $coin'$ to Bob, along with a verifiable escrow $v = VE_{Arb}(end; pk_A)$ of the endorsement $end$, labeled with the signature scheme's public key.

2. Alice sends Bob ciphertext $c_A$ of her file.[17] Bob calculates $h_{c_A} = hash(c_A)$.[18]

3. Bob sends Alice ciphertext $c_B$ of his file. Alice calculates $h_{c_B} = hash(c_B)$.

4. Alice sends Bob an escrow $e = E_{Arb}(K_A; h_{f_A}, h_{f_B}, h_{c_A}, h_{c_B}, time)$ and her signature $s = sign_{sk_A}(e)$ on that escrow. The escrow $e$ should encrypt a key and should be labeled with four hash values $h_{f_A}, h_{f_B}, h_{c_A}, h_{c_B}$, and a *time* value. If any of the hash values do not match Bob's knowledge of those values, or if the *time* value is deviated too much from Bob's knowledge of the time (e.g., almost one timeout difference), then Bob aborts.[19] Moreover, if the signature $s$ on the escrow $e$ does not verify with the public key $pk_A$ sent in step 1 as part of the verifiable escrow $v$, Bob aborts the protocol.

5. Bob sends Alice his key $K_B$. Alice checks if the key $K_B$ decrypts the ciphertext $c_B$ correctly. If not, Alice does not proceed with the next step, and runs *AliceResolve*, although she might have to run it again just after the timeout to be able to resolve.

6. Alice sends Bob her key $K_A$. Bob checks if the key $K_A$ decrypts the ciphertext

[17] Alice and Bob can use their choice of (symmetric) encryption schemes (not necessarily the same). This only requires us to add the definition of the encryption scheme used to the messages exchanged.

[18] These will be Merkle hashes [109] for efficiency reasons, as discussed in Section 2.4.3.

[19] We do not require tight synchronization. So, for example, the *time* value can just contain hours, and not minutes and seconds.

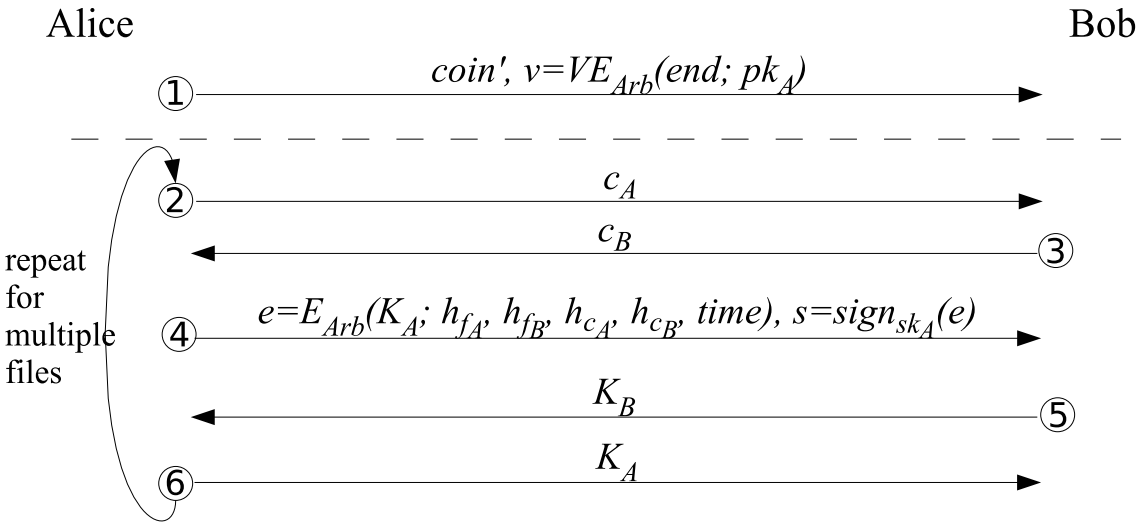


Figure 2.1: Our barter protocol with timeouts.

$c_A$ correctly. If not, he runs *BobResolve*; he must do so before the timeout.[20]

Once step 1 is completed, cheap steps 2-6 can be repeated to exchange more files, as long as no dispute occurs. Alice and Bob need not know beforehand how many or which files/blocks to exchange. Whenever they decide to exchange blocks (before every step 2), it is enough for them to just obtain their hashes from the Tracker. Actually, in BitTorrent, once you ask for hash of a file, the Tracker provides you with the hashes of all the blocks in that file already. Thus, connecting the Tracker for each block is not necessary in real life.

Below we present the resolution protocols in case of a dispute between Alice and Bob. The Arbiter never gets involved in a transaction unless there is a dispute.

### 2.4.1 BobResolve

Bob needs to contact the Arbiter before the timeout for resolution (current time $<$ *time* in escrow $e$ + *timeout*), since otherwise the Arbiter is not going to honor his

[20]Bob can run BobResolve immediately after a *message timeout*. He need not wait for a long time for Alice.

request. Assuming Bob resolves before the timeout, he provides the Arbiter with the escrow $e$ and signature $s$ that he received in step 4, and also the verifiable escrow $v$ he received in step 1 from Alice. The escrow $e$ should be labeled with four hash values $h_{f_A}, h_{f_B}, h_{c_A}, h_{c_B}$, and a *time* value. The verifiable escrow $v$ should be labeled with a public key $pk_A$ for a signature scheme. If the labels of the escrows are ill-formed, the Arbiter will not honor the request. The Arbiter checks the signature $s$ using the public key in the verifiable escrow $v$, and if it verifies, he asks Bob to present his correct key $K_B$ that verifies using the VerifyKey protocol in Section 2.4.3 (i.e., it decrypts a ciphertext with hash $h_{c_B}$ to a plaintext with hash $h_{f_B}$). If Bob succeeds in giving the correct key, the Arbiter stores the key $K_B$, decrypts the escrow $e$ and hands in the key $K_A$ from the escrow to Bob. Bob checks if $K_A$ decrypts Alice's file $f_A$ correctly. If not, he proves this to the Arbiter using the technique in Section 2.4.3 and gets the endorsement *end* in the verifiable escrow $v$ from the Arbiter.[21] Notice that only Bob may succeed in the BobResolve protocol with the Arbiter because any other party will fail to provide the correct key matching hashes of Bob's files (see Section 2.5.1).

### 2.4.2 AliceResolve

When Alice contacts the Arbiter for resolution, she asks for Bob's key $K_B$. If such a key exists, then the Arbiter sends $K_B$ to her.[22] $K_B$ has already been verified, so Alice does not need to perform any further action. If such a key does not exist yet,

[21] The Arbiter can abort this trade forgetting the $K_B$ in such a case. This is not necessary according to our definition (and can even be considered unfair), but it can be used as a way to punish cheating Alice even more. In the worst case, if non-atomicity of the Arbiter is allowed for efficiency reasons, Alice can obtain $K_B$ before Bob proves $K_A$ to be incorrect, effectively turning our protocol into a buy protocol.

[22] If the Arbiter is allowed to be non-atomical for efficiency reasons, then he needs to ask Alice for her key $K_A$, verifying it using the VerifyKey protocol in Section 2.4.3 before giving her $K_B$. This represents a tradeoff between the atomicity and efficiency of the Arbiter, which can be resolved arbitrarily, although it can also be used as a tougher punishment for cheaters.

Alice should come back after the timeout. If, even after the timeout $K_B$ does not exist, then Alice is assured that it will never exist, and can consider that particular trade as aborted.

### 2.4.3 Subprotocols

We use two subprotocols from Belenkiy et al. [16] that make the interaction with the Arbiter efficient. One protocol is used to prove that a key is not correct, while the other is used to prove that the key is in fact correct. For efficiency, Merkle hashes [109] are used in these subprotocols (see Belenkiy et al. [16] for more information on the protocols and the use of Merkle hashes).

**Proving a key is not correct**: Showing that a key $K$ does not decrypt a ciphertext $c$ with hash $h_c$ to a plaintext $f$ with hash $h_f$ can be done efficiently, as Belenkiy et al. suggests. Carol, to prove the key is not correct, gives the Arbiter a part $c_i$ of data which does not decrypt correctly. The Arbiter can check if the given part $c_i$ matches the Merkle tree hash of the ciphertext, and $Dec_K(c_i)$ does not match the hash of the plaintext, using the proof provided by Carol.

**Proving a key is correct**: Using a challenge-response protocol (Belenkiy et al. VerifyKey protocol), one can prove that a key is correct. The Arbiter asks for proofs of the key decrypting correctly on random chunks. If Bob can reply correctly to all chunks providing valid proofs for Merkle hashes, the the Arbiter accepts Bob's key. If Bob corrupts $1/m$ fraction of the file, and the Arbiter verifies $k$ random parts, then the Arbiter will catch Bob with a probability of at least $1 - (1 - \frac{1}{m})^k$ [16].

The security of both algorithms relies on the security of the universal one-way hash functions as described in Section 2.5.1.

**Algorithm 2.4.1**: VerifyKey from Belenkiy et al. [16]

**Arbiter's Input**: Two Merkle hashes $h_f$ and $h_c$, key $K$
**Bob's Input**: Ciphertext $c = c_0..c_n$, key $K$

`Step 1: Arbiter's challenge`
The arbiter sends Bob a set of random indices $I$.
`Step 2: Bob's response`
Bob replies with $c_i, cproof_i, fproof_i$ for every $i \in I$, where $cproof_i$ proves that $c_i$ is in the Merkle tree corresponding to $h_c$, and $fproof_i$ proves that $f_i = Dec_K(c_i)$ is in the Merkle tree corresponding to $h_f$.
`Step 3: Verification`
The arbiter *accepts* the key if Bob responds with valid $c_i, cproof_i, fproof_i$ for every $i \in I$, and *rejects* otherwise.

## 2.5 Security Analysis

In this section, we assume that we are given a one-way function, a universal one-way hash function, a chosen plaintext secure encryption scheme, a chosen plaintext secure verifiable escrow scheme, a chosen ciphertext secure escrow scheme, an unforgeable signature scheme, and an e-cash scheme which is unforgeable, anonymous and unlinkable. For precise definitions of security of these primitives, please see the references [75, 84, 119, 118, 109, 93, 57, 45, 44, 16]. In particular, we can use the instantiation in Section 2.8.

**Theorem 2.5.1.** *Our efficient barter protocol with timeouts as given in Section 2.4 is a secure optimistic fair exchange protocol according to Definition 2.3.3 in Section 2.3.*

*Proof.* It is obvious that our protocol satisfies the optimistic completeness (and therefore the completeness) property. We prove the fairness of our protocol over the fairness game defined in Section 2.3. Remember that our fairness condition states that either both parties obtain the other party's file, or one party obtains the other party's file while the other party obtains the e-coin (effectively turning into a buy protocol), or no party obtains anything.

An honest party will always use independent keys for each ciphertext (s)he sends.

Furthermore, endorsed e-cash [44] forces the users to use independent $(coin', end)$ pairs in different exchanges by using randomness contributed by both parties involved in the exchange. Our goal is that even if the adversary corrupts all other parties in the system (except the TTPs), he cannot obtain more than the union of what each of these individual corrupted parties was supposed to obtain from an honest trade with the honest user.

**Security of the Resolution Protocols**:

We first prove the security of our resolution protocols, as long as one of the participants is honest. Afterward, for the rest of the proofs, we will assume those are secure and do not worry about them.

**Claim 1.** *If BobResolve and AliceResolve protocols are executed in the $i^{th}$ exchange, $i^{th}$ exchange will be fair on its own.*

*Proof.* ***BobResolve***: When an honest Bob contacts the Arbiter, he provides the correct key $K_B$ and obtain the decryption of the escrow $e$ from the Arbiter. If this escrow contained the correct key $K_A$, then we are done. Otherwise, Bob can prove so (as in Section 2.4.3) and then the Arbiter hands out the endorsement *end* to Bob. This endorsement is valid due to the security of the verifiable escrow scheme (it can be shown by a reduction). Therefore, an honest Bob will obtain either the correct key or the endorsement of Alice.

If a dishonest Bob contacts the Arbiter, he cannot provide an incorrect key to the Arbiter and make him accept. This can easily be shown by reduction to the security of universal one-way hash functions [118] (see Section 2.5.1) or the VerifyKey protocol of Belenkiy et al. [16] (Section 2.4.3). If dishonest Bob provided the Arbiter his correct key $K_B$ and obtained honest Alice's correct key $K_A$, the only way he can be unfair against an honest Alice is to obtain her coin *end* in addition. But, Bob cannot obtain

*end* because he either has to forge Alice's signature on another escrow $e'$ of some junk key $K'_A$ which does not decrypt correctly, or he could break our assumption on the hash functions by providing some ciphertext with description $h_{c_A}$ which does not give a plaintext with description $h_{f_A}$ when decrypted using Alice's key $K_A$ in the escrow $e$. So, a dishonest Bob cannot obtain the endorsement of an honest Alice. Furthermore, he can obtain Alice's correct key $K_A$ only if he deposits his correct key $K_B$.

***AliceResolve***: In this protocol, Alice contacts the Arbiter and asks for Bob's key. If Bob deposited his key $K_B$ to the Arbiter, then Alice obtains it. From BobResolve, we know that if a key $K_B$ exists, it is correct. In case Alice was dishonest and obtained this key $K_B$ from the Arbiter, we know that honest Bob has already received either the correct key or e-coin of Alice using BobResolve. In case where Alice was honest but Bob was dishonest, we know he could not obtain both the correct key and endorsement of Alice. □

Hence, we can conclude that the resolution protocols do not help the adversary to win the game, and so if the adversary wants to be unfair in the $i^{th}$ exchange, he will not execute a resolution protocol for that exchange. Next we split the analysis of our main protocol into two cases: the case where the honest party plays the role of Alice, and the case where he plays the role of Bob.

**Case 1: Honest Alice vs dishonest Bob**:

**Claim 2.** *Suppose Bob succeeds in obtaining honest Alice's e-coin with non-negligible probability. Then we can construct an adversary $A_C$ breaking the e-cash scheme with non-negligible probability by playing the fairness game with Bob.*

*Proof.* $A_C$ is given a challenge *coin*$'$ and her goal is to output an endorsement *end*. [23] She guesses an index $i$ that Bob will succeed in being unfair, and replaces the

[23] A detailed proof will give $A_C$ two oracles, one for *coin*$'$ creation, and one for *end* creation. Then,

$coin'^{(i)}$ by the given $coin'$. Since $A_C$ does not know the $end^{(i)}$, she puts garbage into the verifiable escrow $v^{(i)}$, and sends it to Bob. She fakes the verifiability by using the simulator for the verifiable escrow [7, 45].[24] For all the other interactions, $A_C$ acts exactly as an honest Alice would. Since $A_C$ is honest, the verifiable escrow $v^{(i)}$ will never be decrypted by the Arbiter (shown in Claim 1), and by the security of verifiable escrow, the adversary cannot obtain the endorsement by decrypting it, nor can the adversary distinguish it from a verifiable escrow of a valid endorsement (can be shown by a straightforward reduction to CPA-security of the verifiable escrow scheme, since the verifiable escrow $v$ will never be decrypted because Alice is honest). At some point, Bob outputs an endorsement $end^{(j)}$ with non-negligible probability. The probability that $i = j$ is non-negligible by definition (the total number of barters $n$ is a polynomial in $\mathsf{SP}$ as defined in Section 2.3). If the indices match ($i = j$), $A_C$ outputs the $end^{(i)}$. Therefore, $A_C$ breaks the endorsed e-cash [44] with non-negligible probability, by endorsing an unendorsed coin $coin'$ without the endorsement $end$. □

**Claim 3.** *Suppose Bob, without calling BobResolve, succeeds in obtaining one of honest Alice's files $f_A^{(j)}$ with non-negligible probability before step 6 of $j^{th}$ exchange for some $j$ (Alice will perform step 6 only if she obtained the correct key $K_B^{(j)}$ from Bob). Then we can construct an adversary $A_E$ which breaks the encryption scheme Alice uses with non-negligible probability.*

*Proof.* $A_E$ generates her files using the setup phase. Then she guesses an index $i$ that Bob will succeed in being unfair, and sends two files to the challenger of the encryption scheme. $A_E$ is given back a challenge ciphertext $c_A$ and her goal is to

$A_C$ will play a CCA-security like game with the e-cash scheme. The challenge $coin'$ will be the one used in the $i^{th}$ exchange, on which $A_C$ cannot query the endorsement oracle.

[24] The verifiable escrow simulator can require simulating the public parameters too, but this is allowed and is indistinguishable from real public parameters due to the security of the verifiable escrow scheme.

decide which file she sent was encrypted. She replaces the $c_A^{(i)}$ by $c_A$. For the rest of the interaction, $A_E$ behaves as an honest Alice. $A_E$ does not know the key $K_A^{(i)}$, but she can fake the escrow $e^{(i)}$ by encrypting junk in it. Due to the security of the escrow scheme, Bob cannot distinguish it from an honest escrow (can be shown by a straightforward reduction to CCA-security of the escrow scheme). At the end, Bob returns a plaintext $f_A^{(j)}$. If the guessed $i$ was correct ($i = j$), then $A_E$ returns $f_A^{(i)}$ and wins with the same probability as Bob does. Since $A_E$ interacts with Bob only polynomially many times, the event $i = j$ has non-negligible probability, and since Bob has non-negligible probability of obtaining Alice's file, then $A_E$ has non-negligible probability of breaking the encryption scheme used. □

**Case 2: Honest Bob vs dishonest Alice**:

The argument is symmetric to Claim 3. The symmetric version of $A_E$ can easily be reconstructed as $B_E$ in this scenario, indistinguishable from an honest Bob. Hence, if Alice obtains Bob's file before step 5, $B_E$ breaks Bob's encryption scheme. After step 5, Alice already has Bob's file, and can choose not to send her key in step 6. But, the security of BobResolve guarantees that Bob can obtain Alice's key or e-coin in exchange to his file from the Arbiter (shown in Claim 1).

Combining these results, fairness for the honest party is guaranteed in all the exchanges, regardless of him playing the role of Alice or Bob. □

### 2.5.1 Universal One-Way Hash Functions

Let $H_k$ be a family of hash functions, where $k$ is the security parameter. We assume that the following experiment has negligible probability of success for any polynomial-time adversary $A$, for sufficiently large $k$: We have a file $f$ and a hash function $hash \leftarrow H_k$ uniformly chosen from the family. Given that file $f$ and the hash function's

description *hash* (which effectively also means giving $hash(f)$) as input, $A$ returns a $c, K$ pair, where $hash(Dec_K(c)) = hash(f)$ but $Dec_K(c) \neq f$. Remember that $A$ cannot control the file's hash, due to the trusted content and verification algorithm generation process, hence he needs to find a targeted collision.

This requirement is equivalent to the security of Universal One Way Hash Functions (UOWHF) [118]. We first reduce our assumption to the UOWHF assumption. Specifically, let $A$ be a polynomial-time adversary succeeding in the above attack with non-negligible probability. We can construct an adversary $B$ which finds a collision in our UOWHF as follows: When $B$ is given $(f, hash)$, he runs $A$ on $(f, hash)$ to obtain $(c, K)$. $B$ then checks if $Dec_K(c) = f$, in which case it fails. Otherwise, if $Dec_K(c) \neq f$ but $hash(Dec_K(c)) = hash(f)$, then $B$ outputs $Dec_K(c)$ as the collision. As easily seen, $B$ has the same success probability as $A$, and has polynomial runtime complexity.

The reverse reduction is also possible. Let $B$ succeed in attacking UOWHF with non-negligible probability. $A$, when given $(f, hash)$ as the challenge, runs $B$ on $(f, hash)$ to get $c'$ with $hash(c') = hash(f)$ and $c' \neq f$. $A$ then picks a random key $K$, and returns $(c = Enc_K(c'), K)$ as the answer. Obviously, $hash(c' = Dec_K(c)) = hash(f)$ but $c' = Dec_K(c) \neq f$. Hence, our assumption is equivalent to the UOWHF target collision-resistance assumption.

Our discussion above applies in our trusted content setting, where the content and verification algorithm generation process is trusted. If we allow the adversary to generate his own content (thus content generation is not trusted), he can as well generate bogus content. Yet, if we are in a semi-trusted setting where the adversary is allowed to generate his own content as long as it is not bogus (e.g., he can generate a movie file that really is showing the movie), then we need to use collision-resistant hash functions for security. The reasoning is that the content may be generated after

the hash function is chosen by the Tracker. This will not affect the practice, since all widely-used hash functions are assumed to be collision-resistant.

### 2.5.2 Privacy Analysis

None of the exchanged material contains information to identify Alice or Bob (not even Alice's signature, since it is a temporary -just for the exchange-, not permanent). Moreover, even an adversary performing multiple exchanges with the same honest party cannot link those exchanges together using the protocol messages since the honest party uses fresh keys every time and endorsed e-cash is unlinkable (IP address linking or similar means might be possible, but our protocol does not create any additional means of identification and linking). Furthermore, the Arbiter does not necessarily know who he is talking to, apart from the fact that the resolution is on a particular exchange (possibly identified by a random exchange ID). The Arbiter may be able to find out whether he is talking to Alice or Bob, but not who Alice or Bob is. Anonymous communication techniques such as onion routing [61] can be used when necessary. Lastly, e-cash [44] is anonymous, and thus even when Bob deposits the e-coin, no one can know it was Alice's e-coin (unless she double-spends).

## 2.6 Efficient Barter without Timeouts

We provide another protocol which does not make use of timeouts. In this case, both parties give e-coins to each other as a warranty. A similar setup applies here, where Bob is also required to have withdrawn an e-coin. Furthermore, Bob also generates a public-private key pair for his signature scheme. Details that were explained in our previous protocol will be omitted here.

1. ***a.*** Alice sends her unendorsed coin $coin'_A$, along with the verifiable escrow

$v_A = VE_{Arb}(end_A; pk_A)$ of the endorsement to Bob.

***b.*** Bob sends his unendorsed coin $coin'_B$, along with his verifiable escrow $v_B = VE_{Arb}(end_B; pk_B)$ of his endorsement to Alice.

2. ***a.*** Alice sends $c_A$ to Bob. Bob computes $h_{c_A} = hash(c_A)$.

   ***b.*** Bob sends $c_B$ to Alice. Alice computes $h_{c_B} = hash(c_B)$.

3. ***a.*** Alice picks a random value $r$ from the domain of a one-way function $g$, and computes $g(r)$. Alice sends her escrow $e_A = E_{Arb}(K_A; h_{f_A}, h_{f_B}, h_{c_A}, h_{c_B}, g(r))$ and her signature $s_A = sign_{sk_A}(e_A)$ on her escrow to Bob. Bob aborts the protocol if the signature $s_A$ does not verify under $pk_A$ in $v_A$ or the hash values do not match Bob's knowledge of those values.

   ***b.*** Bob sends his escrow $e_B = E_{Arb}(K_B; h_{f_A}, h_{f_B}, h_{c_A}, h_{c_B}, g(r))$ and his signature $s_B = sign_{sk_B}(e_B)$ on his escrow to Alice. Alice calls AliceAbort below if the signature $s_B$ does not verify under $pk_B$ in $v_B$, or the hash values or $g(r)$ do not match Alice's knowledge of those values.

4. ***a.*** Alice sends her key $K_A$ to Bob.

   ***b.*** Bob sends his key $K_B$ to Alice.

Regarding efficiency, again, **step 1 has to be completed only once per peer, and then multiple files can be exchanged by carrying out steps 2-4** as long as both parties are honest, amortizing the cost of the coin and verifiable escrow exchange in step 1.

The escrows in step 3 are a bit different than the previous protocol. First, there is no *time* value attached, since no timeouts are used. Furthermore, both escrows need to contain a value $g(r)$ where $g$ is a one-way function, and only Alice knows $r$. This is achieved by requiring Alice to pick a random $r$ in step 3.a, and then put $g(r)$ in the label of the escrow. After receiving Alice's escrow $e_A$, Bob also incorporates $g(r)$

into the label of his escrow $e_B$.[25]

The new AliceResolve and BobResolve algorithms are both very similar to the BobResolve in our barter protocol with timeouts (of course, both parties use the escrows and signatures received from the other party, AliceResolve gets $K_B$ by giving $K_A$, and there are no timeouts), and they should be run if the key Alice or Bob receives at step 4 is not correct, respectively.

The logic behind getting rid of the timeouts is similar to the idea in ASW [7]. If Alice wants to abort the protocol (because something was wrong with the message she received in step 3.b, or she did not receive any response, she can do so by contacting the Arbiter using the AliceAbort protocol below. She no longer needs to wait until after the *timeout*. After receiving (or not receiving) Alice's message at step 3.a, Bob can simply abort locally if anything is wrong.

### 2.6.1 AliceAbort

Alice contacts the Arbiter, handing him her escrow $e_A$, her signature $s_A$ on that escrow, and her verifiable escrow $v_A$ that contains the public key $pk_A$ for the signature. The Arbiter checks the signature first. If it verifies, he requires Alice to give a value $r$ so that $g(r)$ matches the one-way function value in the label of the escrow $e_A$ (therefore Bob cannot succeed in this protocol). Then, the rest proceeds similar to the AliceResolve in our previous protocol. Alice asks the Arbiter for Bob's key $K_B$. If such a key exists (because Bob resolved before Alice aborted), then the Arbiter sends $K_B$ to Alice. $K_B$ has already been verified, so Alice does not need to perform any further action. If such a key does not exist yet though, the Arbiter considers that particular trade as aborted, and will perform no further resolutions regarding

[25] This is showing how the Arbiter can distinguish Alice and Bob using one-way functions, as discussed in previous footnotes. Other possible measures having the same effect can also be taken.

this particular barter.[26] (Remember, Alice needed to come back after the timeout in our previous protocol.)

### 2.6.2 Analysis of Barter without Timeouts

The advantage of this protocol is that there is no need for timeouts. Alice can safely abort the protocol (using AliceAbort) without waiting in case Bob tries to cheat in step 3.b. Bob can simply abort unilaterally if Alice tries to cheat in step 3.a. Since it is very similar to our protocol with timeouts, we are not presenting a detailed analysis for this protocol. Alice performs almost exactly the same moves as in our previous protocol, and hence all the proofs there can be applied here, extendable to both Alice and Bob, with minor modifications due to minor differences in the resolution protocols.

**Theorem 2.6.1.** *Our efficient barter protocol without timeouts in Section 2.6 is a secure optimistic fair exchange protocol due to the Definition 2.3.3 in Section 2.3.*

*Proof.* Omitted due to extreme similarity with the proof of our protocol with timeouts. The proof of AliceResolve is now the symmetric version of BobResolve before. The proof of AliceAbort is very similar. Furthermore, the corresponding adversaries $A_C$, $A_E$, $B_C$, and $B_E$ are very straightforward to construct. □

The privacy analysis of this protocol is the same as our protocol with timeouts. Besides, the generalization above also applies to this protocol.

**Theorem 2.6.2.** *Our efficient barter protocol without timeouts preserves the privacy of the honest participants even when Arbiter resolution is required.*

[26] Similar footnotes as before applies. If, for example, we do not want to rely on the security of the Belenkiy et al. VerifyKey protocol here, Alice can prove that Bob's key was incorrect -if that is the case- and get his e-coin from the Arbiter. If Bob already resolved, he must have taken Alice's correct key or e-coin. Hence, the exchange is fair, becoming e-coin to e-coin exchange in such a case.

*Proof.* Same as the proof for our protocol with timeouts. □

## 2.7 Generalized Version

We have shown an instance of our protocol which uses hashes for verification, and endorsed e-cash for payment. In general, **our protocols can employ any secure verification algorithm** (see Definition 2.3.2) provided by the Tracker, instead of the hashes. Similarly, **our protocols can easily make use of other payment methods** (see [4] for a compilation) or signatures instead of e-cash, but then privacy of the participants will not be preserved. The modification is straightforward, and involves just replacing the verifiable escrow of the e-coin with a verifiable escrow of any other form of payment.

## 2.8 Efficiency Analysis

The efficiency of Alice's and Bob's parts in the protocol can be further improved, although this would require the Arbiter to perform more work. To improve Alice's and Bob's efficiency, Bob sends the file unencrypted in step 5, instead of separately sending the ciphertext in step 3 and the key in step 5, thus eliminating step 3 completely (a similar logic might also apply to steps 2 and 6). But, in that case, the Arbiter needs to keep the whole file for resolution purposes instead of only a very short key as in the current case. Since such trusted third parties can become the bottlenecks of the system, we prefer having the least amount of work to be done by the Arbiter, and let users perform slightly more work instead. Moreover, if secrecy of the files is desired, they will be encrypted anyways.

We consider a concrete instantiation of our protocol using endorsed e-cash [44], Camenisch-Shoup verifiable escrow [45], AES encryption [57], DSS signatures [119],

and RSA-OAEP public key encryption for (non-verifiable) escrow [20]. Our protocol has only neglectable overhead over just doing an unfair exchange. Sending the ciphertexts in steps 2 and 3 just corresponds to sending the files in any (even unfair) exchange.[27] The keys sent in steps 5 and 6 are extremely short messages (16 bytes each for 128-bit AES keys). For a fair exchange, step 4 is still very cheap since the only primitives used are an ordinary (non-verifiable) escrow (just a public key encryption), and a signature (A DSS signature created using a 1024-bit key is about 40 bytes, while an RSA-OAEP encryption with a 1024-bit key is about 128 bytes).

Assuming IO and CPU can be overlapped, encryption of files will not add any time. Furthermore, signatures and escrows take only a few milliseconds. The most time consuming step is sending the blocks themselves, which has to be done in any case (and encryption does not increase size). The only real overhead is the first step, where the verifiable escrow (and endorsed e-cash, if used) is costly (see below).

Our protocol, in addition to guaranteeing fair barter efficiently, is optimized for multi-barter situations. One such situation is a file sharing scenario as in BitTorrent [52, 16]. The peers Alice and Bob are expected to have a long-term barter relationship. Hence, **step 1 needs to be carried out only once per peer, and remaining cheap steps 2-6 would be repeated for each block, whereas previous protocols required a costly step like step 1 to be performed for each block**. This greatly amortizes the costly step 1 in our protocol, when multiple blocks (or files) are exchanged, **even when the files/blocks to be exchanged are not pre-defined** (they need to be defined only before each execution of step 2).

To give some numbers, consider an average BitTorrent file of size $2.8GB$ made up of about $2,500$ blocks [89]. Using previous optimistic fair exchange protocols,

[27] We can in general assume that the I/O and CPU can be pipelined so that the encryption will not add more time to uploading the files.

this requires $2,500$ costly steps (one per block). Our C++ implementation using endorsed e-cash [44] and Camenisch-Shoup verifiable escrow [45] takes about 1 seconds of computation for step 1 (most of which is the verifiable escrow) on an average computer ($2GHz$). This corresponds to $2500 \times 2seconds =$ **42 minutes** of computation overhead. Considering a BitTorrent client that connects to about 40 peers, using our protocol, this overhead becomes just **40 seconds**, which is neglectable when exchanging such a big amount of data (this cost will be dominated by the file transfer times). Our network overhead is similarly neglectable (around $40KB$ per peer, almost all of which is the one-time cost of step 1, about half of it being endorsed e-cash). This corresponds to about $2500 \times 2 \times 40KB =$ **200 MB** total overhead using previous schemes, and only $40 \times 40KB =$ **1.6 MB** total overhead using our scheme (for a $2.8GB$ file).

As for the Arbiter, he checks a signature, sometimes decrypts a (verifiable) escrow, and performs the VerifyKey protocol of Belenkiy et al. [16] (see Section 2.4.3). The signature check and ordinary escrow decryption takes only milliseconds, the verifiable escrow decryption, when necessary, can take a few hundred milliseconds. The bottleneck is the data that the Arbiter needs to download for the VerifyKey protocol, which is about $22chunks \times 16KB = 352KB$ [16]. An important point to note is that *the amount of data the Arbiter's needs to download is independent of the size of the file that is being exchanged.*[28]

Without considering distributed denial of service (DDoS) attacks, let us provide some numbers for evaluation. To have an idea, consider a P2P system of $1,700,000$ users, exchanging $2.8GB$ files on the average [89]. Exchanging two such files means exchanging $5.6GB$ of data. If 1% of all users are malicious, this can correspond to $17,000$ exchanges requiring an arbiter at a given time (where one user is honest and

[28] Merkle proofs are logarithmic in number of the blocks in the file, but are much smaller in size than the data blocks themselves in practice.

the other is malicious. If both of them are malicious, this number reduces to half of it). We said, in case of a dispute, a peer should upload $352KB$ of data to the Arbiter. Assume that the same upload speed is used when trading files and contacting the Arbiter. If we assume the worst case scenario where the Arbiter can handle only one user at a time and every user is active at all times, this requires having 2 arbiters; with 10% malicious user ratio, we need 11 arbiters. Under the very realistic assumption that an arbiter can handle 25 users at a time (e.g., assuming 25 times as fast download speed of the Arbiter as the upload speed of the users [53]), we will need 1 arbiter in this system (even with 10% malicious user ratio).

We believe, in many situations, our more efficient protocol with timeouts will be sufficiently useful. Yet, to provide options, we chose to present another efficient barter protocol that does not require the use of timeouts. Our protocol without timeouts requires *two* costly operations (step 1) instead of *one* in our protocol with timeouts. As in our protocol with timeouts, this cost is independent of the number of files exchanged, and becomes neglectable when multiple or large files are exchanged. The cost of step 1 will be doubled for both parties, yet for the rest of the protocol the cost will stay almost the same. The Arbiter's cost will be doubled though, due to the need to perform two costly resolutions (AliceResolve is as costly as BobResolve now). Nevertheless, using similar numbers as above, if our arbiter can handle 25 users at a time, we still need only 1 arbiter even with 10% malicious user ratio. Some more efficiency evaluation, limitations and possible solutions are discussed in the next section.

Finally, note that for both of our protocols, there is no growing history that needs to be kept. In terms of storage, both parties need to store at most one verifiable escrow, and the messages and blocks for that exchange. Any messages related to previously exchanged blocks can be discarded. Thus, during an exchange, the storage

overhead of our protocol is less than 1KB.

## 2.9 Limitations and Future Work

One limitation of our work is the need for the exchanging parties to trust the Arbiter. Alice trusts the Arbiter not to give away both her e-coin and the key to her file. Even though giving away the key only makes the exchange unfair, giving away the coin may result in even an honest Alice becoming a *double-spender*.[29] One possible way to reduce this need for the trust would be using several arbiters, who do not necessarily know each other. Alice and Bob can mutually agree on a specific arbiter, *the Arbiter*, before the protocol begins. Since, there is no registration with the Arbiter in our protocol, any arbiter can accomplish the job.

Fortunately, if a proof of dishonesty is requested, neither the Arbiter, nor Bob, nor anyone else can frame an honest Alice.[30] The Arbiter may be asked to prove Alice's guilt by presenting a verifiable escrow, a non-verifiable escrow and a signature on it, along with the proofs that Bob's key decrypts correctly yet Alice's key in the (non-verifiable) escrow does not. Due to the security of these primitives, no one can frame an honest Alice. Of course, this requires the Arbiter to store all past resolutions, and Alice's privacy has already been invaded by the double-spending detection. In order to prevent a malicious Alice from framing the Arbiter by intentionally double-spending, we can require either Alice's or the Arbiter's signature when a coin is being deposited. We leave the issue of efficiently reducing the need to trust the Arbiter or verifying the Arbiter's behavior without violating Alice's privacy as a future work.

As for the bottleneck that can be caused by the central Arbiter, Avoine et al. [11] show how to employ secret sharing techniques [140, 27] to distribute the shares of the

[29]This does not result in Alice losing money, but losing her anonymity.

[30]Of course, this requires yet another trusted entity, called the *Judge*.

secrets among arbiters. This will decrease the amount of job each arbiter needs to perform, yet it will reduce the efficiency of our resolution protocols. It also allows for dilution of trust. As argued in Section 3, the same techniques can be applied to our protocol with timeouts. In Chapter 4, we show how to outsource computation, which can be used as a means to distribute the work of our trusted parties. The Brownie Project [34] is analyzing this strategy to distribute the arbiter and the bank in their BitTorrent deployment.

As in many deployments, it is possible to mount a distributed denial of service (DDoS) attack on the arbiters by continuously performing fake barters and resolving with an arbiter. We leave the protection against such attacks (by means like blacklisting IP addresses) to system and network security researchers. Alternative strategies of reducing the arbiters' load were already discussed above.

Another limitation is that Bob does not need to do any work to be able to send a response to Alice in our protocol with timeouts, so he can just send junk. Hence, Bob can mount a distributed denial of service attack against Alice. Yet, he still needs to upload a large file (and can be required to upload first), wasting considerable amount of resources (time and bandwidth). Moreover, Alice will not be trading with Bob once he cheats. We leave the issue of analyzing the extent of such attacks, catching such an attacker, and proving such an attack occurred as an open problem. This attack is not possible when our protocol without timeouts is used, since both parties need to do equal amount of work.

In terms of the storage load associated with the trusted parties, the techniques from [39] can be applied. Using those techniques, the Bank can have a limited storage, as opposed to a groving storage. For example, if every e-coin is valid for a limited but long time (e.g., one month), then the bank needs to keep track of only the transactions that happened in the past period, instead of all past transactions. Note

that the Arbiter also only needs to have a short-term memory of past resolutions.

Lastly, our fairness definition states that a file and a payment can be fairly traded, as in previous works [7, 16, 44, 107, 102]. The economics of this system, deciding on how much a file is worth fairly, is outside the scope of this chapter. The participants can somehow agree on the price before our protocol begins (variable pricing), or alternatively a system can set the price that will apply to all participants (fixed pricing). In Belenkiy et al. [16], the authors assume each block in the BitTorrent system are worth one e-coin. We leave this pricing issue as an interesting application-dependent open problem.

## 2.10 Conclusion

There already are many scenarios where peers trade content [52, 89]. These systems unfortunately rely on the honesty of the peers for providing fairness, partly because of the high cost incurred by the previous fair exchange protocols [5, 6, 7, 8, 16, 44, 111]. Our protocols uniquely limit the use of the costly primitives (verifiable escrow and e-cash) to once (or twice) per peer, as opposed to per file/block. We have shown in Section 2.8 that there are one or two orders of magnitude efficiency gains over previous protocols. Besides, most of the existing systems already rely on similar trusted parties [5, 6, 7, 8, 16, 40, 44, 50, 52, 89, 111, 122]. Therefore, for the first time, by using our protocols, such bartering systems will experience almost no performance loss, while the benefit of providing fairness guarantees will be very noticeable indeed (e.g., see [16] for how the use of fair exchange can solve the free-riding problem of BitTorrent). Already, the Brownie Project [34] is adopting our protocols in their BitTorrent deployment.

As a guideline, we suggest that systems which expect long-term barter relationships and are not willing to use timeouts use our protocol without timeouts, but

systems that will conduct mainly short-term barters and can tolerate timeouts use our protocol with timeouts.

# Chapter 3

# Trusting the Cloud

## 3.1 Introduction

Optimistic fair exchange is a very useful primitive in distributed system design with many applications including contract signing, electronic commerce, or even peer-to-peer file sharing [5, 6, 7, 8, 14, 16, 63, 96, 110, 111]. In a fair exchange protocol, Alice and Bob want to exchange some items, and they want to do so fairly. Fairness intuitively refers to Alice getting Bob's item and Bob getting Alice's item at the end of the protocol, or neither of them getting anything, even if one of them maliciously deviates from the protocol. For technical definitions of optimistic fair exchange protocols, we refer the reader to Chapter 2.

It has been shown that no general fair exchange protocol can provide complete fairness without a trusted entity [122], called the *arbiter*. In an optimistic fair exchange protocol, the arbiter is not involved unless there is a dispute between the participants. But having a single trusted entity is one of the biggest problems that make the use of such protocols hard in practice. Therefore, the use of multiple arbiters is generally motivated by reducing the trust put on the arbiter (see Chapter 2).[1] A

[1]It is possible to have multiple arbiters deployed for reducing the load, but if only one of them is

very natural question is how to achieve fairness in the absence of a single trusted arbiter; for example, what if we have $n$ arbiters only a fraction of whom we want to put our trust in? It is clear that this can be achieved using byzantine agreement or secure multi-party computation techniques [81, 22, 24, 47] with $\Omega(n^2)$ communication, but can we do better than that? In particular, can we do anything in a setting where the arbiters need not communicate with each other to resolve disputes? This issue is highly relevant especially for peer-to-peer settings in which the arbiters do not even know each other, and may not have enough resources for complicated schemes. Furthermore, if the scheme gets more costly, it will be hard to incentivize multiple arbiters to arbitrate, since they will get overloaded.

Avoine and Vaudenay (AV) [11] address this problem in their paper by using verifiable secret sharing techniques to employ multiple arbiters in their fair exchange protocol for a P2P system. In their setting, two peers are performing a fair exchange, and a number of other peers constitute the arbiters. They provide bounds on the number of arbiters that should be honest for their protocol to be fair (see Section 3.7). A crucial point is that the protocol uses global timeout mechanisms, which assumes all arbiters have access to -loosely- synchronized clocks, and the arbiters are autonomous (they do not communicate with each other). They leave two important issues as open questions: (1) Can an optimistic fair exchange protocol without timeouts provide fairness (since it is hard to achieve synchronization in a P2P setting) when employing multiple autonomous arbiters? (2) Can any other optimistic fair exchange protocol with timeouts achieve better bounds on the number of arbiters that need to be honest?

Unfortunately, in this chapter, we answer both of these questions negatively. Inspired by state-of-the-art optimistic fair exchange protocols with a single arbiter, we define a general class of optimistic fair exchange protocols with multiple arbiters,

employed per exchange, we do not consider that protocol as having distributed arbiters.

called "distributed arbiter fair exchange" (DAFE) protocols. Informally, in a DAFE protocol, if one of the participants fails to send a correctly formed message, the other participant must contact some subset of the arbiters and get correctly formed responses from them in order to make the exchange fair.[2] Two main properties of a DAFE protocol are its abort/resolve semantics and the autonomy of multiple arbiters used, as discussed in Section 3.2. In a DAFE protocol, the arbiters are autonomous; they do not talk to each other, but talk only to Alice and Bob. A third property is the state machine semantics of the participants. We show that this class of protocols capture currently known state-of-the-art optimistic fair exchange protocols extended to use multiple distributed arbiters in a very intuitive manner, as shown in Section 3.2.1. Under this framework, in Section 3.4 we analyze scenarios that can occur during the execution of instances of optimistic fair exchange protocols, and prove some predicates every such protocol must satisfy to be able to provide semantic fairness, which is a property that needs to be satisfied by all optimistic fair exchange protocols.

In Section 3.5, we prove that no DAFE protocol can provide fairness meaningfully[3], answering the first open question negatively. In Section 3.6, we prove impossibility of DAFE protocols using threshold-based mechanisms (any $k$ arbiters are enough for resolution) even when the autonomous arbiters assumption is relaxed. For protocols using general set-based mechanisms (any $k$ arbiters will not be enough for resolution, specific sets of arbiters need to be contacted), we cannot prove impossibility in this relaxed setting, but we conjecture that such protocols are not possible. However, our impossibility results can be overcome in the timeout model (where all arbiters have access to loosely synchronized clocks) and also in case the arbiters can communicate. We use our framework to analyze the existing AV protocol [11] in this timeout model

[2]Of course, if no message is sent yet, there is no need to contact arbiters, which is not an interesting case to analyze anyway.

[3]We prove that multiple arbiters are no better (or actually worse) than a single arbiter in terms of trust in the DAFE framework.

in Section 3.7, showing how easy it is to apply our framework. We prove that the bounds on the required number of honest arbiters proven earlier for that protocol are optimal, and hence answer the second open question also negatively.

These results mean that many optimistic fair exchange protocols that want to efficiently distribute their arbiters may need to employ synchronized clocks. And even in this case, they cannot hope to require fewer honest arbiters than the Avoine and Vaudenay protocol [11]. If they do not want to employ synchronized clocks, then they may need to employ costly solutions like secure multi-party computation or Byzantine agreement (or cheaper Byzantine fault-tolerance techniques [1] secure under weaker adversarial models).

## 3.2 Definition of a DAFE Protocol

In this section, we define a general optimistic fair exchange model that fits currently known state-of-the-art optimistic fair exchange schemes that uses an arbiter, and has semantics for aborting and resolving that we define below.

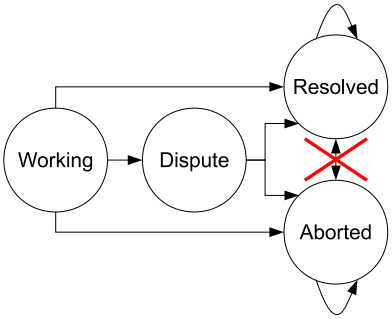


Figure 3.1: Semantic view of the state machines of the participants.

All the participants (Alice, Bob and the arbiters) are interactive Turing Machines (ITMs)[4]. Those ITMs have the following 4 semantic states: *Working* , *Aborted* , *Resolved* , *Dispute* (see Figure 3.1). These semantic states can correspond to multiple states in the actual ITM definitions of the participants, but these abstractions will be used to prove our results.

[4]The ITMs have access to –possibly synchronized– clocks for timeout mechanisms.

The ITM of each participant starts in the *Working* state. Semantically, *Working* state denotes any state that the actual ITM of a participant is in when the protocol is still taking place. When a participant does not receive the expected correctly formed message from the other participant, he can possibly abort or decide to contact the arbiters for resolving or aborting with them, in which case the ITM of that participant enters its *Dispute* state. If everything goes well in the protocol execution (all messages received from the other party are correctly formed), then the ITM of a participant transitions to the *Resolved* state directly from the *Working* state. Otherwise, if the arbiters needed to be contacted, the ITM first visits the *Dispute* state, and then transitions to either *Resolved* or *Aborted* state. Arbiters' *Dispute* state is dummy, and hence not needed in our analysis. Furthermore, when in Section 3.6 we relax one of our assumptions, even Alice and Bob will not have this *Dispute* state.

When the protocol ends, Alice and Bob are allowed to end only in *Aborted* or *Resolved* states. If Alice or Bob ends at its *Resolved* state, then, by definition, (s)he must have obtained the exchange item from the other party. When the protocol ends, if the ITM of a participant is not in its *Resolved* state, it is considered to be in its *Aborted* state.

Using these semantic definitions, even an adversarial ITM can be considered to have those 4 states (since it either obtains the other party's item and hence ends at its *Resolved* state, or not therefore ending at its *Aborted* state). The adversarial ITM does not necessarily have a *Dispute* state, but this will not affect any results presented in this chapter. One can think that the moment the honest party's ITM enters its *Dispute* state, the adversarial ITM also enters its *Dispute* state.

We will talk about only complete DAFE protocols (remember definition of an optimistic fair exchange protocol from Section 2.3): when both participants are honest, they end at their *Resolved* states. Since our goal here is to analyze fairness of such

protocols, the only interesting case is when we have one honest party denoted $H$ and one malicious party denoted $M$. We will not consider cases where both parties are malicious since there is no honest party to protect.

**Definition 3.2.1** (End of the Protocol)**.** *We say that the protocol has ended if (1) the honest party ended up being in her either* Resolved *or* Aborted *state, and (2) the adversary produced its final output at its either* Resolved *or* Aborted *state after running at most a polynomial number of steps (polynomial in some security parameter).*

Now that we defined our participants carefully, we can state our assumptions on them and define DAFE protocols.

Distributed Arbiter Fair Exchange (DAFE) protocols: DAFE protocols are optimistic fair exchange protocols that can be characterized with the following:

- Exclusive states assumption
- Connection between arbiters' state and Alice's and Bob's
- Autonomous arbiters assumption

Exclusive states assumption: This assumption states that the *Resolved* and *Aborted* states are mutually exclusive. For an arbiter, those states informally mean whether or not the arbiter helped one of the parties to resolve or abort. We assume that there is no combination of state transitions that can take an honest *arbiter* from the *Aborted* state to the *Resolved* state, or vice versa. In most existing protocols, this corresponds to the fact that the arbiter will not abort with a participant first and then decide to resolve with him or the other participant, or vice versa. An honest arbiter can keep executing abort (or resolve) protocols with other participants in the

exchange while he is in the *Aborted* (or *Resolved* , respectively) state, but can not switch between states for different participants.

**Definition 3.2.2** (Aborting and Resolving with an Arbiter)**.** *If a participant interacts with an arbiter and aborts with him, the arbiter goes to his* Aborted *state, from where he will never switch to his* Resolved *state. Similarly, if a participant resolves with an arbiter, the arbiter goes to his* Resolved *state, from where he will never switch to his* Aborted *state.*[5]

**Definition 3.2.3** (Aborted and Resolved Protocol Instance)**.** *A protocol instance is called aborted if both Alice and Bob ended at their* Aborted *states, and called resolved if both Alice and Bob ended at their* Resolved *states.*

Connection between arbiters' state and Alice's and Bob's: A resolution makes sense if at least one of the parties has not resolved yet. In such a case, Alice or Bob can end in their *Resolved* states (unless they already are in their *Resolved* states) only if a set of arbiters end in their *Resolved* states. This set of arbiters can be different for Alice or Bob. Actually, there can be more than one set of arbiters that is enough for this resolution. All these will be clear in later sections when we define those sets of arbiters that will be sufficient for resolution.

Autonomous arbiters assumption: We assume that the honest arbiters' decisions are made autonomously, without taking into account the decisions of the other arbiters. Arbiters can arrive at the same decision seeing the same input, but they will not consider each other's decision while making their own decisions. In particular, this means no communication takes place between honest arbiters (malicious arbiters can do anything they want).

[5]Due to the exclusive states assumption, these happen only if an arbiter is not already in his *Resolved* or *Aborted* state, respectively.

Our goal in this is to distribute the trust efficiently. Without autonomy, byzantine fault tolerance or secure multi-party computation techniques [81, 22, 24, 47] can be applied, yielding costly solutions ($\Omega(n^2)$ communication when $n$ arbiters are employed). Furthermore, autonomy of the arbiters render the deployment of such a real system practical, since no coordination of the arbiters is necessary.

Yet, a dependence between the arbiters' decisions can be generated by Alice or Bob, by contacting the arbiters with some specific order. Therefore, to model the autonomy, we require the protocol design to direct the honest participants to contact all the arbiters without any order. More formally, when the ITM of an honest participant decides to contact the arbiters for dispute resolution, the participant creates the message to send to all of the arbiters before receiving any response from any arbiter. One can model this with the *Dispute* state in which the message to send to the arbiters are prepared all at once. We will call this simultaneous (or unordered) resolve/abort. Note that this only constrains honest Alice or Bob. A malicious party can introduce dependence between messages to arbiters and responses from other arbiters. Later in Section 3.6 we will relax this autonomy assumption and discuss its consequences. We realize that this assumption is not necessary for most of our results, but helps making the presentation clearer.

All optimistic fair exchange protocols need to satisfy the following semantic fairness property.

Semantic Fairness: The semantic fairness property states that at the end of the protocol, Alice and Bob both end at the same state (they both end at their *Aborted* states, or they both end at their *Resolved* states). In other words, we need the protocol instance to be either resolved or aborted as in Definition 3.2.3, for every possible instance of the protocol.[6]

[6]There will not be any cases where the honest party ends at its *Resolved* state whereas the malicious party ends at its *Aborted* state and this affects our results. Therefore, this semantic fairness definition is enough for our purposes. Furthermore, it is subjective whether or not to

Optimistic fair exchange protocols should also satisfy the *timely resolution* property, meaning that the honest party need not wait indefinitely for any message from any other party. He can have a local timeout mechanism with which he can decide to proceed without waiting. In particular, he can end his side of the protocol any time he wants, ending at his *Resolved* or *Aborted* state, according to the rules we defined above. Note that in general providing timely resolution guarantees necessitates mutually exclusive *Resolved* and *Aborted* states, and a way for the arbiters to transition to their *Aborted* states through interaction with other parties or through the use of timeouts.

Regular DAFE protocols do not have global timeout mechanisms, and the sets of arbiters that Alice or Bob can resolve with are well-defined by the protocol, and does not change once the honest party is in its *Dispute* state. We will show an extended version called DAFE with timeouts (DAFET) where the protocols are allowed to use timeouts. At the timeout specified by the protocol, honest arbiters transition into their *Aborted* states. This is done using the (loosely synchronized) clocks of the ITMs. We call this event "an arbiter timeouts". We allow the possible sets of arbiters to resolve with to change at this timeout. This timeout model bypasses the impossibility results for DAFE protocols. These will be clear later.

We will first provide examples of existing optimistic fair exchange protocols with intuitive extensions to employ multiple autonomous arbiters and show how they fit our DAFE classification. Then, after defining some notation, we will analyze different possible protocol instances under different scenarios, and possible protocol types. We then show that it is impossible for some common types of DAFE protocols to provide semantic fairness, thus warning researches not to pursue that direction. We also analyze some positive results using global timeout mechanisms, and prove the

consider a case where two parties end at different states as fair.

optimality of the bounds of the AV protocol, showing the usability of our framework for easy analysis. We then discuss the role of autonomous arbiters and timeouts in our results and elaborate on different ideas.

### 3.2.1 Sample DAFE Protocols

Many currently known optimistic fair exchange protocols can be considered as special cases of DAFE protocols in which there is only one arbiter. In this section, we also discuss a way to extend them to employ multiple autonomous arbiters. Unfortunately, this means, those extended protocols cannot provide fairness, as we will prove later in this chapter that no DAFE protocol can provide fairness. Precisely, our impossibility result states that all arbiters need to be trusted in a DAFE protocol, hence they are not realistic. For the special single-arbiter case, this points out to the trust assumption on the arbiter.

To the best of our knowledge, all currently known optimistic fair exchange protocols adhere with our framework. As a representative of optimistic fair exchange protocols, we will analyze a protocol due to Asokan, Shoup and Waidner (ASW) [7]. They have two versions of their protocol: one version that uses timeout-based aborts (can be converted to a DAFET protocol, see Section 3.7), and one that does not employ timeouts (we will discuss now). It is considered one of the state-of-the-art signature exchange protocols, and is the first completely fair optimistic exchange protocol. A state-of-the-art optimistic fair exchange protocol for exchanging files was given in Chapter 2, and all our discussion here applies to that protocol too. The ASW protocol without timeouts is described in Appendix A.11 for reference.

In terms of the state semantics of the participants, it is clear that the ending states of the participants can be parsed into *Aborted* and *Resolved* states which are mutually exclusive. Furthermore, honest participants are not allowed to transition

between *Aborted* and *Resolved* states. In particular, once Alice aborts with the arbiter taking him to his *Aborted* state, he will refuse resolving with Bob. Since there is only one arbiter, it is autonomous. As for the connection between arbiter's state and Alice's and Bob's, it is clear that in case of a dispute, their state depends on the arbiter's.

Now, if we want to extend those protocols to use multiple autonomous arbiters, one easy way is to employ verifiable secret sharing techniques [11, 128, 77]. The state-of-the-art optimistic fair exchange protocols employ verifiable escrows [37, 45, 7, 96] under the (one and only) arbiter's public key. The intuition behind using verifiable escrows is that the recipient can verify, without learning the actual content, that the encrypted content is the content that is promised and the arbiter can decrypt it. Verifiable secret sharing techniques can be employed to split the promised secret per arbiter. Each of these secrets will be encrypted under a different arbiter's public key. The recipient can still verify those encrypted shares can be decrypted and combined to obtain the promised secret, thereby effectively achieving the same goal as a verifiable escrow, but for multiple arbiters. For a detailed explanation of how to use verifiable secret sharing in distributing the arbiters, we refer the reader to [11].

When we extend the ASW protocol to use multiple autonomous arbiters, instead of this verifiable escrow, the participants will use verifiable secret sharing techniques as explained above and in [11]. Regardless of whether threshold- or set-based secret sharing mechanisms are used, the resolution procedure now requires contacting multiple arbiters. For example, if the threshold for the secret sharing method used is $k$, the resolution will involve contacting at least $k$ arbiters.

In terms of the state semantics of the participants, it is clear that the ending states of the participants can be parsed into *Aborted* and *Resolved* states which are mutually exclusive. Because we assume the arbiters are contacted simultaneously,

the autonomy of the arbiters hold. As for the connection between arbiters' state and Alice's and Bob's, since resolution needs $k$ shares, and secure secret sharing and encryption methods are used, a participant can obtain the other participant's exchange item if and only if (s)he resolves with at least $k$ arbiters (in case of a dispute). This relationship makes perfect sense when multiple autonomous arbiters are used, since the main goal in distributing the arbiter is distributing the trust. Therefore, the goal is to find some number of honest arbiters each one of which will individually contribute to dispute resolution between participants by resolving or aborting with them. When arbitrary sets are used instead of thresholds, it is easy to see all these arguments will still apply.

The same techniques can be applied to the other state-of-the-art optimistic fair exchange protocol in Chapter 2 designed to exchange multiple files between participants. There a verifiable escrow is employed for escrowing the payment (endorsement of an unendorsed e-coin [44]) sent by the participants. All the arguments for the ASW protocol also apply to that. Again, verifiable secret sharing techniques as discussed above will be used instead of the verifiable escrow. The resolution mechanism will be similar to the ones we described for the extended ASW protocol. As for the state semantics, a participant goes to her/his *Resolved* state if (s)he gets other participant's file or e-coin, and goes to his/her *Aborted* state otherwise.

In Section 3.4 we will analyze possible scenarios in an optimistic fair exchange protocol. The first two scenarios will be applicable to this extended protocol types, as we show in Section 3.5, where we analyze protocols that have the same structure as ASW protocol.

## 3.3 Notation

Remember that in a fair exchange scenario, Alice and Bob want to exchange some items fairly. In case of a dispute, they need to contact the arbiters. They are allowed to take the following two actions with the arbiters: *abort* or *resolve*. As noted in Definition 3.2.2, aborting with an honest arbiter takes him to his *Aborted* state, whereas resolving with him would take him to his *Resolved* state.[7] Remember, those states are mutually exclusive, and there is no transition between them, direct or indirect. We assume that the arbiters are autonomous: They do not take into account other arbiters' decision while acting. More formally, the honest participant contacts all arbiters simultaneously (her messages to arbiters do not depend on any response from any of the arbiters).

Let $N$ denote the set of all arbiters, where there are a total of $n$ of them ($|N| = n$). An honest arbiter acts as specified by the protocol. Let $F$ be the set of arbiters who are friends with a malicious participant. Those arbiters are adversarial.[8]

Define two sets $\mathcal{H}_{\mathcal{R}}$ and $\mathcal{M}_{\mathcal{R}}$, which are sets of sets. Any set $H_R \in \mathcal{H}_{\mathcal{R}}$ is a set of arbiters that is sufficient for the *honest* party to *resolve* (as defined in Section 3.2 during the discussion about the connection between arbiters' state and Alice's and Bob's). Similarly, any set $M_R \in \mathcal{M}_{\mathcal{R}}$ is a set of arbiters that is sufficient for the *malicious* party to *resolve*. Therefore, by definition, in case of a dispute, the honest party will end at her *Resolved* state **if and only if** she resolves with **all** the arbiters in **any one** of the sets in $\mathcal{H}_{\mathcal{R}}$ (unless she already is in her *Resolved* state). Similarly, the malicious party will end at his *Resolved* state **if and only if** he resolves with **all** the arbiters in **any one** of the sets in $\mathcal{M}_{\mathcal{R}}$ (unless he already is in his *Resolved* state). *For DAFE protocols, these sets are well-defined by the protocol description,*

[7] This happens only if an arbiter is not already in its *Resolved* or *Aborted* state, respectively.

[8] For example, they may appear as aborted to the honest party, but they may still resolve with the malicious party.

*and do not change once the honest party enters its* Dispute *state.*

A special case of these sets can be represented as thresholds. Let $T_H$ be the number of arbiters the honest party needs to contact for resolution. Similarly, $T_M$ denotes the number of arbiters the malicious party needs to contact for resolution. Thus, the set $\mathcal{H}_{\mathcal{R}}$ is composed of all subsets of $N$ with $T_H$ or more arbiters. Similarly, the set $\mathcal{M}_{\mathcal{R}}$ is composed of all subsets of $N$ with $T_M$ or more arbiters.

Define $R_H$ as the set of arbiters the honest party $H$ has already resolved with, and $R_M$ as the set of arbiters the malicious party $M$ has already resolved with. Also define $R_A$ as the set of all arbiters that are available for $H$ for resolution. Initially, when the dispute resolution begins, we assume that $R_H = \varnothing$, $R_M = F$, and $R_A = N - F$ (and all arbiters are available for resolution to the malicious party). We furthermore have the following actions and their effects on these sets:

**Action 1** ($H$ resolves with an arbiter $X$)**.** *As a result,* $R_H$ *becomes* $R_H \cup \{X\}$.

**Action 2** ($M$ resolves with an arbiter $X$)**.** *As a result,* $R_M$ *becomes* $R_M \cup \{X\}$.

**Action 3** ($H$ aborts with an arbiter $X \in R_A$)**.** *As a result,* $R_A$ *becomes* $R_A - \{X\}$.

**Action 4** ($M$ aborts with an arbiter $X \in R_A$)**.** *As a result,* $R_A$ *becomes* $R_A - \{X\}$.

Note that we do not care what these sets actually are, or whether or not one can find such sets of sets. For our impossibility result, it is enough that conceptually these sets of sets exist.

As in previous work on optimistic fair exchange [7, 96], we assume that the adversary can re-order messages, delay the honest party's messages to the arbiters, insert his own messages, etc. But he cannot delay honest party's messages indefinitely: the honest party eventually reaches the arbiters that he wants to contact initially, and this occurs before the timeout if the protocol uses timeout mechanisms.

### 3.3.1 DAFET Protocols (DAFE Protocols with Timeouts)

In DAFET protocols, we allow for timeouts by giving the arbiters access to loosely synchronized clocks. Instead of actions 3 and 4 above (honest or malicious party aborting), the following action is allowed:

**Action 5** (An arbiter $X \in R_A - R_H - R_M$ times out)**.** *As a result,* $R_A$ *becomes* $R_A - \{X\}$.

Another difference between DAFE and DAFET protocols is the sets $\mathcal{H}_{\mathcal{R}}$ and $\mathcal{M}_{\mathcal{R}}$ being static and dynamic, respectively. DAFE protocols define such sets as static: the overall set of arbiters that needs to be contacted for resolution does not change with time once the honest party enters its *Dispute* state (hence the notation $\mathcal{H}_{\mathcal{R}}$ and $\mathcal{M}_{\mathcal{R}}$). Consider a DAFE type protocol that employs dynamic sets like this: Bob can resolve only with arbiters that Alice has already resolved with. We can think of it as Bob's set initially being empty, and then getting populated. Unfortunately, this protocol does not guarantee timely resolution (unless there is a timeout in the protocol) since Bob may need to wait indefinitely for Alice.

In contrast, we allow DAFET protocols to employ dynamic sets (hence the notation $\mathcal{H}_{\mathcal{R}}(t)$ and $\mathcal{M}_{\mathcal{R}}(t)$). These sets may depend on the timeout and possibly the parties' actions in that particular instance of the protocol. Consider the following two cases as illustrative examples: Some type of protocols allow, let's say, Alice to resolve only after a timeout. Some other type of protocols allow Alice to resolve only with an arbiter that Bob has already resolved with (or vice versa). In analyzing such types of protocols, we will consider $\mathcal{H}_{\mathcal{R}}(t)$ and $\mathcal{M}_{\mathcal{R}}(t)$ as dynamic, letting them change with those actions. We discuss the relation between the use of timeouts and dynamic sets in fair exchange protocols more in Section 3.8.

We will consider any action that results in a change in those sets as new time steps, but there is no need to treat other events as separate time steps since they do

not constitute a significant part of the analysis. Therefore, one can think as if any party can contact any number of arbiters at a given time step $t$. $t = 0$ denotes the time when the dispute resolution begins (the time the honest party enters its *Dispute* state, not the time the protocol execution begins).

Lastly, the set of friends of a malicious party can also change with time, if the adversary is allowed to adaptively corrupt arbiters. In that case, we will use the notation $F(t)$.

## 3.4 Framework for Analysis of DAFE Protocols

In this section, we will provide our framework for analyzing DAFE (and DAFET) protocols. Our framework is composed of different scenarios that can take place during the execution of an instance of a DAFE protocol. Once we have lemmas related to those scenarios stating the necessary (not necessarily sufficient) conditions that need to be satisfied so that the given scenario satisfies the semantic fairness property, then we can analyze different protocol types in the next section. For example, the extended ASW protocol discussed in the previous section will be a protocol of type 0 (in Section 3.5) and will employ scenarios 0 and 0 (depending on which one of Alice and Bob is malicious). Since our results are impossibility or lower bound type of results, it is enough to analyze necessary (but maybe not sufficient) conditions. In all our scenarios (except the last one), we assume that neither party is in the *Resolved* state yet. We consider dynamic resolution sets for our scenario analysis, since static sets are a special case of dynamic sets.

### 3.4.1 Scenario 1: $M$ can Abort

In this scenario, we consider a protocol instance where the malicious party has the ability to abort and resolve. The honest party can abort and resolve too, but the results still apply even if he is restricted to only resolve action. In this scenario, actions 1, 2, and 4 in Section 3.3 are possible. Our results in this section will remain valid regardless of action 3 being possible.

**Lemma 3.4.1.** *Every DAFE protocol instance needs to make sure that there exists a time $t$ when $\forall M_R \in \mathcal{M}_\mathcal{R}(t)$ $\exists H_R \in \mathcal{H}_\mathcal{R}(t)$ s.t. $H_R \subseteq M_R - F(t)$.*

*Proof.* Assume otherwise: At any time in the protocol instance $\exists M_R \in \mathcal{M}_\mathcal{R}(t)$ s.t. $\forall H_R \in \mathcal{H}_\mathcal{R}(t)$ $H_R \not\subseteq M_R - F(t)$. The malicious party can break fairness as follows: He aborts with the set of arbiters $R_A - M_R$, and resolves with the set of arbiters $M_R$. Since no $H_R$ is now a subset of the available arbiters $R_A = M_R - F(t)$, the honest party cannot resolve, while the malicious party already resolved. Thus this protocol instance is unfair (does not satisfy semantic fairness). □

**Corollary 3.4.1.1.** *At any given time $t$ during the protocol instance before the protocol is resolved for $H$, we need $\forall M_R \in \mathcal{M}_\mathcal{R}(t)$ $M_R \not\subseteq F(t)$ since otherwise we need $\exists H_R \in \mathcal{H}_\mathcal{R}(t)$ s.t. $H_R = \varnothing$.*

**Corollary 3.4.1.2.** *We need a time $t$ to exist satisfying $\exists H_R \in \mathcal{H}_\mathcal{R}(t)$ s.t. $H_R \cap F(t) = \varnothing$ since otherwise the lemma cannot be satisfied ($H$ can never resolve).*

**Corollary 3.4.1.3.** *Using threshold-based mechanisms, we need that there exists a time $t$ that satisfies $T_H \leq T_M - |F(t)|$.*

**Corollary 3.4.1.4.** *Using threshold-based mechanisms, at any given time $t$ during the protocol instance before the protocol is resolved for $H$, we need $T_M > |F(t)|$ since otherwise we need $T_H \leq 0$.*

**Corollary 3.4.1.5.** *Using threshold-based mechanisms, we need a time $t$ to exist satisfying $T_H \leq n - |F(t)|$ since otherwise $H$ can never resolve.*

### 3.4.2 Scenario 2: Only $H$ can Abort

In this scenario, we assume that the malicious party has the ability to resolve only, whereas the honest party can abort and resolve. In this scenario, actions 1 to 3 in Section 3.3 are possible (action 4 is not possible).

**Lemma 3.4.2.** *Every DAFE protocol instance needs to make sure that there exists a time $t$ when $\forall M_R \in \mathcal{M}_\mathcal{R}(t)\ \exists H_R \in \mathcal{H}_\mathcal{R}(t)$ s.t. $H_R \subseteq M_R - F(t)$.*

*Proof.* Assume otherwise: At any given time $\exists M_R \in \mathcal{M}_\mathcal{R}(t)$ s.t. $\forall H_R \in \mathcal{H}_\mathcal{R}(t)\ H_R \not\subseteq M_R - F(t)$. The malicious party can break fairness as follows: When $H$ wants to abort the protocol, $M$ lets abort messages to all arbiters in $R_A - M_R$ to reach their destination, but intercept the messages to $M_R - F(t)$ ($F(t)$ really does not matter since his friends will help him anyways). He then resolves with $M_R$. Even if $H$ notices this, he cannot go and resolve since there is no set $H_R \in \mathcal{H}_\mathcal{R}(t)$ that will allow him to. Therefore, this protocol instance also does not satisfy semantic fairness. □

Note that Lemma 3.4.2 is the same as Lemma 3.4.1, and therefore all the corollaries apply to this scenario too.

### 3.4.3 Scenario 3: $H$ can Resolve only after Timeout

In this scenario, aborts can be caused by timeouts only. The malicious party can resolve before and after the timeout, but the honest party can resolve only after the timeout. Therefore, actions 2 and 5 are possible, but not 3 and 4. Action 1 is possible only after the timeout.

**Lemma 3.4.3.** *Every DAFET protocol instance needs to make sure there exists a time $t$ when $\forall M_R \in \mathcal{M}_\mathcal{R}(t)\ \exists H_R \in \mathcal{H}_\mathcal{R}(t)$ s.t. $H_R \subseteq M_R - F(t)$.*

*Proof.* Assume otherwise: At any given time $\exists M_R \in \mathcal{M}_\mathcal{R}(t)$ s.t. $\forall H_R \in \mathcal{H}_\mathcal{R}(t)\ H_R \nsubseteq M_R - F(t)$. The malicious party can break fairness as follows: $M$ resolves with $M_R$ before the timeout. When the timeout occurs, all arbiters in $R_A - R_H - R_M$ to go to their *Aborted* states ($R_H$ being the empty set), which means now $R_A = M_R - F(t)$. But $H$ cannot resolve with the remaining available arbiters and hence this protocol instance is not semantically fair. □

Note that Lemma 3.4.3 is the same as Lemma 3.4.1, and therefore all the corollaries apply to this scenario too.

### 3.4.4 Scenario 4: $M$ already Resolved

All of the scenarios above assumed that both $H$ and $M$ start in their *Working* states when they are performing the resolutions. Yet, it might be perfectly possible that the resolution starts at a point in the protocol where one of the parties has already resolved (and hence is in its *Resolved* state). If $H$ has already resolved, then there is no point to further analyze, since we do not care if the protocol is fair to the malicious party. But if $M$ has already resolved, then we need the following lemma to hold:

**Lemma 3.4.4.** *Every DAFE protocol instance needs to make sure that there exists a time $t$ when $\exists H_R \in \mathcal{H}_\mathcal{R}(t)$ s.t. $H_R \cap F(t) = \varnothing$.*

*Proof.* Assume at all times $\forall H_R \in \mathcal{H}_\mathcal{R}(t)\ H_R \cap F(t) \neq \varnothing$. The malicious party has already resolved but since all possible ways to resolve for $H$ has to go through one of the malicious party's friends, he has no hope of resolving. □

This lemma corresponds to corollary 3.4.1.2 and hence corollary 3.4.1.5 also applies here.

## 3.5 Impossibility Results on DAFE Protocols

The previous section analyzed possible scenarios in DAFE and DAFET protocol instances. In this section, we will analyze DAFE protocol types, using the results from different scenarios that might come up in instances of such protocols. We will conclude that no DAFE protocol can provide fairness under any realistic assumption. DAFET protocols using dynamic sets are possible indeed, and we analyze an existing DAFET protocol in Section 3.7.

For every protocol type, we will consider the following two cases: The case where the honest player plays the role of Alice, and the case where he plays the role of Bob. We denote the set of sets for Alice to resolve as $\mathcal{A}_{\mathcal{R}}(t)$; similarly $\mathcal{B}_{\mathcal{R}}(t)$ is for Bob to resolve. The difference in types of protocols related to these sets being static or dynamic will play a big role. For ease of analysis (and since it is enough for the impossibility results in this section) we will assume the friend list $F(t)$ of the malicious party is static (does not change with time).[9] Since this is a weaker adversary, our impossibility results will also apply when we consider stronger (adaptive) adversaries.[10] We will use $F_A$ to denote friends of a malicious Alice, and $F_B$ to denote friends of a malicious Bob.

In the DAFE protocol types below, we will consider the sets $\mathcal{A}_{\mathcal{R}}(t)$ and $\mathcal{B}_{\mathcal{R}}(t)$ as static (therefore using the notation $\mathcal{A}_{\mathcal{R}}, \mathcal{B}_{\mathcal{R}}$), which eases the use of the lemmas. With static sets, we do not need to consider different times in the protocol instance. A lemma saying there must exist a time $t$ can be simplified by just looking at the initial sets.

[9] This corresponds to the familiar "static corruption model" in many other works.

[10] A more powerful adversary can dynamically corrupt arbiters, having a dynamic set of friends.

### 3.5.1 Protocol 1: Alice and Bob can Abort and Resolve

In this type of protocols, Alice is given the ability to abort and resolve, and Bob is also given the ability to abort and resolve.

**Case 1: Honest Alice vs. Malicious Bob**: This case falls under Scenario 1, which means (for the static case) any DAFE protocol needs to have $\forall B_R \in \mathcal{B}_{\mathcal{R}}$ $\exists A_R \in \mathcal{A}_{\mathcal{R}}$ s.t. $A_R \subseteq B_R - F_B$.

**Case 2: Malicious Alice vs. Honest Bob**: This case also falls under Scenario 1, which means (again for the static case) any DAFE protocol needs to have $\forall A_R \in \mathcal{A}_{\mathcal{R}}$ $\exists B_R \in \mathcal{B}_{\mathcal{R}}$ s.t. $B_R \subseteq A_R - F_A$.

These two cases lead to the conclusion that every protocol instance needs two sets $A_R \in \mathcal{A}_{\mathcal{R}}$ and $B_R \in \mathcal{B}_{\mathcal{R}}$ s.t. $A_R = B_R \subseteq \{\text{trusted arbiters}\}$. These arbiters must be trusted, and so there is no point in distributing the arbiters. It is even worse: If any of these arbiters are corrupted, the DAFE protocol fails to be fair. Therefore, no such realistic DAFE protocol can exist.

When considering threshold-based schemes, this corresponds to the requirement that $T_B \leq T_B - F_A - F_B$, which means no party should have any friends for such a protocol to be fair. If even one arbiter is corrupted, the protocol becomes unfair. Therefore, no such realistic DAFE protocol can exist. Since set-based mechanisms cover threshold-based ones, we will not discuss threshold-based schemes separately again unless necessary. All impossibility results proven for set-based mechanisms directly apply in the context of threshold-based ones.

### 3.5.2 Protocol 2: Only one party can Abort

In this type of protocols, Alice is given the ability to abort and resolve, whereas Bob is given only the ability to resolve. Analysis of protocols that are symmetric to this type of protocols (where Bob can abort and resolve, and Alice can only resolve) obviously

yields to the same conclusions.

**Case 1: Honest Alice vs. Malicious Bob**: This case falls under Scenario 2, which requires that DAFE protocols need to make sure $\forall B_R \in \mathcal{B}_\mathcal{R}\ \exists A_R \in \mathcal{A}_\mathcal{R}$ s.t. $A_R \subseteq B_R - F_B$.

**Case 2: Malicious Alice vs. Honest Bob**: This case falls under Scenario 1, which means any DAFE protocol needs to have $\forall A_R \in \mathcal{A}_\mathcal{R}\ \exists B_R \in \mathcal{B}_\mathcal{R}$ s.t. $B_R \subseteq A_R - F_A$.

We can conclude as in the previous section (Section 3.5.1) that every protocol instance needs two sets $A_R \in \mathcal{A}_\mathcal{R}$ and $B_R \in \mathcal{B}_\mathcal{R}$ s.t. $A_R = B_R \subseteq \{\text{trusted arbiters}\}$. Again, this means there is no point in distributing the arbiters in terms of trust. Remember that threshold-based versions have the same impossibility.

Unfortunately, the versions of the state-of-the-art optimistic fair exchange protocols we analyzed in Section 3.2.1 *without* any timeouts fall under this protocol category. Note that, this means, using static resolution sets and autonomous arbiters, those protocols cannot be extended to use multiple arbiters and remain fair.

## 3.6 Relaxing Autonomous Arbiters Assumption

In this section, we extend our framework by relaxing the autonomous arbiters assumption to allow for ordered aborts by the honest party and therefore include a broader range of protocols in our framework. We still assume that the honest arbiters do not try to communicate, but now the honest parties can contact the arbiters following some particular order. We immediately notice that the only places where we need that assumption are Scenario 2 and Protocol 2. Results about all other scenarios and protocols stay unchanged when we do the relaxation by removing the explicit *Dispute* state in the ITM definitions of the honest participants (Alice and Bob), thus allowing them to contact the arbiters with some specific order. Yet, we still are not

considering byzantine fault tolerance or secure multi-party computation techniques.

### 3.6.1 Scenario 2 Revisited

In Section 3.4.2, we analyzed the scenario in which the malicious party has the ability to resolve only, whereas the honest party can abort and resolve. We analyzed that scenario using the autonomous arbiters assumption. Below, we will remove the requirement that arbiters are contacted simultaneously, and revisit our analysis.

#### Scenario 2 with Threshold-based Mechanisms

Here, we are limiting our protocol instances to the case where only threshold-based mechanisms are used.[11] This means, the sets $\mathcal{H}_{\mathcal{R}}(t)$ and $\mathcal{M}_{\mathcal{R}}(t)$ are of the specific form we have described before. Remember, the set $\mathcal{H}_{\mathcal{R}}(t)$ is composed of all subsets of $N$ with $T_H$ or more arbiters. Similarly, the set $\mathcal{M}_{\mathcal{R}}(t)$ is composed of all subsets of $N$ with $T_M$ or more arbiters. $T_H$ and $T_M$ are the corresponding thresholds.

**Lemma 3.6.1.** *Every DAFE protocol instance needs to make sure there exists a time $t$ when $T_H \leq T_M - |F(t)|$.*

*Proof.* Assume otherwise: At all times $T_H > T_M - |F(t)|$. Malicious party can break fairness as follows: When $H$ wants to abort the protocol (as directed by the protocol, most probably triggered by an incorrect input from the malicious party), $M$ waits until $H$ aborts with $n - T_H + 1$ arbiters. $H$ can no longer resolve after this point since there are less than $T_H$ arbiters left in the set of available arbiters $R_A$. At this point, $M$ intercepts any more abort messages from $H$ and resolves with $T_M - |F(t)|$ honest arbiters (as well as $|F(t)|$ friends). Therefore, this protocol instance is unfair (does not satisfy semantic fairness). □

[11] Section 3.6.1 removes the threshold limitation and allows for any set-based resolution mechanism.

Notice that Lemma 3.6.1 is the same as Corollary 3.4.1.3. Therefore, Corollaries 3.4.1.4 and 3.4.1.5 also apply here.

**Scenario 2 General Case**

Now, we remove all the restrictions we made on our scenario in the previous sub-scenarios. This means, we allow for any set-based resolution mechanism, and we even allow the protocol to specify an order of arbiters for aborting, possibly depending on the execution of the protocol instance. One can think of it as the honest party aborting with one arbiter at every time step, and reconsidering his decision to abort each time. Therefore, the arbiters are no longer completely autonomous.

**Lemma 3.6.2.** *Every DAFE protocol instance needs to make sure that at all times $t$ $\forall M_R \in \mathcal{M}_{\mathcal{R}}(t)$ $M_R \not\subseteq F(t)$ (before $H$ has resolved) AND there exists a time $t$ when $\exists H_R \in \mathcal{H}_{\mathcal{R}}(t)$ s.t. $H_R \cap F(t) = \varnothing$.*

*Proof.* Assume there exists a time when $\exists M_R \in \mathcal{M}_{\mathcal{R}}(t)$ $M_R \not\subseteq F(t)$ (before $H$ has resolved). Malicious party can break fairness as follows: When $H$ wants to abort the protocol, $M$ lets him abort with all the arbiters. Then, he goes and resolves with $M_R$, all members of which are his friends.

Now assume at all times $\forall H_R \in \mathcal{H}_{\mathcal{R}}(t)$ $H_R \cap F(t) \neq \varnothing$. Malicious party can break fairness by just resolving with any $M_R \in \mathcal{M}_{\mathcal{R}}(t)$. Since all possible ways to resolve for $H$ has to go through one of the malicious party's friends, he has no hope of resolving. □

In this general scenario, as in the previous cases, we would like to be able to prove that any DAFE protocol instance needs to make sure there exists a time $t$ when $\forall M_R \in \mathcal{M}_{\mathcal{R}}(t)$ $\exists H_R \in \mathcal{H}_{\mathcal{R}}(t)$ s.t. $H_R \subseteq M_R - F(t)$. Even though this seems a very plausible and realistic conclusion, several problems arise with its proof.

The general idea is to use an adversary very similar to the one in Section 3.4.2. So, the adversary will let $H$ to abort with any arbiter in $R_A - M_R$. Then, if $H$ wants to abort with an arbiter in $M_R - F(t)$, $M$ will intercept and resolve with $M_R$. The problem is that this works depending on the order of aborts. There might be a possible protocol construction and order specification that makes sure $H$ can still resolve once he detects this behavior. We do not know of and could not come up with such a construction, due mostly to the fact that $F(t)$ is unknown to the honest party, and hence designing a protocol instance using an order that works without knowing $F(t)$ seems impossible. Even though the order may work for some protocol instances, having an order that works with high probability (that works on all but negligible fraction of protocol instances) does not seem possible. Furthermore, the moment we allow for more powerful adversaries, since the order of arbiters for the honest participant to abort is public, the adversary might "bribe" some "key" arbiters to become his friends and make sure the ordering fails to provide fairness (in the dynamic/adaptive corruption model). We admit that we have no proof for this general case with less powerful adversaries, but we conjecture that the same predicate for scenario 3.4.2 as before will hold.

### 3.6.2 Protocol 2 Revisited (More Impossibility Results)

In this type of protocols, Alice is given the ability to abort and resolve, whereas Bob is given only the ability to resolve. Analysis of protocols that are symmetric to this type of protocols (where Bob can abort and resolve, and Alice can only resolve) obviously yields to the same conclusions. The predicate for case 1 changes when we relax our autonomous arbiters assumption. Case 2 stays the same. Remember, the resolution sets we consider here are static.

**Case 1: Honest Alice vs. Malicious Bob**: This case falls under Scenario

2, which requires special treatment when arbiters are not contacted simultaneously for aborting. For threshold-based mechanisms, every DAFE protocol needs to have $T_A \leq T_B - |F_B|$. For the most general case of DAFE protocols, we need $\forall B_R \in \mathcal{B}_\mathcal{R}$ $B_R \not\subseteq F_B$ AND $\exists A_R \in \mathcal{A}_\mathcal{R}$ s.t. $A_R \cap F_B = \varnothing$ (see Lemma 3.6.2 in Section 3.6.1).

**Case 2: Malicious Alice vs. Honest Bob**: This case falls under Scenario 1, which means any DAFE protocol needs to have $\forall A_R \in \mathcal{A}_\mathcal{R}$ $\exists B_R \in \mathcal{B}_\mathcal{R}$ s.t. $B_R \subseteq A_R - F_A$. Remember, Corollary 3.4.1.3 (using threshold-based mechanisms) require $T_B \leq T_A - |F_A|$.

Regarding DAFE protocols using threshold-based arbiter resolution mechanisms, we can conclude (from the two cases above) that no such meaningful protocol can exist ($T_A \leq T_B - |F_B|$ and $T_B \leq T_A - |F_A|$ gives $T_A \leq T_A - |F_A| - |F_B|$, which means all the arbiters need to be trusted). Hence, there is no point in distributing the arbiters in terms of trust. It is even worse since we need to trust every single arbiter, and the protocol cannot be fair even if only one arbiter is corrupt.

Regarding general set-based DAFE protocols, we cannot conclude an immediate impossibility. But following our discussion above, we conjecture that no such useful protocol can exist.

Unfortunately, as we have shown in Section 3.2.1, the versions of the state-of-the-art protocols we analyzed in Section 3.2.1 *without* any timeouts fall under this protocol category. So the impossibility with threshold-based mechanisms, and our conjecture apply to very common real cases, even when the arbiters are not contacted simultaneously by the honest party.

## 3.7 Applying DAFET Framework to Prove Optimality

In this section, we analyze an existing DAFET protocol that uses dynamic resolution sets: The set of arbiters needed by a party for resolution changes during the course of the execution of the protocol instance. By adjusting resolution sets reactively, this protocol can provide semantic fairness.

**AV Protocol [11]** This protocol is due to Avoine and Vaudenay (AV) [11]. In this protocol, timeouts are used for aborting (it is a DAFET protocol). It is a three-step protocol in which Alice starts by sending verifiable secret shares encrypted under each arbiter's public key. Then, Bob responds with his secret, and Alice responds with her secret. To resolve, Bob contacts $k$ arbiters to get the decrypted shares and reconstruct the secret of Alice (where $k$ is the threshold for the secret sharing scheme). Before giving the decrypted share, each honest arbiter asks for the secret of Bob.[12] Hence, the set $\mathcal{B}_{\mathcal{R}}(t)$ contains all subsets of $N$ with $k$ or more arbiters and $\mathcal{A}_{\mathcal{R}}(t)$ is initially empty[13].

The state semantics obviously coincide with our 3-state definition. The participants either succeed in obtaining the other party's exchange item and hence end at their *Resolved* state, or they fail to do so and end at their *Aborted* state. The honest arbiters will either help both participants, or abort at the timeout and help neither.

Even though in the AV protocol the honest arbiters directly contact Alice when Bob resolves with them, we can see it as the arbiters storing Bob's secret, and Alice contacting them to obtain Bob's secret later on. Since Alice can only resolve after Bob,

[12]The user should refer to [11] for any more details.

[13]It does not contain the empty set, it is empty. This means no set of arbiters is sufficient for Alice to resolve.

and Bob has to resolve before the timeout, it is safe to think of this protocol as letting Alice to resolve only after the timeout. Unlike the protocols in Section 3.5 which were proven impossible to be fair, this protocol uses dynamic resolution sets that help it achieve fairness (we talk about the relationship between timeouts and dynamic resolution sets in Section 3.8). So, sets $\mathcal{H}_{\mathcal{R}}(t)$ and $\mathcal{M}_{\mathcal{R}}(t)$ change according to the following additional rule regarding the actions (remember the actions in Section 3.3):

**Action 6** (Bob resolves with an arbiter $X \in R_A$)**.** *As a result, the set* $\{X\}$ *is added to the set of sets* $\mathcal{A}_{\mathcal{R}}(t)$*.*

This rule is there since in the AV protocol, when Bob contacts an honest arbiter, that arbiter contacts Alice and sends Bob's whole secret. It guarantees that the moment a malicious Bob resolves with any honest arbiter, Alice is guaranteed to be able to resolve. Let us analyze the two cases and see how this protocol satisfies the lemmas regarding scenarios.

**Case 1: Honest Alice vs. Malicious Bob**: This case falls under Scenario 3, which means any DAFET protocol needs to make sure there exists a time when $\forall B_R \in \mathcal{B}_{\mathcal{R}}(t)\ \exists A_R \in \mathcal{A}_{\mathcal{R}}(t)$ s.t. $A_R \subseteq B_R - F_B$.

**Case 2: Malicious Alice vs. Honest Bob**: Depending on at which point of the protocol the resolution begins, malicious Alice might have already resolved, thus this case falls under Scenario 4, which requires that there exists a time when $\exists B_R \in \mathcal{B}_{\mathcal{R}}(t)$ s.t. $B_R \cap F_A = \varnothing$.

**Lemma 3.7.1.** *AV protocol cannot provide semantic fairness unless for all times* $t$ $\forall B_R \in \mathcal{B}_{\mathcal{R}}(t)\ B_R \not\subseteq F_B$ *AND for some time* $t\ \exists B_R \in \mathcal{B}_{\mathcal{R}}(t)$ *s.t.* $B_R \cap F_A = \varnothing$*.*

*Proof.* It follows directly from the analysis of the cases above using corollary 3.4.1.1 for case 1. □

The AV protocol achieves semantic fairness using dynamic sets as follows: The

set $\mathcal{A}_{\mathcal{R}}(t)$ is initially empty. When Bob contacts an arbiter $X$, action 6 above takes place, and hence the set $\{X\}$ is added to the set of sets $\mathcal{A}_{\mathcal{R}}(t)$ (the threshold for Alice effectively becomes 1). Therefore, once Bob contacts an honest arbiter (not one of his friends), then Alice is guaranteed to be able to resolve. This saves an honest Alice against a malicious Bob (case 1). In case 2, as long as Bob can find a set of honest arbiters that he can resolve with, he is saved against malicious Alice.

Actually, the AV protocol [11] uses threshold-based mechanisms instead of set-based ones, therefore we have the following corollary:

**Corollary 3.7.1.1.** *AV protocol cannot provide semantic fairness unless* $|F_B| < T_B$ *AND* $T_B \leq n - |F_A|$.

It is important to notice that the AV paper [11] proves essentially the same result: They prove that the same bound is also sufficient for their protocol. Thus, we have proven that the bounds proven in that paper are tight and hence the protocol is optimal in that sense. Furthermore, this result is applicable to all protocols of the same type; no DAFET protocol of the same type can achieve better bounds. In particular, the same technique of employing multiple autonomous arbiters can be used on [7] and our construction in Chapter 2 (as described in Section 3.2.1) to convert their timeout-based versions to DAFET protocols, and the same lemma will hold. This shows how our framework can easily be applied to prove optimality of a protocol and extended to other protocols of the same type.

As the corollary immediately reveals, when using $n$ arbiters, to obtain maximum tolerance, one should set the threshold for Bob $T_B = n/2$ so that the protocol tolerates up to $n/2-1$ friends of each participant. Of course, this greatly reduces the efficiency of the resolution of the optimistic fair exchange protocol.

Even with such an interesting modification to the protocol, Vaduenay protocol needs to make the following assumption [11]: The threshold for the secret sharing

scheme used for distributing the arbiters must be greater than the number of friends the malicious party can have. This limits the applicability of the protocol in real scenarios. If the threshold is set very high to tolerate worse situations, then the efficiency greatly decreases. Otherwise, if the threshold is low, than the tolerance against malicious behavior is low.

## 3.8 Discussion: Timeouts and Dynamic Resolution Sets

As we have proved in Section 3.5, no realistic DAFE protocol can provide fairness, whereas Section 3.7 shows an existing DAFET protocol that employs timeouts. Therefore, we can conclude that timeouts play an important role in optimistic fair exchange protocols when we would like to employ multiple autonomous arbiters. Even without completely autonomous arbiters, Section 3.6.2 shows an impossibility of DAFE protocols using threshold-based mechanisms, and even with set-based mechanisms, it is not clear how such a DAFE protocol can be constructed.

Timeouts are tied to the use of dynamic sets in general (as we did for DAFET protocols). When only one party can resolve before the timeout, static resolution sets lose their meaning since the resolution set for the party who cannot resolve before the timeout is empty until the timeout. That set gets defined only after the timeout, which results in that set being dynamic in a very basic sense. The dynamism prevents the adversary from coming up with a strategy that violates fairness. As shown in Section 3.7, this helps AV protocol achieve semantic fairness. Of course, a careful protocol design is still necessary since timeouts and dynamically changing sets by themselves do not mean that the protocol will be trivially fair. One may further argue that dynamically changing resolution sets is a more important concept that

plays a big role in this (im)possibility result, but it is easy to see that timeouts are natural mechanisms to achieve this dynamism.

This suggests that even though timeouts may not be a nice feature in terms of system design, it really helps when the system needs to be extended to use multiple autonomous arbiters (together with the use of dynamically changing resolution sets).

## 3.9 Conclusion and Future Work

In this chapter, we presented a framework to analyze DAFE protocols, which are natural extensions of optimistic fair exchange protocols to make them use multiple autonomous arbiters (those who do not communicate with each other). Autonomy is useful for realistic (efficient) protocols, especially in P2P settings. Using the presented framework, we answered two open questions since [11]. We have proved that DAFE protocols (optimistic fair exchange protocols that employ multiple autonomous arbiters and does not have timeout mechanisms) cannot provide fairness in a realistic setting. Even when we extended our framework by relaxing the autonomy assumption about the arbiters, we found out that even broader classes of optimistic fair exchange protocols fall under our impossibility results. We then switched to the DAFET model to include timeouts and dynamically changing sets of arbiters to resolve with. We analyzed one existing DAFET protocol [11] using our framework and proved that the previous bounds on the required number of honest arbiters are optimal. No DAFET protocol of the same type can achieve better bounds, since our framework can easily be used to come up with generalized results. We also showed that timeouts and dynamic resolution sets play an important role in the design of such distributed arbiter fair exchange protocols.

Unfortunately, this means many optimistic fair exchange protocols that want to efficiently distribute their arbiters may need to employ synchronized clocks. And even

in this case, they cannot hope to require fewer honest arbiters than the Avoine and Vaudenay protocol [11]. If they do not want to employ synchronized clocks, then they may need to employ costly solutions like secure multi-party computation or Byzantine agreement.

One may want to settle down for weaker security guarantees against weaker adversaries to achieve cheaper solutions than Byzantine agreement. Using Byzantine fault tolerance techniques in [1], the arbiters can keep updating some value that is related to the resolution semantics of the fair exchange. Unfortunately, when aborts are considered, it is not clear if the same techniques can be applied here. We leave research in this direction as an open problem.

Finally, our techniques may be applicable to other functionalities that can be implemented using secure multi-party computation. By designing an appropriate framework, we may prove more general results about achieving the same functionality using autonomous multiple parties. We leave such a generalization as an interesting open problem.

# Chapter 4

# Computing in the Cloud

## 4.1 Introduction

Many tasks exhibit an arbitrarily high appetite for computational resources. Distributed systems that coordinate computational contributions from thousands or millions of participants have become popular as a way to tackle these challenges. Examples include systems such as SETI@home [138] and Rosetta@home [133], which seek to analyze huge amounts of data in the search for extra-terrestrial life and a better understanding of protein folding, respectively. In these systems, every additional computational element added to the system provides greater utility.

This study is motivated by our efforts to build peer-to-peer systems that rely on cryptographic electronic cash (e-cash) to provide incentives for participation [16]. Such a system would prevent free-riding without sacrificing the privacy of its participants. Using results from our implementation in Chapter 7, we realized that the deposits at the bank can cause a bottleneck. With lots of e-cash transactions going on in a high-churn peer-to-peer (P2P) environment, the bank can be overloaded. One way of reducing this load is the barter protocol described in Chapter 2, since no money changes hands if everything goes well. Another way is to outsource the bank's

job to untrusted contractors, possibly the same set of users as the underlying P2P system.

The naive solution is to simply give each peer a program to run (such as an e-coin verifier) and the input to this program (an e-coin). The peer would run the program and report the answer. There are several problems with this approach. First, without a reward, there is no incentive for participants to do any work. Second, even if the participants were compensated for their contribution, there is no incentive to perform the computation faithfully. Peers may report an answer at random or, perhaps, report an answer that they know *a priori* to be the most likely output of the computation (e.g., that most e-coins are valid). Worse, if participants are malicious, they may choose to behave irrationally in order to force the bank to perform more work or accept incorrect results.

These problems are not limited to our e-cash application. SETI@home users have developed their own clients, for both malicious and selfish reasons [114, 127] (see Section 4.1.1). Multi-player games cannot assume that players will not modify their clients to give themselves an in-game advantage. In general, this problem lies at the heart of cloud computing.

Our solution assumes that there is some currency or credit system with which we can reward or fine contractors depending on their performance. This could be a reputation or credit system in which good contractors are awarded higher scores, or an actual currency which can be exchanged for some other services. This allows us to set incentives such that rational contractors will compute jobs correctly.

In this chapter, we analyze how to the boss can set fines and rewards, and how often it will have to double-check the contractors' results in order to enforce the incentive structure. In Section 4.3, we define a game-theoretic framework to analyze different scenarios. Section 4.4 shows how to use collision resistant hash functions

to increase the probability of getting a correct answer without increasing the fine-to-reward ratio and the amount of double-checking. In Section 4.5 we examine means of performing checks on contractors' answers, and consider outsourcing the same computation to multiple contractors, double-checking only if they disagree, as a way to reduce the amount of centralized double-checking. We also look at the effect of offering a bounty to a user who catches another contractor returning a wrong answer. Finally, in Section 4.6, we examine how to limit the damage that can be caused by malicious and colluding contractors, who seek to maximize the amount of centralized double-checking, or decrease the accuracy of submitted results.

### 4.1.1 Related Work

Resource-sharing cluster systems such as Spawn [147], Popcorn [132], and Tycoon [98] focus on the efficient allocation of grid resources by providing auction mechanisms which award distributed resources to the highest bidder. Auctions provide a way to stem demand as computation becomes more expensive. However, these systems typically assume a federated—and friendly—environment where many parties wish to share a pool of trusted resources. Once awarded, resources are assumed to be available for use by the winner, without concern for malicious entities.

Our work has more in common with public-resource computing systems such as Distributed.net [62] and BOINC [29], which parcel and distribute computation to vast armies of volunteer users. BOINC provides scientific projects such as SETI@Home [138] and Rosetta@Home [133] with computational resources drawn from the idle CPU cycles of its users' home PCs, and its projects have attracted millions of participants. Greater participation is incentivized through a point system that rewards users who complete more work units with higher status on "leaderboards" published on the web.

BOINC's credits are not fungible—they are useful only for social status—yet even

this incentive has greatly motivated participation, leading some to develop their own clients in an effort to claim more credit [146, 114]. In one case, a SETI@Home user developed an "optimized" client which returned outputs irreproducible by the official client, yet were otherwise indistinguishable. In another case, a patched client was released that simply performed no computation, returning bogus results [114, 127]. These examples and others provide inspiration for our model, which aims to address the problem of malicious and "corner-cutting" contractors who seek greater rewards by deviating from officially-sanctioned methods.

Systems based on Byzantine fault tolerance (BFT) [2] provide safety and liveness guarantees given a certain tolerable fraction of malicious users; typically at least two-third of participants must act correctly. The BAR model [100] provides incentive-compatible BFT primitives to extend these guarantees to both altruistic (i.e., correct) and rational nodes that may deviate from suggested protocols in pursuit of greater utility. Like these approaches, we also aim to incentivize rational nodes, but do not assume a quorum of correct nodes; instead we focus on incentives and probabilistic guarantees on accuracy that apply for varying fractions of altruistic and malicious users.

Checking intermediate computations has also been discussed for the problem of inverting one-way functions, where predefined intermediate steps are checked [85], and in general by redoing the computation until a randomly chosen intermediate step [72]. Molnar [114] suggests that contractors be required to provide a hash of the results of intermediate computations in order to force them to use the official algorithm. This is very similar to the approach we discuss in Section 4.4. However, the idea of using hashes was not formalized, and there was no discussion of how to combine this approach with incentive strategies for rational contractors.

## 4.2 Model

A central authority, the *boss*, will reward *contractors* to perform computational tasks, or *jobs*, on its behalf. The goal is to reduce the demand on the boss's own computational resources. We assume that contractors continually request new job assignments from the boss, but that they may freely choose when to stop requesting new jobs.

The boss will reward a contractor $r$ for correctly completing a job. If the boss finds out that the contractor returned an incorrect result, the boss will fine the contractor $f$, which is subtracted from the contractor's accumulated earnings. The boss will not assign a job to a contractor unless the contractor has enough credit to pay the potential fine. As a result, we are concerned with reducing the fine-to-reward ratio ($f/r$): too high a ratio makes it harder for contractors to participate. As we will later see, there is a trade-off between the work the boss has to do and the $f/r$ ratio.

Our definition of a *job* captures any efficiently computable task and its inputs. For the e-coin verification scenario, the only way the boss can make sure that a contractor properly verified an e-coin is to reverify it herself. Similarly, for the Folding@home project, the boss must refold the protein. For jobs in NP, the verification is much easier. However, the boss can only check an answer if the output of the computation is deterministic. If the job uses a randomized algorithm, the boss must provide the contractor with a random tape (i.e., a seed to a pseudo-random number generator).

The results of some jobs may be easier to predict than others. Consider a naïve decision problem formulation of the SETI@home project, "Is alien life detectable in this radio telescope data?" Or, for the e-coin verification task, "Do these values represent a valid e-coin?" A rational contractor may decide to conserve its computational resources and simply guess the most likely answer ("no" and "yes", respectively). We describe a hashing technique to detect incorrect answers, even for such highly skewed answer distributions.

Our payment- and penalty-based incentives assume the presence of an underlying economic framework in which the boss can enforce fines and rewards. In [16], peers use e-cash to exchange files; if the bank wishes to outsource tasks, it can easily increase and deduct account balances directly. BOINC similarly directly rewards users with credit that raises a user's ranking on the leadership board. A service provider boss (e.g., a storage server) might reward contractors by providing them better service (e.g., more storage), and fine them by reducing the service provided (e.g., limiting their storage space). Real currencies might also be used if contractors offer the fine amount as deposit with the boss. Our model assumes only that a boss is able to withdraw $f$ from and pay $r$ to contractors.

## 4.3 Basic Construction

Consider a contractor who has just been assigned a job by the boss. He faces two options: first, he may perform the job honestly, and receive a reward $r$. If we define the cost of computing a single job using the algorithm provided by the boss as $\mathsf{cost}(1)$, the expected utility $u(1)$ of an honest contractor is $u(1) = r - \mathsf{cost}(1)$. In this case, we assume that the boss sets $r$ large enough to provide positive utility for the contractor, or he will refuse the job.

The contractor's second option is to return an output using an algorithm different from that specified by the boss. This might be possible, for example, if the contractor possesses *a priori* knowledge of the output distribution: it can simply guess the most likely output. Or, more generally, suppose the contractor has access to an alternative algorithm which provides a correct output with probability $q$ (e.g., SETI@home "optimized" client). Here, the contractor may still receive $r$, but risks being fined $f$ if the boss discovers he has submitted an incorrect result.

We denote the probability that this *lazy contractor* will be caught submitting

an incorrect result as $p$. However, we do not assume that the boss will be able to detect each incorrect result submitted and fine the guilty contractor: since checking the correctness of a submitted result may unduly waste computational resources. (We defer discussion of methods for checking results to Section 4.5.) Thus we can decompose $p$ into two different values: the probability that the contractor's result is incorrect, and the probability that the result will be checked, when it is incorrect.

$$p = Pr[check \text{ — } incorrect]\ Pr[incorrect]$$

We can analyze these two probabilities separately. First, let $c$ be the probability that a contractor's result will be checked, conditioned on that contractor returning an incorrect result: $c = Pr[check \text{ — } incorrect]$. The check can be performed by the boss or by other contractors. This also describes the case when the probability of a check is independent of the contractor's answer (e.g., if the boss simply checks a fraction $c$ of submitted outputs itself).

Next, we return to our definition of $q$, the probability of the contractor returning the correct answer using an alternate method. Clearly the probability that the contractor's answer is incorrect is $1 - q$. Thus

$$p = c(1 - q)$$

We also define the cost of the alternate method for obtaining a correct result with probability $q$ as $\mathsf{cost}(q)$. We assume this cost is at most $\mathsf{cost}(1)$—otherwise, the contractor would simply run the suggested algorithm—and at least 0.

We can now define the expected utility $u(q)$ of a contractor, taking into account the probability $p$ of being caught and his cost, as

$$u(q) = r(1 - p) - fp - \mathsf{cost}(q)$$

The contractor will receive a reward unless he is caught cheating, in which case he

will be fined. Note that when $q = 1$, the contractor is performing the job correctly, and thus $p = 0$ and $u(q) = u(1)$ from our previous definition.

For a rational contractor, selecting a value of $q < 1$ and earning the expected utility $u(q)$ may present a lucrative choice, resulting in a potentially incorrect output. However, the boss can provide incentives to perform jobs correctly by setting $f$, $r$, and $c$.

**Theorem 4.3.1.** *If the boss sets the fine-to-reward ratio to $f/r \geq (1-p)/p$ where $p = c(1-\varepsilon)$ then a rational contractor will return correct outputs at least $\varepsilon$ of the time.*

*Proof.* To prove this, we need to show that for any $q' < \varepsilon$, the resulting utility $u(q') < u(\varepsilon)$. Since we cannot argue about the cost functions of contractors realistically (contractors may value their resources differently, and it might also depend on the state of the contractor like his current load), we want to show $\forall q' < \varepsilon, u(q') \leq 0$. Remember, $u(q') = r(1-p') - fp' - \mathsf{cost}(q')$, where $p' = c(1-q')$. If we set $f/r \geq 1/p - 1 \geq 1/p' - 1$, then we guarantee that $r(1-p') - fp' \leq 0$. Thus, given such an $f, r, c$, any contractor who is not correct with probability at least $\varepsilon$ will have negative utility. This means any rational contractor will either perform the job with accuracy at least $\varepsilon$, or will refuse to do the job. □

**Corollary 4.3.1.1.** *Any rational contractor will use the least costly algorithm that provides correct answers with at least $\varepsilon$ probability.*

## 4.4 Accuracy and Hash Functions

By setting the fine-to-reward ratio as above, the boss can require rational contractors to compute jobs correctly above a certain minimum accuracy requirement. Yet, obtaining high accuracy might require an infeasibly high fine-to-reward ratio, and for

some applications even a small fraction of inaccurate results might be unacceptable.

Our concern is that there might be some alternate algorithm that costs the contractor very little (in terms of computation), and that produces the correct answer with some fairly high probability $\epsilon$ (e.g., guessing a coin to be valid in the e-cash verification scenario). To prevent the contractor from using such an algorithm, we might have to set the fine-to-reward ratio unreasonably high.

Ideally we would like to ensure that the contractor actually runs the algorithm that we choose. Thus, instead of simply returning an answer, we could ask the contractor to send us the results of every intermediate computation. If we assume that the intermediate computations are small enough steps that the only way to get the correct intermediate result is by actually running the appropriate computation, then this will be sufficient to convince the boss that the contractor has run the computation correctly. Finally, to prevent the contractor from having to send a very large amount of information, we have him use a cryptographic hash function to hash all of this information into one short string. More formally:

**Definition 4.4.1.** *An algorithm is assumed to be composed of a finite number of* ***atomic operations****. Each atomic operation is assumed to take a state information and output another state information. The* ***inner state*** *of an algorithm is defined as the concatenation of all the input/output states of the atomic operations of the algorithm, along with the definition of the algorithm in terms of atomic operations. The original algorithm for a given job is the one prescribed by the boss to the contractor. A hash function deterministically maps the inner state of an algorithm to a random $l$-bit string. Define* ***negligible*** *probability $neg = O(2^{-l})$.*

We would like to assume that all algorithms which produce the correct result either have cost $\mathsf{cost}(1)$ or negligible success probability. However, there is always a potential mixed strategy which with some probability runs the original algorithm and

with some probability makes a random guess of the inner state. Thus, we make the following assumption:

**Assumption 4.4.1** (Unique Inner State Assumption)**.** (FOR INPUT DISTRIBUTION $D$ AND NEGLIGIBLE $neg'$)

*Let* $\mathsf{cost}(1)$ *be the cost of the original algorithm. We assume that any algorithm which has expected cost* $\gamma\mathsf{cost}(1)$ *(given a random input from* $D$*) will produce the correct inner state with probability at most* $\gamma + (1-\gamma)neg'$ *(provided* $0 \leq \gamma \leq 1$*).*

Then we can say that a similar statement holds even after the application of the hash function:

**Theorem 4.4.1.** *Let* $\mathsf{cost}(1)$ *be the cost of the original algorithm. Let* $D$*,* $neg' < neg$ *be such that the unique inner state assumption holds. Then under unique inner state assumption and the random oracle model*[1]*, any algorithm which when given a random input from* $D$ *has expected cost* $\delta\mathsf{cost}(1) < \mathsf{cost}(1)$ *will produce the correct hash of the inner state with probability at most* $\delta + (1-\delta)neg$ *(provided* $0 \leq \delta \leq 1$*).*

*of Theorem 4.4.1.* Consider the operation of the algorithm on a particular input. There are two ways that an algorithm can output the correct hash value. First, the algorithm might have queried the random oracle (to obtain the hash output) at the same inner state value as the original algorithm. That means by the unique inner state assumption that this operation must have cost $\gamma\mathsf{cost}(1)$ and succeed with probability $\gamma + (1-\gamma)neg'$. Second, the algorithm might have produced the same hash without querying the random oracle at using the correct inner state. This has only negligible probability under the random oracle model. We have said that the algorithm has expected cost $\delta\mathsf{cost}(1)$. That means that it can be taking the first approach (following the correct probability) on at most $\frac{\delta}{\gamma}$ fraction of the inputs. Thus, on all other inputs,

[1]The random oracle model is commonly used in cryptography. It assumes that the hash function behaves like a truly random function.

it has at best *neg* probability of success. That means that it's total success probability can be at most $\frac{\delta}{\gamma}(\gamma + (1-\gamma)neg') + (1-\frac{\delta}{\gamma})neg \leq \delta + (1-\delta)neg$. □

Finally, we conclude that if we set the parameters appropriately, a rational contractor will always use the original algorithm.

**Theorem 4.4.2.** *Suppose that definition 4.4.1 holds for our input distribution. If* $\frac{f}{r} \geq \frac{1}{c}$*, and* $r > \mathsf{cost}(1)$ *and* $c > neg/(1-neg)$*, then a rational contractor will use the original algorithm for the job.*

*Proof.* Running the original algorithm results in utility $r - \mathsf{cost}(1)$. By theorem 4.4.1, any other algorithm will either have cost greater than $\mathsf{cost}(1)$ (and thus obviously lower utility), or will have cost $\delta\mathsf{cost}(1) < \mathsf{cost}(1)$ and success probability $\delta+(1-\delta)neg$. That means the total utility will be $(\delta + (1-\delta)neg)r - (1-\delta-(1-\delta)neg)cf + (1-\delta-(1-\delta)neg)(1-c)r - \delta\mathsf{cost}(1)$. If $f, r, c$ satisfy the conditions described in the theorem, then this utility will always be strictly less than $r - \mathsf{cost}(1)$, so the rational contractor will always run the original algorithm. □

Using a hash function with output length 160 bits (e.g., SHA-1), the boss can easily set $f, r, c$ appropriately so that every rational contractor will use the original algorithm. For the rest of the chapter, we can then assume $p \cong c$.

## 4.5 When to Check an Answer

In Section 4.3, we analyzed how to set the fine-to-reward ratio $f/r$ in terms of $p$, the probability that a contractor will be caught; e.g., by setting $f/r = (1-p)/p$ the boss can provide incentives to rational contractors. In this section, we will examine different strategies the boss can use to actually catch the contractors. We will analyze $c = \Pr[check|incorrect]$, the probability that the boss or other contractors will check

the answer of a contractor, conditioned on that contractor returning an incorrect answer.

### 4.5.1 Double Checking

A simple strategy is for the boss to randomly double-check an answers it gets with probability $t$. Here, the boss cannot know whether a job is incorrect until it has checked it, so $c = t$. Setting a low value of $t$ allows the boss to reduce the amount of work needed for double-checking—but since $c$ is inversely proportional to $f/r$, a high $f/r$ may present an impractical barrier for contractors seeking jobs.

### 4.5.2 Hiring Multiple Contractors

The boss can try to minimize the amount of checking he has to do by farming out the same job to multiple contractors. The boss then double-checks a submitted result only if the contractors disagree.

The problem is that if all contractors output the same false answer, the boss will never catch them. In fact, the contractors find themselves in a situation similar to the the iterated prisoner's dilemma. The best strategy for all the contractors is to employ a tit-for-tat mechanism: they should cheat until another contractor performs the computation honestly [131].

We begin our analysis by assuming that a fraction $h$ of the contractors will always perform the computation honestly: we call these contractors *diligent*. Later, we will show how to do away with this assumption. Suppose the boss chooses $m$ contractors at random and assigns them the same job. We can describe $c$ as the probability a contractor will be caught by other contractors if he submits an incorrect answer.

**Theorem 4.5.1.** *Suppose the boss farms out a job to $m$ contractors, each of which are honest with probability $h$, then the probability that a cheating contractor will be*

*caught is* $c = 1 - (1 - h)^{m-1}$.

*Proof.* A contractor who submits an incorrect result will be caught only if there exists a diligent contractor in the group working on the same job. The probability that all of the other $m - 1$ contractors are non-diligent is $L = (1 - h)^{m-1}$. Thus the probability that at least one of the other $m - 1$ contractors is diligent is $c = 1 - L$. □

**Corollary 4.5.1.1.** *Suppose the boss farms out a job to m contractors, which are honest with probability h, then by computing f/r using* $p \cong c = 1 - (1 - h)^{m-1}$ *in section 4.3, the boss can guarantee that all rational contractors will act honestly all the time.*

This strategy still requires the boss to perform work when the results submitted by contractors are in disagreement. In a system where all the contractors are rational, there should be no disagreement at all. But if malicious or colluding contractors are present, they may try to force the boss to double-check by returning an incorrect answer. We analyze this behavior in Section 4.6.

### 4.5.3 Hybrid Strategy

The boss can also pursue a hybrid strategy: he can farm out a job to multiple contractors *and* randomly double-check some of the answers. Thus even if all contractors collude to give the same wrong answer, the boss can still catch them.

**Theorem 4.5.2.** *Suppose the boss farms out a job to m contractors, which are honest with probability h. The boss also randomly double-checks the jobs with probability t when all the results agree. Then,* $c = 1 - (1 - t)(h^m + (1 - h)^m)$.

*Proof.* The boss definitely checks the answer if there is at least one diligent and one cheating contractor in the group. This has probability $1 - h^m - (1 - h)^m$. In any

other case (probability $h^m + (1-h)^m$), all answers will agree and the boss will check with probability $t$. Therefore, we get $c = (1 - h^m - (1-h)^m) + (h^m + (1-h)^m)t = 1 - (1-t)(h^m + (1-h)^m)$. □

### 4.5.4 Employing Bounties

Now let us discuss how to shed the assumption that there are diligent contractors. In the iterated prisoner's dilemma it is assumed that in each round, a contractor plays against the same group of other contractors. In our scenario, the boss will randomly choose a new group of contractors for each job. The contractors are really playing a single round of the prisoner's dilemma many times with a different group of contractors. Thus, if we set $f/r$ properly, the dominant strategy will be for the contractors to act honestly.

**Definition 4.5.1** (Nash Equilibrium)**.** *A Nash Equilibrium exists if all players choose a strategy, and no player can improve his utility by changing his strategy.*

The table below computes the expected utilities $u(1)$ and $u(q)$ for a contractor depending on whether the other players all chose to be diligent or lazy. As before, $q$ refers to the probability that a lazy contractor returns the correct answer. Please see Section 4.4 for how to use hashing to set $q$ arbitrarily close to 0.

| | |
|---|---|
| All Diligent | $u(1) = r - \mathsf{cost}(1)$<br>$u(q) = rq - f(1-q) - \mathsf{cost}(q)$ |
| All Lazy | $u(1) = r - \mathsf{cost}(1)$<br>$u(q) = r - \mathsf{cost}(q)$ |

There are two Nash equilibria: If all other players cheat, a rational player will also cheat. If at least one player is honest, a rational player must also be honest.

We can break the cheating equilibria by introducing a bounty. If the contractors

disagree on the output, the boss will check the computation and award $b$ to all contractors who output the correct answer. Now the expected utility for being diligent when everyone else chooses to be lazy is $u(1) = r - \mathsf{cost}(1) + b(1-q)$.

**Theorem 4.5.3.** *Suppose the boss asks two contractors to perform a job. Then the boss must set $f/r > 0$ and give a bounty of $b \geq r/(1-q)$ to honest contractors whenever they catch a cheating contractor.*

*Proof.* We have that $r \geq \mathsf{cost}(1) \geq \mathsf{cost}(q)$. First, if all other players are diligent then a contractor is better off also acting honestly as long as

$$0 \geq rq - f(1-q) - \mathsf{cost}(q) > rq - f(1-q) - r.$$

As a result, we get $f/r > -1$. Since it makes no sense to have a negative fine (paying contractors for wrong answers) and since a negative reward (taking away money for right answers) discourages participation, we set $f/r > 0$. Second, if even one player is lazy, then the contractor has an incentive to be diligent as long as $r - \mathsf{cost}(1) + b(1-q) \geq r - \mathsf{cost}(q)$. The boss needs to set

$$b \geq \frac{r}{1-q} \geq \frac{\mathsf{cost}(1) - \mathsf{cost}(q)}{1-q}.$$

□

## 4.6 Malicious Contractors

Malicious (or Byzantine) contractors attack the system: they want to reduce the accuracy of job results or increase the amount of double-checking the boss must do. They are irrational, or may pursue a utility function outside our model. Yet, to be able to stay in the system, they must keep at least a zero balance of utility (if they cannot afford the fine, they will not be hired by the boss). Malicious contractors

may also collude, through centralized control (as in the Sybil attack), via external communication, and even by sharing resources (the reward $r$).

### 4.6.1 Independent Malicious Contractors

Even a malicious contractor must maintain a certain minimum balance in his bank account. Otherwise, the boss will not ask him to perform jobs. Thus, a malicious contractor intent on submitting as many incorrect results as possible must also compute jobs correctly *some* fraction of the time.

**Definition 4.6.1.** *A malicious contractor will return the correct answer $x$ fraction of the time, and an incorrect answer $y$ fraction of the time; thus $x + y = 1$.*

We compute the utility of a single malicious contractor as

$$u(m) = xr + y(1-p)r - ypf,$$

where $x$ and $y$ are defined above and $p$ is the probability that the contractor will be caught. We want to know how large a value $y$ can the malicious contractor get away with while still maintaining a non-negative utility.

**Definition 4.6.2.** *Let $d$ be the deterrent factor, where the boss sets $f/r = d/p$. Observe that if $d = 1 - p$, this corresponds to our basic construction. Larger values of $d$ indicate that the boss has decided to deter maliciousness by increasing the $f/r$ ratio without decreasing the checks.*

**Theorem 4.6.1.** *The fraction of incorrect results $y$ that a malicious contractor can return to the boss is less than or equal to $1/(p + d)$.*

*Proof.* The malicious contractor needs to have a non-negative balance: $0 \leq u(m) = xr + y(1-p)r - ypf$. We substitute $f = rd/p$ and $x = 1 - y$ in the inequality to get $0 \leq (1-y)r + y(1-p)r - yrd$. We get rid of $r$, and solve to get $y \leq 1/(p+d)$. □

**Corollary 4.6.1.1.** *Suppose the boss hires only one contractor for each job and sets $f/r = d/p$. Then the probability that the boss accepts an incorrect result is $g(1-p)/(p+d)$, where $g$ is the fraction of malicious contractors in the system.*

Note that if the boss only randomly double-checks with some fixed probability, no malicious contractor can cause the boss to perform more work. However, in the setting where the same job is outsourced to multiple contractors and checked if there is disagreement, a malicious contractor can force the boss to perform a check by submitting an incorrect result, hence causing disagreement among the group.

### 4.6.2 Colluding Malicious Contractors

In our multiple-contractors scenario, the boss assigns each job to a randomly-selected group of size $m$, double-checking only when the contractors output different results, and fining those who submit an incorrect answer. We will examine two types of attacks by colluding contractors. In the first, the colluding contractors will try to trick the boss into accepting an incorrect answer. In the second, they will force the boss to perform extra checking by causing disagreements.

**Theorem 4.6.2.** *If the fraction of colluding contractors in the system is $g$, the probability that the boss accepts an incorrect result is at most $g^m$.*

*Proof.* The only way to trick the boss is if all the contractors in the group are colluders. For a group of size $m$, the probability that all group members are colluders is $g^m$. □

Colluding contractors may wish to force the boss to devote more resources to performing checks. The colluders can take advantage of the fact that if there is at least one colluder in the group, then one colluder can submit a wrong answer while the rest can submit the right answer and collect the reward. As a result, the overall

utility of the colluding group can be high enough to allow the group to continue participating in the system.

**Theorem 4.6.3.** *The amount of work the boss needs to perform due to a group of maliciously colluding contractors which make up a g fraction of all the contractors is at most $pgm/(p+d)$.*

The proof of Theorem 4.6.3 requires the following Lemma. We omit the proof of the Lemma, which follows from the Binomial Theorem and basic algebra.

**Lemma 4.6.1.** *Let $P(k,m) = \binom{m}{k} g^k (1-g)^{m-k}$ be the probability that there are exactly $k$ colluders in a group of size $m$. Furthermore, let $A = \sum_{k=1}^{m} P(k,m)$, be the probability that there is at least one colluder in the group. Then, $A = 1-(1-g)^m$. Finally, let $B = \sum_{k=1}^{m} P(k,m)k$. Then, $B = gm$.*

*Proof of Theorem 4.6.3.* We will first define the total utility of the colluding contractors. The contractors' strategy is simple. If there is at least one colluder in the group chosen by the bank, then one of the colluders will output a wrong answer with probability $y = 1-x$ while the rest output the correct answer. Then the total utility of the colluders for one job will be $xkr + y((k-1)r - f)$ for $k$ colluders (with probability $x = 1-y$, all colluders will get the reward by outputting the correct answer, and with probability $y$ only one of them will get fined while the rest will be rewarded). If we sum over the probability that the there are $k$ colluding contractors in a group of size $m$, we get the total expected utility of the colluders.

$$\begin{aligned} u(c) &= \sum_{k=1}^{m} P(k,m)[xkr + y((k-1)r - f)] \\ &= xrB + yrB - yrA - yArd/p \end{aligned}$$

if we do the substitutions for $A, B$ and $f$. The colluders want to maximize $y$ while keeping their total utility positive: $u(c) \geq 0$. Then, rearranging the equation above

gives us

$$y \leq \frac{B}{A(1+d/p)} = \frac{pgm}{(p+d)A}.$$

Next, we note that, a job will provide this group of colluders the ability to cheat in order to make the boss work more only if there is at least one colluder in that group. So, $A$ fraction of the jobs will enable the colluders to force the boss for a check. Therefore, by multiplying $y$ with $A$, we obtain the fraction of the time colluders can cause the boss to work, which is at most $pgm/(p+d)$. □

## 4.7 Evaluation

Throughout the chapter we have presented various methods by which the bank can tune the fine-to-reward ratio through setting other parameters. In this section, we show how the boss can select system parameters that balance performance trade-offs with protection against malicious contractors. We begin with the selection of the ratio $f/r$ and group size $m$, depending on the percentage of honest contractors $h$ in the system. The trade-off between high fine-to-reward ratio (which may present a barrier to entry for contractors) and large group size (which may unnecessarily waste effort due to redundant computation) is depicted in Figure 4.1. It can be seen from the figure that even a group size of 2 is enough to allow a reasonable fine-to-reward ratio, even in the presence of a very low percentage of honest contractors. Obviously, the higher the percentage of honest contractors, the smaller the group size required.

In the figure, we assumed that all the other contractors are rational. Assuming that the boss's view of the percentage of honest contractors is not higher than that of the contractors', the fine-to-reward ratios shown on the figure will provide incentives for rational contractors to always behave honestly. Next, we analyze the effect of irrational malicious and colluding contractors on the system when we set the fine-to-reward ratio so as to incentivize rational contractors.

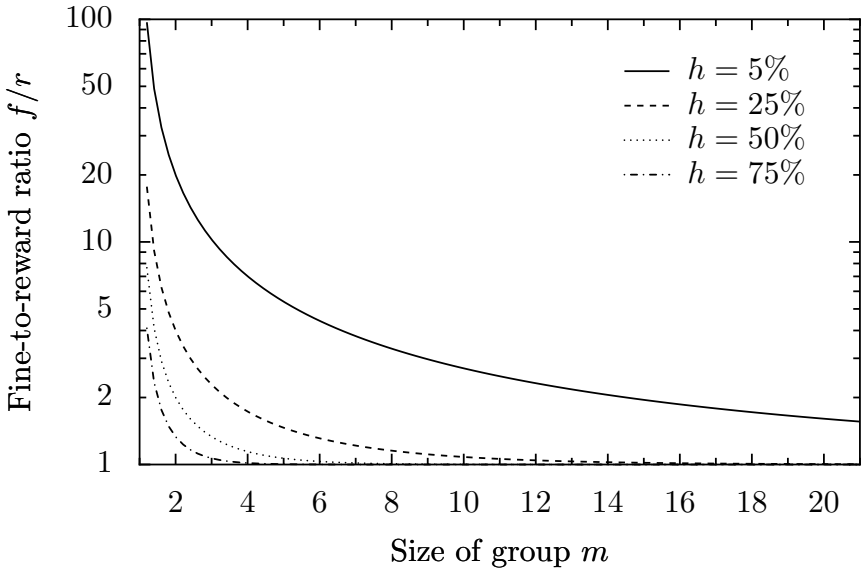


Figure 4.1: Example parameter settings for $f/r$ and $m$ that provide valid incentives assuming a fraction $h$ of honest users. (Theorem 4.5.1)

Figure 4.2 shows the percentage of bogus results the irrational malicious and colluding contractors, who are not incentivized by our scheme, can cause the boss to accept. The boss can adjust the deterrent factor to deter malicious contractors by increasing the fine-to-reward ratio without decreasing the probability of catching them. The figure shows the case when the boss employs 2 contractors per job, and thus represents a worst-case multiple-contractor scenario. When more contractors are employed, the fraction of bogus results accepted by the boss will be lower, since the colluders need to control the entire group in order to cheat the boss.

Next, in Figure 4.3, we see the fraction of extra double-checking work the colluding contractors can force the boss to perform. The figure again uses a group size of 2. Increasing the group size makes things worse in this case: the reason is that the colluders can make the boss work only if there is at least one of them in the group the boss selects. When the group size increases, the chance of that happening increases. An interesting point to make is that if the boss's probability of catching the colluders increases, then he obviously needs to perform more work. Luckily, the fraction of

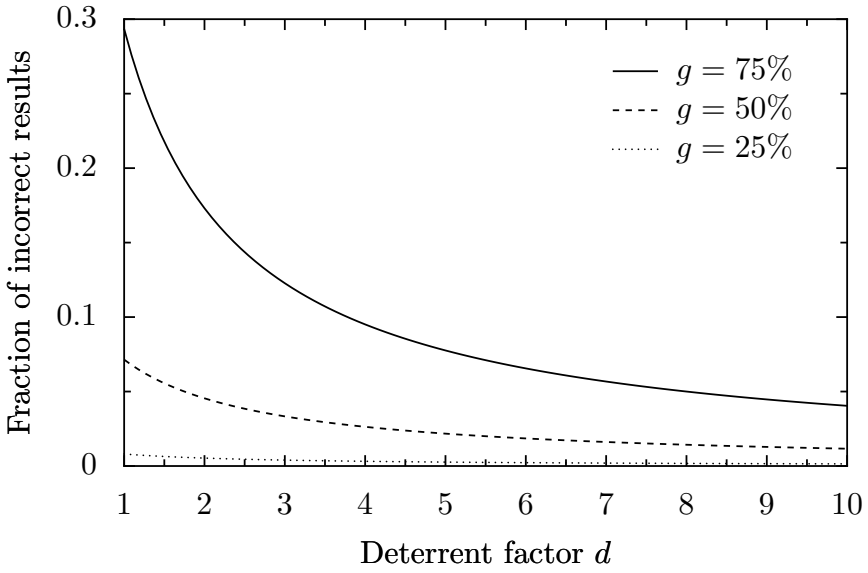


Figure 4.2: The maximum fraction of incorrect results that the boss will accept due to a fraction $g$ of malicious contractors, for different settings of the deterrent factor $d$. (Corollary 4.6.1.1)

bogus results that will be accepted is bounded as in Figure 4.2.

Note that the number of honest contractors do not affect the performance of the system, in terms of both the percentage of bogus results and extra work for the boss, once the fine-to-reward ratio is set. This is the case because once the ratio is set according to the fraction of honest contractors, then every rational contractor will have incentive to perform the job correctly. If the system is dynamic and the percentage of honest contractors decrease, the fine-to-reward ratio needs to be readjusted.

Our system can deter maliciousness without very high fine-to-reward ratios or large group sizes even if there are very few honest contractors in the system. In most cases (except when there is an extremely low number of honest users, i.e., $h = 0.05$, or an extremely high number of malicious users, i.e., $g = 0.75$), a deterrent factor of $d = 5$ and a group size of $m = 2$ is enough to result in a practical fine-to-reward ratio ($f/r \leq 25$), while guaranteeing at most 10% of bogus results and about 15% more work in very unrealistic highly adversarial scenarios (75% malicious), or almost no

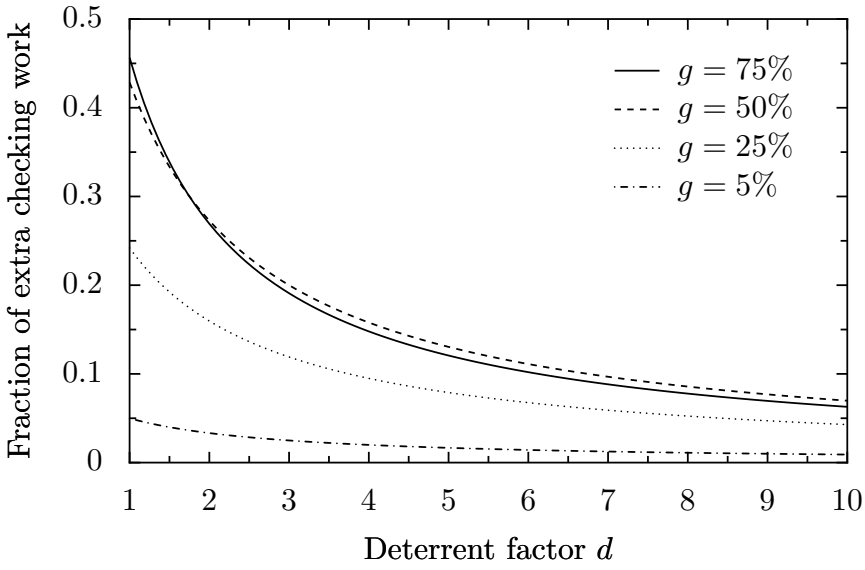


Figure 4.3: The maximum amount of extra double-checking work that a group of malicious colluders controlling a fraction $g$ of all contractors can force the boss to perform, for different settings of the deterrent factor $d$. (Theorem 4.6.3)

bogus results and about 5% more work in more realistic scenarios (5% malicious).

## 4.8 Conclusion and Future Work

We have presented different techniques that can be applied for incentivizing outsourced computation, through redundant computation by the boss or other contractors. The hashing technique prevents the use of other algorithms than prescribed by the boss. Then, we showed how to set the fine-to-reward ratio in presence of irrational honest users (Section 4.5.2), or when the contractors cannot collude in large scale in the long run (Section 4.5.4). Finally, we have shown that using our techniques, a reasonable fine-to-reward ratio can incentivize all rational users to behave honestly, and limit the damage by irrational malicious contractors.

All of these techniques aim to decrease the amount of work our centralized boss needs to perform. We assumed that this boss can afford to pay all rewards and is

capable of fining the contractors: another possibility is that multiple bosses might be in agreement with an entity of such power. Then, before a job is outsourced, each contractor might provide an escrow of the fine, so that the boss can claim it if cheating is detected. Additionally, bosses might provide different incentive structures $f/r$ to different peers, offering higher prices to those willing to accept larger fines. In such a decentralized environment, designing a distributed, budget-balanced mechanism provides a direction for our future work.

The currency used by our system could also serve other purposes, e.g., to buy data as in the currency-based P2P system of Belenkiy *et al.* [16]. In future work, we will study the effects of outsourcing e-coin verification on this system's virtual economy. Finally, analyzing possible damage done by the malicious contractors in the bounty framework is left as future work.

# Chapter 5

# Storing in the Cloud

## 5.1 Introduction

In cloud storage systems, where a client outsources the storage of her data to a server, the server (or peer) that stores the client's data is not necessarily trusted. Therefore, users would like to check if their data has been tampered with or deleted. However, outsourcing storage of very large files (or whole file systems) to remote servers presents an additional constraint: the client should not download all stored data in order to validate it since this may be prohibitive in terms of bandwidth and time, especially if the client performs this check frequently (therefore *authenticated data structure* solutions [144] cannot be directly applied in this scenario).

Ateniese et al. [9] have formalized this using a model called *provable data possession* (PDP). In this model, data (often represented as a file $F$) is preprocessed by the client, and metadata used for verification purposes is produced. The file is then sent to an untrusted server for storage, and the client may delete the local copy of the file. The client keeps some (possibly secret) information to check server's responses later. The server proves the data has not been tampered with by responding to challenges sent by the client. The authors present several variations of their scheme under

different cryptographic assumptions. These schemes provide probabilistic guarantees of possession, where the client checks a random subset of stored blocks with each challenge.

| **Scheme** | PDP [9] | Scalable PDP [10] | DPDP I | DPDP II |
|---|---|---|---|---|
| **Server computation** | $O(1)$ | $O(1)$ | $O(\log n)$ | $O(n^{\epsilon} \log n)$ |
| **Client computation** | $O(1)$ | $O(1)$ | $O(\log n)$ | $O(\log n)$ |
| **Communication** | $O(1)$ | $O(1)$ | $O(\log n)$ | $O(\log n)$ |
| **Model** | Random Oracle | Random Oracle | Standard | Standard |
| **Append blocks** | ✓ | ✓* | ✓ | ✓ |
| **Modify blocks** | | ✓* | ✓ | ✓ |
| **Insert blocks** | | | ✓ | ✓ |
| **Delete blocks** | | ✓* | ✓ | ✓ |
| **Prob. of detection** | $1-(1-f)^C$ | $1-(1-f)^C$ | $1-(1-f)^C$ | $1-(1-f)^{\Omega(\log n)}$ |

Table 5.1: Comparison of PDP schemes: original PDP scheme [9]; Scalable PDP [10]; our scheme based on authenticated skip lists (DPDP I); and our scheme based on RSA trees (DPDP II). A star (*) indicates that a certain operation can be performed only a limited (pre-determined) number of times. We denote with $n$ the number of the blocks of the file, with $f$ the fraction of the corrupted blocks, and with $C$ being the number of challenged blocks (typically a constant independent of $n$). In all constructions, the storage space is $O(1)$ at the client and $O(n)$ at the server.

However, PDP and related schemes [9, 65, 90, 139] apply only to the case of static, archival storage, i.e., a file that is outsourced and never changes (simultaneously with our work, Ateniese et al. [10] present a scheme with somewhat limited dynamism, which is discussed in detail in the related work section). While the static model fits some application scenarios (e.g., libraries and scientific datasets), it is crucial to consider the dynamic case, where the client updates the outsourced data—by inserting, modifying, or deleting stored blocks or files—while maintaining data possession guarantees. Such a dynamic PDP scheme is essential in practical cloud computing systems for file storage [92, 101], database services [105], and peer-to-peer storage [94, 115].

In this chapter, we provide a definitional framework and efficient constructions for *dynamic provable data possession* (DPDP), which extends the PDP model to

support provable *updates* on the stored data. Given a file $F$ consisting of $n$ blocks, we define an update as either insertion of a new block (anywhere in the file, not only append), or modification of an existing block, or deletion of any block. Therefore our update operation describes the most general form of modifications a client may wish to perform on a file.

Our DPDP solution is based on a new variant of authenticated dictionaries, where we use *rank* information to organize dictionary entries. Thus we are able to support efficient authenticated operations on files at the block level, such as authenticated insert and delete. We prove the security of our constructions using standard assumptions.

We also show how to extend our construction to support data possession guarantees of a hierarchical file system as well as file data itself. To the best of our knowledge, this is the first construction of a provable storage system that enables efficient proofs of a whole file system, enabling verification at different levels for different users (e.g., every user can verify her own home directory) and at the same time not having to download the whole data (as opposed to [87]). Our scheme yields a provable outsourced versioning system (e.g., CVS), which is evaluated in Section 5.8 by using traces of CVS repositories of three well-known projects.

### 5.1.1 Contributions

The main contributions of this work are summarized as follows:

1. We introduce a formal framework for *dynamic provable data possession* (DPDP);

2. We provide the first efficient *fully dynamic* PDP solution;

3. We present a rank-based authenticated dictionary built over a skip list. This

construction yields a DPDP scheme with logarithmic computation and communication and the same detection probability as the original PDP scheme (DPDP I in Table 5.1);

4. We give an alternative construction (Section 5.6) of a rank-based authenticated dictionary using an RSA tree [126]. This construction results in a DPDP scheme with improved detection probability but higher server computation (see DPDP II in Table 5.1);

5. We present practical applications of our DPDP constructions to outsourced file systems and versioning systems (e.g., CVS, with variable block size support); **(6)** We perform an experimental evaluation of our skip list-based scheme.

Now, we outline the performance of our schemes. Denote with $n$ the number of blocks. The *server computation*, i.e., the time taken by the server to process an update or to compute a proof for a block, is $O(\log n)$ for DPDP I and $O(n^{\epsilon} \log n)$ for DPDP II; the *client computation*, i.e., the time taken by the client to verify a proof returned by the server, is $O(\log n)$ for both schemes; the *communication complexity*, i.e., the size of the proof returned by the server to the client, is $O(\log n)$ for both schemes; the *client storage*, i.e., the size of the meta-data stored locally by the client, is $O(1)$ for both schemes; finally, the *probability of detection*, i.e., the probability of detecting server misbehavior, is $1 - (1 - f)^C$ for DPDP I and $1 - (1 - f)^{\Omega(\log n)}$ for DPDP II, for fixed logarithmic communication complexity, where $f$ is the ratio of corrupted blocks and $C$ is a constant, i.e., independent of $n$.

We observe that for DPDP I, we could use a dynamic Merkle tree (e.g., [99, 116]) instead of a skip list to achieve the same asymptotic performance. We have chosen the skip list due to its simple implementation and the fact that algorithms for updates in the two-party model (where clients can access only a logarithmic-sized portion of the

data structure) have been previously studied in detail for authenticated skip lists [125] but not for Merkle trees.

### 5.1.2 Related Work

The PDP scheme by Ateniese et al. [9] provides an optimal protocol for the *static* case that achieves $O(1)$ costs for all the complexity measures listed above. They review previous work on protocols fitting their model, but find these approaches lacking: either they require expensive server computation or communication over the entire file [79, 120], linear storage for the client [137], or do not provide security guarantees for data possession [136]. Note that using [9] in a dynamic scenario is insecure due to replay attacks. As observed in [67], in order to avoid replay attacks, an authenticated tree structure that incurs logarithmic costs must be employed and thus constant costs are not feasible in a dynamic scenario.

Juels and Kaliski present proofs of retrievability (PORs) [90], focusing on static archival storage of large files. Their scheme's effectiveness rests largely on preprocessing steps the client conducts before sending a file $F$ to the server: "sentinel" blocks are randomly inserted to detect corruption, $F$ is encrypted to hide these sentinels, and error-correcting codes are used to recover from corruption. As expected, the error-correcting codes improve the error-resiliency of their system. Unfortunately, these operations prevent any efficient extension to support updates, beyond simply replacing $F$ with a new file $F'$. Furthermore, the number of queries a client can perform is limited, and fixed a priori. Shacham and Waters have an improved version of this protocol called Compact POR [139], but their solution is also static (see [65] for a summary of POR schemes and related trade-offs).

In our solution, we regard error-correcting codes or encryption as external to our system. If the user wants to have more error-resiliency, she can provide us with a file

that has error-correcting codes integrated (or an encrypted file if secrecy is desired). This provides our protocol with the same guarantees as the POR protocols. Since our construction does not modify the file and assumes no property on it, our system will work in perfect compliance.

Simultaneously with our work, Ateniese et al. have developed a dynamic PDP solution called Scalable PDP [10]. Their idea is to come up with all future challenges during setup and store pre-computed answers as metadata (at the client, or at the server in an authenticated and encrypted manner). Because of this approach, the number of updates and challenges a client can perform is limited and fixed a priori. Similarly, this also means that the probabilistic detection guarantee is fixed during setup, whereas in our scheme the client can decide on the probabilistic detection guarantee for each challenge independently and on the fly.

Also, one cannot perform block insertions anywhere (only append-type insertions are possible). Furthermore, each update requires re-creating all the remaining challenges, which is problematic for large files. Under these limitations (otherwise the lower bound of [67] would be violated), they provide a protocol with optimal asymptotic complexity $O(1)$ in all complexity measures giving the same probabilistic guarantees as our scheme. Lastly, their work is in the random oracle model whereas our scheme is provably secure in the standard model (see Table 5.1 for full comparison).

As an implementation problem, their dynamic scheme does not use the block numbers the file system uses, and hence they need a mapping between the block numbers in the scheme and actual block numbers in the file system if one wants to implement a file system or a versioning system, as we do, atop their work. This is due to the fact that during deletion, they keep all the block IDs and treat a block with a deleted ID as nonexistent. In our scheme using rank-based authenticated skip lists, this is never a problem, since block IDs are no longer used as search keys in the

skip list.

Finally, our work is closely related to memory checking, for which lower bounds are presented in [67, 117]. Specifically, in [67] it is proved that all non-adaptive and deterministic checkers have read and write query complexity summing up to $\Omega(\log n / \log \log n)$ (necessary for sublinear client storage), justifying the $O(\log n)$ cost in our scheme. Note that for schemes based on cryptographic hashing, an $\Omega(\log n)$ lower bound on the proof size has been shown [51, 145]. Related bounds for other primitives have been shown by Blum et al. [28]. Nevertheless, our scheme achieves optimal client storage by requiring $O(1)$ space.

## 5.2 Model

We build on the PDP definitions from [9]. We begin by introducing a general DPDP scheme and then show how the original PDP model is consistent with this definition.

**Definition 5.2.1** (DPDP Scheme)**.** *In a DPDP scheme, there are two parties. The* ***client*** *wants to off-load her files to the untrusted* ***server****. A complete definition of a DPDP scheme should describe the following (possibly randomized) efficient procedures:*

- $\mathsf{KeyGen}(1^k) \rightarrow \{\mathsf{sk}, \mathsf{pk}\}$ *is a probabilistic algorithm run by the* ***client****. It takes as input a security parameter, and outputs a secret key* $\mathsf{sk}$ *and a public key* $\mathsf{pk}$*. The client stores the secret and public 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 an algorithm run by the* ***client*** *to prepare (a part of) the file for untrusted storage. As input, it takes secret and public keys, (a part of) the file $F$ with the definition* $\mathsf{info}$ *of the update to be performed (e.g., full re-write, modify block $i$, delete block $i$, add a block after block $i$, etc.), and the previous metadata $M_c$. The output is an "encoded" version*

*of (a part of) the file* $\mathsf{e}(F)$ *(e.g., by adding randomness, adding sentinels, encrypting for confidentiality, etc.), along with the information* $\mathsf{e}(\mathsf{info})$ *about the update (changed to fit the encoded version), and the new metadata* $\mathsf{e}(M)$*. The client sends* $\mathsf{e}(F), \mathsf{e}(\mathsf{info}), \mathsf{e}(M)$ *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 an algorithm run by the* ***server*** *in response to an update request from the client. The input contains the public key* $\mathsf{pk}$*, the previous version of the file* $F_{i-1}$*, the metadata* $M_{i-1}$ *and the client-provided values* $\mathsf{e}(F), \mathsf{e}(\mathsf{info}), \mathsf{e}(M)$*. Note that the values* $\mathsf{e}(F), \mathsf{e}(\mathsf{info}), \mathsf{e}(M)$ *are the values produced by* PrepareUpdate*. The output is the new version of the file* $F_i$ *and the metadata* $M_i$*, along with the metadata to be sent to the client* $M'_c$ *and its proof* $P_{M'_c}$*. The server sends* $M'_c, P_{M'_c}$ *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 server's behavior during the update. It takes all inputs of the* PrepareUpdate *algorithm,*[1] *plus the* $M'_c, P_{M'_c}$ *sent by the server. It outputs acceptance (*$F$ *can be deleted in that case) or rejection signals;*

- $\mathsf{Challenge}(\mathsf{sk}, \mathsf{pk}, M_c) \rightarrow \{c\}$ *is a probabilistic procedure run by the* ***client*** *to create a challenge for the server. It takes the secret and public keys, along with the latest client metadata* $M_c$ *as input, and outputs a challenge* $c$ *that is then sent to the server;*

- $\mathsf{Prove}(\mathsf{pk}, F_i, M_i, c) \rightarrow \{P\}$ *is the procedure run by the* ***server*** *upon receipt of a challenge from the client. It takes as input the public key, the latest version of the file and the metadata, and the challenge* $c$*. It outputs a proof* $P$ *that is sent to the client;*

[1] However, in our model $F$ denotes part of some encoded version of the file and not part of the actual data (though for generality purposes we do not make it explicit).

- $\mathsf{Verify}(\mathsf{sk}, \mathsf{pk}, M_c, c, P) \rightarrow \{\mathsf{accept}, \mathsf{reject}\}$ *is the procedure run by the* ***client*** *upon receipt of the proof $P$ from the server. It takes as input the secret and public keys, the client metadata $M_c$, the challenge $c$, and the proof $P$ sent by the server. An output of accept ideally means that the server still has the file intact. We will define the security requirements of a DPDP scheme later.*

We assume there is a hidden input and output *clientstate* in all functions run by the client, and *serverstate* in all functions run by the server. Some inputs and outputs may be empty in some schemes. For example, the PDP scheme of [9] does not store any metadata at the client side. Also $\mathsf{sk}, \mathsf{pk}$ can be used for storing multiple files, possibly on different servers. All these functions can be assumed to take some public parameters as an extra input if operating in the public parameters model, although our construction does not require such modifications. Apart from $\{\mathsf{accept}, \mathsf{reject}\}$, algorithm $\mathsf{VerifyUpdate}$ can also output a new client metadata $M_c$. In most scenarios, this new metadata will be set as $M_c = M_c'$.

Retrieval of a (part of a) file is similar to the challenge-response protocol above, composed of $\mathsf{Challenge}, \mathsf{Prove}, \mathsf{Verify}$ algorithms, except that along with the proof, the server also sends the requested (part of the) file, and the verification algorithm must use this (part of the) file in the verification process. These algorithms are outlined below:

- $\mathsf{Retrieve}(\mathsf{sk}, \mathsf{pk}, M_c, \mathsf{info}) \rightarrow \{\mathsf{e}(\mathsf{info}), c\}$ is a probabilistic procedure run by the **client** to request (a part of) the file from the server, along with the associated proof. It takes the secret and public keys, along with the latest client metadata $M_c$, and the information about what to request as input, and outputs the encoded information and an associated challenge $c$ that is then sent to the server;

- $\mathsf{Send}(\mathsf{pk}, F_i, M_i, \mathsf{e}(\mathsf{info}), c) \rightarrow \{F'_{\mathsf{e}(\mathsf{info})}, P\}$ is the procedure run by the **server** upon receipt of a retrieval request from the client. It takes as input the public key, the

latest version of the file and the metadata, the information about requested (parts of the) file, and the challenge $c$ on the requested blocks. It outputs the requested (part of the) file along with a proof $P$ that is sent to the client;

- $\mathsf{Obtain}(\mathsf{sk}, \mathsf{pk}, M_c, \mathsf{info}, c, F'_{\mathsf{e(info)}}, P) \rightarrow \{F_{\mathsf{info}}\}$ is the procedure run by the **client** upon receipt of the (part of the) file and its associated proof $P$ from the server. It takes as input the secret and public keys, the client metadata $M_c$, the request information $\mathsf{info}$, the challenge $c$, the (part of the) file $F'_{\mathsf{e(info)}}$, and the proof $P$ sent by the server. The client checks if the proof corresponds to the (part of the) file that is received, and outputs the (decoded) (part of the) file. If the proof does not verify using the blocks received and the associated challenge, then the client outputs null.

We also note that a PDP scheme is consistent with the DPDP scheme definition, with algorithms $\mathsf{PrepareUpdate}$, $\mathsf{PerformUpdate}$ and $\mathsf{VerifyUpdate}$ specifying an update that is a full re-write (or append).

**Definition 5.2.2** (PDP Scheme)**.** *A PDP scheme is consistent with the DPDP scheme definition, with algorithms* $\mathsf{PrepareUpdate}$*,* $\mathsf{PerformUpdate}$ *and* $\mathsf{VerifyUpdate}$ *specifying an update that is a full re-write (or append).*

As stated above, PDP is a restricted case of DPDP. The PDP scheme of [9] has the same algorithm definition for key generation, defines a restricted version of $\mathsf{PrepareUpdate}$ that can create the metadata for only one block at a time, and defines $\mathsf{Prove}$ and $\mathsf{Verify}$ algorithms similar to our definition. It lacks an explicit definition of $\mathsf{Challenge}$ (though one is very easy to infer). $\mathsf{PerformUpdate}$ consists of performing a full re-write or an append (so that *replay* attacks can be avoided), and $\mathsf{VerifyUpdate}$ is used accordingly, i.e., it always accepts in case of a full re-write or it is run as in DPDP in case of an append. It is clear that our definition allows a broad range of DPDP (and PDP) schemes.

We now define the security of a DPDP scheme, inspired by the security definitions of [9, 65]. Note that the restriction to the PDP scheme gives a security definition for PDP schemes compatible with the ones in [9, 10].

**Definition 5.2.3** (Security of DPDP)**.** *We say that a DPDP scheme is secure if for any probabilistic polynomial time (PPT) adversary who can win the following data possession game with non-negligible probability, there exists a PPT extractor algorithm* $\mathsf{Ext}$ *that can extract (at least) the challenged parts of the file with high probability by resetting and challenging the adversary polynomially many times. We think about the extractor as composed of the following two parts: Assume the challenger picks random blocks* $\mathsf{info}$ *that he wants to challenge. Then,* $\mathsf{ExtChal}(\mathsf{pk}, \mathsf{sk}, M_c, \mathsf{info}) \to \{c_{\mathsf{info}}\}$ *produces random challenges for those blocks, and* $\mathsf{ExtBlock}(\mathsf{pk}, \mathsf{sk}, M_c, \mathsf{info}, c^i_{\mathsf{info}}, P^i_{\mathsf{info}}) \to \{F_{\mathsf{info}}\}$ *extracts those blocks given polynomially-many such challenges and verifying proofs.*

Data Possession Game*: Played between the challenger who plays the role of the client and the adversary who acts as a server.*

1. Keygen*: The challenger runs* $\mathsf{KeyGen}(1^k) \to \{\mathsf{sk}, \mathsf{pk}\}$ *and sends the public key* $\mathsf{pk}$ *to the adversary;*

2. ACF Queries*: The adversary is very powerful. The adversary can mount adaptive chosen file (ACF) queries as follows. The adversary specifies a message* $F$ *and the related information* $\mathsf{info}$ *specifying what kind of update to perform (see Definition 5.2.1) and sends these to the challenger. The challenger runs* $\mathsf{PrepareUpdate}$ *on these inputs and sends the resulting* $\mathsf{e}(F), \mathsf{e}(\mathsf{info}), \mathsf{e}(M)$ *to the adversary. Then the adversary replies with* $M'_c, P_{M'_c}$ *which are verified by the challenger using the algorithm* $\mathsf{VerifyUpdate}$*. The result of the verification is told to the adversary. The adversary can further request challenges, return*

*proofs, and be told about the verification results. The adversary can repeat the interaction defined above polynomially-many times;*

3. SETUP*: Finally, the adversary decides on messages* $F_i^*$ *and related information* $\mathsf{info}_i^*$ *for all* $i = 1, \ldots, R$ *of adversary's choice of polynomially-large (in the security parameter* $k$*)* $R \geq 1$*. The ACF interaction is performed again, with the first* $\mathsf{info}_1^*$ *specifying a full re-write (this corresponds to the first time the client sends a file to the server). The challenger updates his local metadata only for the verifying updates (hence, non-verifying updates are considered not to have taken place—data has not changed);*

4. CHALLENGE*: Call the final version of the file* $F$*, created according to the verifying updates the adversary requested in the previous step. The challenger holds the latest metadata* $M_c$ *sent by the adversary and verified as accepting. Now the challenger picks random blocks* $\mathsf{info}$ *and runs the* $\mathsf{ExtChal}$ *algorithm to create a challenge that is sent to the adversary. The adversary returns a proof* $P$*. If the output of* $\mathsf{Verify}(\mathsf{sk}, \mathsf{pk}, M_c, c, P)$ *is accept, then the adversary wins. The challenger has the ability to reset the adversary to the beginning of the challenge phase and repeat this step polynomially-many times for the purpose of extraction. At the end, the challenger provides the* $\mathsf{ExtBlock}$ *algorithm a transcript of his interaction with the adversary. Overall, if the adversary wins the data possession game with non-negligible probability, there must exist an extractor that outputs the challenged blocks with high probability.*

**Definition 5.2.4** (Alternative Security Definition for DPDP)**.** *A DPDP scheme is secure if for any PPT* $f$*-adversary who can win the data possession game with non-negligible probability on* $f$*-fraction of blocks, there exists a PPT* $f$*-extractor algorithm* $f-\mathsf{Ext}$ *that can extract* $f$*-fraction of blocks of the file with high probability by resetting and challenging the adversary polynomially many times.*

**Theorem 5.2.1.** *Definitions 5.2.3 and 5.2.4 are equivalent.*

*Proof.* The $f - \mathsf{Ext}$ employs $\mathsf{Ext}$ on subsets of all $f$-fraction of the blocks each time, until all those blocks are extracted. If the $f$-adversary succeeds with non-negligible probability on those $f$-fraction of the blocks, then $\mathsf{Ext}$ will succeed in extracting subsets of these. For the other direction, as long as the number of challenged blocks is less than or equal to $f * n$, then $\mathsf{Ext}$ can employ $f - \mathsf{Ext}$ for the purposes of extraction. □

**Remark 1.** $0 \leq f \leq 1$.

**Remark 2.** *If $f < 1$ then the extractor cannot extract the whole file. In this case, the DPDP scheme should catch the adversary with some probability. This "probability of detection" will be discussed later.*

Note that our definition coincides with extractor definitions in *proofs of knowledge*. For an adversary that answers a non-negligible fraction of the challenges, a polynomial-time extractor must exist. Furthermore, this definition can be applied to the POR case [65, 90, 139], in which by repeating the challenge-response process, the extractor can extract the whole file with the help of error-correcting codes. The probability of catching a cheating server is analyzed in Section 5.5.

Finally, if a DPDP scheme is to be truly publicly verifiable, the $\mathsf{Verify}$ algorithm should not make use of the secret key. Since that is the case for our construction (see Section 5.4), we can derive a public verifiability protocol usable for official arbitration purposes (see Chapter 6).

## 5.3 Rank-based Authenticated Skip Lists

In order to implement our first DPDP construction, we use a modified authenticated skip list data structure [86]. This new data structure, which we call a *rank-based*

*authenticated skip list*, is based on authenticated skip lists but indexes data in a different way. Note that we could have based the construction on any authenticated search data structure, e.g., Merkle tree [109] instead. This would perfectly work for the static case. But in the dynamic case, we would need an authenticated red-black tree, and unfortunately no algorithms have been previously presented for rebalancing a Merkle tree while efficiently maintaining and updating authentication information (except for the three-party model, e.g., [99]). Yet, such algorithms have been extensively studied for the case of the authenticated skip list data structure [125]. Before presenting the new data structure, we briefly introduce authenticated skip lists.

The authenticated skip list is a skip list [129] (see Figure 5.1) with the difference that every node $v$ above the bottom level (which has two pointers, namely $\mathsf{rgt}(v)$ and $\mathsf{dwn}(v)$) also stores a label $f(v)$ that is a cryptographic hash and is computed using some collision-resistant hash function $h$ (e.g., SHA-1 in practice) as a function of $f(\mathsf{rgt}(v))$ and $f(\mathsf{dwn}(v))$. Using this data structure, one can answer queries like "does 21 belong to the set represented with this skip list?" and also provide a proof that the given answer is correct. To be able to verify the proofs to these answers, the client must always hold the label $f(s)$ of the top leftmost node of the skip list (node $w_7$ in Figure 5.1). We call $f(s)$ the *basis* (or *root*), and it corresponds to the client's metadata in our DPDP construction ($M_c = f(s)$). In our construction, the leaves of the skip list represent the blocks of the file. When the client asks for a block, the server needs to send that block, along with a proof that the block is intact.

### 5.3.1 Rank-based Queries

We can use an authenticated skip list to check the integrity of the file blocks. However, this data structure does not support efficient verification of the indices of the blocks, which are used as query and update parameters in our DPDP scenario. The updates

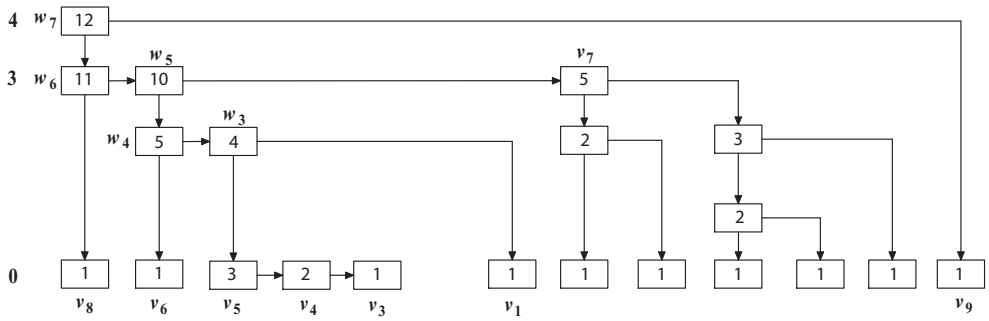


Figure 5.1: Example of rank-based skip list.

we want to support in our DPDP scenario are insertions of a new block after the $i$-th block and deletion or modification of the $i$-th block (there is no search key in our case, in contrast to [86], which basically implements an authenticated dictionary). If we use indices of blocks as search keys in an authenticated dictionary, we have the following problem. Suppose we have a file consisting of 100 blocks $m_1, m_2, \ldots, m_{100}$ and we want to insert a block after the 40-th block. This means that the indices of all the blocks $m_{41}, m_{42}, \ldots, m_{100}$ should be incremented, and therefore an update becomes extremely inefficient. To overcome this difficulty, we define a new hashing scheme that takes into account rank information.

### 5.3.2 Authenticating Ranks

Let $F$ be a file consisting of $n$ blocks $m_1, m_2, \ldots, m_n$. We store at the $i$-th bottom-level node of the skip list a representation $\mathcal{T}(m_i)$ of block $m_i$ (we will define $\mathcal{T}(m_i)$ later). Block $m_i$ will be stored elsewhere by the untrusted server. Each node $v$ of the skip list stores the number of nodes at the bottom level that can be reached from $v$. We call this value the *rank* of $v$ and denote it with $r(v)$. In Figure 5.1, we show the ranks of the nodes of a skip list. An insertion, deletion, or modification of a file block affects only the nodes of the skip list along a search path. We can recompute bottom-up the ranks of the affected nodes in constant time per node.

The top leftmost node of a skip list will be referred to as the *start node*. For example, $w_7$ is the start node of the skip list in Figure 5.1. For a node $v$, denote with $\mathsf{low}(v)$ and $\mathsf{high}(v)$ the indices of the leftmost and rightmost nodes at the bottom level reachable from $v$, respectively. Clearly, for the start node $s$ of the skip list, we have $r(s) = n$, $\mathsf{low}(s) = 1$ and $\mathsf{high}(s) = n$ be the nodes that can be reached from $v$ by following the right or the down pointer respectively. Using the ranks stored at the nodes, we can reach the $i$-th node of the bottom level by traversing a path that begins at the start node, as follows. For the current node $v$, assume we know $\mathsf{low}(v)$ and $\mathsf{high}(v)$. Let $w = \mathsf{rgt}(v)$ and $z = \mathsf{dwn}(v)$. We set

$$\begin{aligned} \mathsf{high}(w) &= \mathsf{high}(v)\,, \\ \mathsf{low}(w) &= \mathsf{high}(v) - r(w) + 1\,, \\ \mathsf{high}(z) &= \mathsf{low}(v) + r(z) - 1\,, \\ \mathsf{low}(z) &= \mathsf{low}(v)\,. \end{aligned}$$

If $i \in [\mathsf{low}(w), \mathsf{high}(w)]$, we follow the right pointer and set $v = w$, else we follow the down pointer and set $v = z$. We continue until we reach the $i$-th bottom node. Note that we do not have to store $\mathsf{high}$ and $\mathsf{low}$. We compute them on the fly using the ranks.

In order to authenticate skip lists with ranks, we extend the hashing scheme defined in [86]. We consider a skip list that stores data items at the bottom-level nodes. In our application, the node $v$ associated with the $i$-th block $m_i$ stores item $x(v) = \mathcal{T}(m_i)$. Let $l(v)$ be the level (height) of node $v$ in the skip list ($l(v) = 0$ for the nodes at the bottom level).

Let $||$ denote concatenation. We extend a hash function $h$ to support multiple

arguments by defining

$$h(x_1, \ldots, x_k) = h(h(x_1)|| \ldots ||h(x_k))\,.$$

We are now ready to define our new hashing scheme:

**Definition 5.3.1** (Hashing scheme with ranks)**.** *Given a collision resistant hash function $h$, the label $f(v)$ of a node $v$ of a rank-based authenticated skip list is defined as follows.*

***Case 0:*** $v = \mathsf{null}$

$$f(v) = 0\,;$$

***Case 1:*** $l(v) > 0$

$$f(v) = h(l(v), r(v), f(\mathsf{dwn}(v)), f(\mathsf{rgt}(v)))\,;$$

***Case 2:*** $l(v) = 0$

$$f(v) = h(l(v), r(v), x(v), f(\mathsf{rgt}(v)))\,.$$

### 5.3.3 Setup

Before inserting any block (i.e., if initially the skip list was empty), the basis, i.e., the label $f(s)$ of the top leftmost node $s$ of the skip list, can easily be computed by hashing the sentinel values of the skip list; —the file consists of only two "fictitious" blocks— block 0 and block $+\infty$.

| ***node*** $v$ | $v_3$ | $v_4$ | $v_5$ | $w_3$ | $w_4$ | $w_5$ | $w_6$ | $w_7$ |
|---|---|---|---|---|---|---|---|---|
| $l(v)$ | 0 | 0 | 0 | 2 | 2 | 3 | 3 | 4 |
| $q(v)$ | 0 | 1 | 1 | 1 | 1 | 5 | 1 | 1 |
| $g(v)$ | 0 | $\mathcal{T}(m_4)$ | $\mathcal{T}(m_5)$ | $f(v_1)$ | $f(v_6)$ | $f(v_7)$ | $f(v_8)$ | $f(v_9)$ |

Table 5.2: Proof for the $5^{th}$ block of the file $F$ stored in the skip list of Figure 5.1.

### 5.3.4 Queries

Suppose now the file $F$ and a skip list on the file have been stored at the untrusted server. The client wants to verify the integrity of block $i$ and therefore issues query $\mathsf{atRank}(i)$ to the server. The server executes Algorithm 5.3.1, described below, to compute $\mathcal{T}(i)$ and a proof for $\mathcal{T}(i)$ (for convenience we use $\mathcal{T}(i)$ to denote $\mathcal{T}(m_i)$).

Let $v_k, \ldots, v_1$ be the path from the start node, $v_k$, to the node associated with block $i$, $v_1$. The reverse path $v_1, \ldots, v_k$ is called the *verification path* of block $i$. For each node $v_j$, $j = 1, \ldots, k$, we define boolean $d(v_j)$ and values $q(v_j)$ and $g(v_j)$ as follows, where we conventionally set $r(\mathsf{null}) = 0$:

$$d(v_j) = \begin{cases} \mathsf{rgt} & j = 1 \text{ or } j > 1 \text{ and } v_{j-1} = \mathsf{rgt}(v_j) \\ \mathsf{dwn} & j > 1 \text{ and } v_{j-1} = \mathsf{dwn}(v_j) \end{cases},$$

$$q(v_j) = \begin{cases} r(\mathsf{rgt}(v_j)) & \text{if } j = 1 \\ 1 & \text{if } j > 1 \text{ and } l(v_j) = 0 \\ r(\mathsf{dwn}(v_j)) & \text{if } j > 1,\ l(v_j) > 0 \text{ and } d(v_j) = \mathsf{rgt} \\ r(\mathsf{rgt}(v_j)) & \text{if } j > 1,\ l(v_j) > 0 \text{ and } d(v_j) = \mathsf{dwn} \end{cases},$$

$$g(v_j) = \begin{cases} f(\mathsf{rgt}(v_j)) & \text{if } j = 1 \\ x(v_j) & \text{if } j > 1 \text{ and } l(v_j) = 0 \\ f(\mathsf{dwn}(v_j)) & \text{if } j > 1,\ l(v_j) > 0 \text{ and } d(v_j) = \mathsf{rgt} \\ f(\mathsf{rgt}(v_j)) & \text{if } j > 1,\ l(v_j) > 0 \text{ and } d(v_j) = \mathsf{dwn} \end{cases}.$$

The proof for block $i$ with data $\mathcal{T}(i)$ is the sequence $\Pi(i) = (A(v_1), \ldots, A(v_k))$ where $A(v) = (l(v), q(v), d(v), g(v))$. So the proof consists of tuples associated with the nodes of the verification path. Boolean $d(v)$ indicates whether the previous node is to the right or below $v$. For nodes above the bottom level, $q(v)$ and $g(v)$ are the

rank and label of the successor of $v$ that is not on the path. The proof $\Pi(5)$ for the skip list of Figure 5.1 is shown in Table 5.2. Due to the properties of skip lists, a proof has expected size $O(\log n)$ with high probability (whp).

**Algorithm 5.3.1**: $(\mathcal{T}, \Pi) = \mathsf{atRank}(i)$

1: Let $v_1, v_2, \ldots, v_k$ be the verification path for block $i$;
2: **return** representation $\mathcal{T}$ of block $i$ and proof $\Pi = (A(v_1), A(v_2), \ldots, A(v_k))$ for $\mathcal{T}$;

### 5.3.5 Verification

After receiving from the server the representation $\mathcal{T}$ of block $i$ and a proof $\Pi$ for it, the client executes Algorithm 3 to verify the proof using the stored metadata $M_c$.

**Algorithm 5.3.2**: $\{\mathsf{accept}, \mathsf{reject}\} = \mathsf{verify}(i, M_c, \mathcal{T}, \Pi)$

1: Let $\Pi = (A_1, \ldots, A_k)$, where $A_j = (l_j, q_j, d_j, g_j)$ for $j = 1, \ldots, k$;
2: $\lambda_0 = 0$; $\rho_0 = 1$; $\gamma_0 = \mathcal{T}$; $\xi_0 = 0$;
3: **for** $j = 1, \ldots, k$ **do**
4: $\quad \lambda_j = l_j$; $\rho_j = \rho_{j-1} + q_j$; $\delta_j = d_j$;
5: $\quad$ **if** $\delta_j = \mathsf{rgt}$ **then**
6: $\quad\quad \gamma_j = h(\lambda_j, \rho_j, \gamma_{j-1}, g_j)$;
7: $\quad\quad \xi_j = \xi_{j-1}$;
8: $\quad$ **else** $\{\delta_j = \mathsf{dwn}\}$
9: $\quad\quad \gamma_j = h(\lambda_j, \rho_j, g_j, \gamma_{j-1})$;
10: $\quad\quad \xi_j = \xi_{j-1} + q_j$;
11: $\quad$ **end if**
12: **end for**
13: **if** $\gamma_k \neq M_c$ **then**
14: $\quad$ **return** reject;
15: **else if** $\rho_k - \xi_k \neq i$ **then**
16: $\quad$ **return** reject;
17: **else** $\{\gamma_k = M_c$ and $\rho_k - \xi_k = i\}$
18: $\quad$ **return** accept;
19: **end if**

Algorithm 3 iteratively computes tuples $(\lambda_j, \rho_j, \delta_j, \gamma_j)$ for each node $v_j$ on the verification path plus a sequence of integers $\xi_j$. If the returned block representation

$\mathcal{T}$ and proof $\Pi$ are correct, at each iteration of the for-loop, the algorithm computes the following values associated with a node $v_j$ of the verification path:

- integer $\lambda_j = l(v_j)$, i.e., the level of $v_j$;
- integer $\rho_j = r(v_j)$, i.e., the rank of $v_j$;
- boolean $\delta_j$, which indicates whether the previous node $v_{j-1}$ is to the right or below $v_j$;
- hash value $\gamma_j = f(v_j)$, i.e., the label of $v_j$;
- integer $\xi_j$, which is equal to the sum of the ranks of all the nodes that are to the right of the nodes of the path seen so far, but are not on the path.

**Lemma 5.3.1.** *If $\mathcal{T}$ is the correct representation of block $i$ and sequence $\Pi$ of length $k$ is the correct proof for $\mathcal{T}$, then the following properties hold for the values computed in iteration $k$ of the for-loop of Algorithm 3:*

1. *Value $\rho_k$ is equal to the number of nodes at the bottom level of the skip list, i.e., the number $n$ of blocks of the file;*
2. *Value $\xi_k$ is equal to $n - i$; and*
3. *Value $\gamma_k$ is equal to the label of the start node of the skip list.*

| ***node*** $v$ | $v_2$ | $v_3$ | $v_4$ | $v_5$ | $w$ | $w_3$ | $w_4$ | $w_5$ | $w_6$ | $w_7$ |
|---|---|---|---|---|---|---|---|---|---|---|
| $l(v)$ | 0 | 0 | 0 | 0 | 1 | 2 | 2 | 3 | 3 | 4 |
| $r(v)$ | 1 | 1 | 2 | 3 | 4 | 5 | 6 | 11 | 12 | 13 |
| $f(v)$ | $\mathcal{T}$ | $\mathcal{T}(m_5)$ | $\mathcal{T}(m_4)$ | $\mathcal{T}(m_3)$ | $f(v_2)$ | $f(v_1)$ | $f(v_6)$ | $f(v_7)$ | $f(v_8)$ | $f(v_9)$ |

Table 5.3: The proof $\Pi'(5)$ as produced by Algorithm 5 for the update "insert a new block with data $\mathcal{T}$ after block 5 at level 1".

### 5.3.6 Updates

The possible updates in our DPDP scheme are insertions of a new block after a given block $i$, deletion of a block $i$, and modification of a block $i$.

To perform an update, the client issues first query $\mathsf{atRank}(i)$ (for an insertion or modification) or $\mathsf{atRank}(i-1)$ (for a deletion), which returns the representation $\mathcal{T}$ of block $i$ or $i-1$ and its proof $\Pi'$. Also, for an insertion, the client decides the height of the tower of the skip list associated with the new block. Next, the client verifies proof $\Pi'$ and computes what would be the label of the start node of the skip list after the update, using a variation of the technique of [125]. Finally, the client asks the server to perform the update on the skip list by sending to the server the parameters of the update (for an insertion, the parameters include the tower height).

We outline the update algorithm performed by the server (performUpdate) in Algorithm 4, and the update algorithm performed by the client (verUpdate) in Algorithm 5. Input parameters $\mathcal{T}'$ and $\Pi'$ of verUpdate are provided by the server, as computed by performUpdate.

Since updates affect only nodes along a verification path, these algorithms run in expected $O(\log n)$ time whp, and the expected size of the proof returned is $O(\log n)$ whp.

To give some intuition of how Algorithm 5 produces proof $\Pi'(i)$, the reader can verify that Table 5.3 corresponds to $\Pi'(5)$, the proof that the client produces from Table 5.2 in order to verify the update "insert a new block with data $\mathcal{T}$ after block 5 at level 1 of the skip list of Figure 5.1". This update causes the creation of two new nodes in the skip list, namely the node that holds the data for the 6-th block, $v_2$, and node $w$ (5-th line of Table 5.3) that needs to be inserted in the skip list at level 1. Note that $f(v_2) = h(0||1||\mathcal{T}, 0||1||\mathcal{T}(\mathsf{data}(v_1)))$ is computed as defined in Definition 5.3.1 and that the ranks along the search path are increased due to the

**Algorithm 5.3.3**: $(\mathcal{T}', \Pi') = \mathsf{performUpdate}(i, \mathcal{T}, \mathsf{upd})$

```
1: if upd is a deletion then
2:     set j = i − 1;
3: else {upd is an insertion or modification}
4:     set j = i;
5: end if
6: set (T', Π') = atRank(j);
7: if upd is an insertion then
8:     insert element T in the skip after the i-th element;
9: else if upd is a modification then
10:    replace with T the i-th element of the skip list;
11: else {upd is a deletion}
12:    delete the i-th element of the skip list;
13: end if
14: update the labels, levels and ranks of the affected nodes;
15: return (T', Π');
```

addition of one more block.

## 5.4 DPDP Scheme Construction

In this section, we present our DPDP I construction. First, we describe our algorithms for the procedures introduced in Definition 5.2.1. Next, we develop compact representatives for the blocks to improve efficiency (blockless verification). In the following, $n$ is the current number of blocks of the file. The logarithmic complexity for most of the operations are due to well-known results about authenticated skip lists [86, 126]. Most of the material of this section also applies to the DPDP II scheme presented in Section 5.6.

### 5.4.1 Core Construction

The server maintains the file and the metadata, consisting of an authenticated skip list with ranks storing the blocks. Thus, in this preliminary construction, we have $\mathcal{T}(b) = b$ for each block $b$. The client keeps a single hash value, called *basis*, which is

**Algorithm 5.3.4**:
$\{\mathsf{accept}, \mathsf{reject}\} = \mathsf{verUpdate}(i, M_c, \mathcal{T}, \mathsf{upd}, \mathcal{T}', \Pi')$

1: **if** upd is a deletion **then**
2: set $j = i - 1$;
3: **else** {upd is an insertion or modification}
4: set $j = i$;
5: **end if**
6: **if** $\mathsf{verify}(j, M_c, \mathcal{T}', \Pi') = \mathsf{reject}$ **then**
7: **return** reject;
8: **else** $\{\mathsf{verify}(j, M_c, \mathcal{T}', \Pi') = \mathsf{accept}\}$
9: from $i$, $\mathcal{T}$, $\mathcal{T}'$, and $\Pi'$, compute and store the updated label $M_c'$ of the start node;
10: **return** accept;
11: **end if**

the label of the start node of the skip list. We implement the DPDP algorithms as follows.

- $\mathsf{KeyGen}(1^k) \rightarrow \{\mathsf{sk}, \mathsf{pk}\}$: Our scheme does not require any keys to be generated. So, this procedure's output is empty, and hence none of the other procedures make use of these keys;

- $\mathsf{PrepareUpdate}(\mathsf{sk}, \mathsf{pk}, F, \mathsf{info}, M_c) \rightarrow \{\mathsf{e}(F), \mathsf{e}(\mathsf{info}), \mathsf{e}(M)\}$: This is a dummy procedure that outputs the file $F$ and information info it receives as input. $M_c$ and $\mathsf{e}(M)$ are empty (not used);

- $\mathsf{PerformUpdate}(\mathsf{pk}, F_{i-1}, M_{i-1}, \mathsf{e}(F), \mathsf{e}(\mathsf{info}), \mathsf{e}(M)) \rightarrow \{F_i, M_i, M_c', P_{M_c'}\}$: Inputs $F_{i-1}, M_{i-1}$ are the previously stored file and metadata on the server (empty if this is the first run). $\mathsf{e}(F)$, $\mathsf{e}(\mathsf{info})$, $\mathsf{e}(M)$, which are output by PrepareUpdate, are sent by the client ($\mathsf{e}(M)$ being empty). The procedure updates the file according to $\mathsf{e}(\mathsf{info})$, outputting $F_i$, runs the skip list update procedure on the previous skip list $M_{i-1}$ (or builds the skip list from scratch if this is the first run), outputs the resulting skip list as $M_i$, the new basis as $M_c'$, and the proof returned by the skip list update as $P_{M_c'}$. This corresponds to calling Algorithm 4 on inputs a block index $j$, the new data $\mathcal{T}$ (in case of an insertion or a modification) and the type of the update

upd (all this information is included in e(info)). Note that the index $j$ and the type of the update upd is taken from e(info) and the new data $\mathcal{T}$ is e($F$). Finally, Algorithm 4 outputs $M_c'$ and $P_{M_c'} = \Pi(j)$, which are output by PerformUpdate. The expected runtime is $O(\log n)$ whp;

- VerifyUpdate(sk, pk, $F$, info, $M_c$, $M_c'$, $P_{M_c'}$) $\rightarrow$ {accept, reject}: Client metadata $M_c$ is the label of the start node of the previous skip list (empty for the first time), whereas $M_c'$ is empty. The client runs Algorithm 5 using the index $j$ of the update, $M_c$, previous data $\mathcal{T}$, the update type upd, the new data $\mathcal{T}'$ of the update and the proof $P_{M_c'}$ sent by the server as input (most of the inputs are included in info). If the procedure accepts, the client sets $M_c = M_c'$ (new and correct metadata has been computed). The client may now delete the new block from its local storage. This procedure is a direct call of Algorithm 5. It runs in expected time $O(\log n)$ whp;

- Challenge(sk, pk, $M_c$) $\rightarrow \{c\}$: This procedure does not need any input apart from knowing the number of blocks in the file ($n$). It might additionally take a parameter $C$ which is the number of blocks to challenge. The procedure creates $C$ random block IDs between $1, \ldots, n$. This set of $C$ random block IDs are sent to the server and is denoted with $c$. The runtime is $O(C)$;

- Prove(pk, $F_i$, $M_i$, $c$) $\rightarrow \{P\}$: This procedure uses the last version of the file $F_i$ and the skip list $M_i$, and the challenge $c$ sent by the client. It runs the skip list prover to create a proof on the challenged blocks. Namely, let $i_1, i_2, \ldots, i_C$ be the indices of the challenged blocks. Prove calls Algorithm 5.3.1 $C$ times (with arguments $i_1, i_2, \ldots, i_C$) and sends back $C$ proofs. All these $C$ proofs form the output $P$. The runtime is $O(C \log n)$ whp;

- Verify(sk, pk, $M_c$, $c$, $P$) $\rightarrow$ {accept, reject}: This function takes the last basis $M_c$ the

client has as input, the challenge $c$ sent to the server, and the proof $P$ received from the server. It then runs Algorithm 3 using as inputs the indices in $c$, the metadata $M_c$, the data $\mathcal{T}$ and the proof sent by the server (note that $\mathcal{T}$ and the proof are contained in $P$). This outputs a new basis. If this basis matches $M_c$ then the client accepts. Since this is performed for all the indices in $c$, this procedure takes $O(C \log n)$ expected time whp.

The above construction requires the client to download all the challenged blocks for the verification. A more efficient method for representing blocks is discussed in the next section.

### 5.4.2 Blockless Verification

We can improve the efficiency of the core construction by employing homomorphic tags, as in [9]. However, the tags described here are simpler and more efficient to compute. Note that it is possible to use other homomorphic tags like BLS signatures [31] as in Compact POR [139].

We represent a block $b$ with its *tag* $\mathcal{T}(b)$. Tags are small in size compared to data blocks, which provides two main advantages. First, the skip list can be kept in memory. Second, instead of downloading the blocks, the client can just download the tags. The integrity of the tags themselves is protected by the skip list, while the tags protect the integrity of the blocks.

In order to use tags, we modify our $\mathsf{KeyGen}$ algorithm to output $\mathsf{pk} = (N, g)$, where $N = pq$ is a product of two primes and $g$ is an element of high order in $\mathbb{Z}_N^*$. The public key $\mathsf{pk}$ is sent to the server; there is no secret key.

The tag $\mathcal{T}(b)$ of a block $b$ is defined by

$$\mathcal{T}(b) = g^b \mod N\,.$$

The skip list now stores the tags of the blocks at the bottom-level nodes. Therefore, the proofs provided by the server certify the tags instead of the blocks themselves. Note that instead of storing the tags explicitly, the server can alternatively compute them as needed from the public key and the blocks.

The Prove procedure computes a proof for the tags of the challenged blocks $m_{i_j}$ ($1 \leq i_1, \ldots, i_C \leq n$ denote the challenged indices, where $C$ is the number of challenged blocks and $n$ is the total number of blocks). The server also sends a combined block $M = \sum_{j=1}^{C} a_j m_{i_j}$, where $a_j$ are random values sent by the client as part of the challenge. The size of this combined block is roughly the size of a single block. Thus, we have a much smaller overhead than for sending $C$ blocks. Also, the Verify algorithm computes the value

$$T = \prod_{j=1}^{C} \mathcal{T}(m_{i_j})^{a_j} \mod N,$$

and accepts if $T = g^M \mod N$ and the skip list proof verifies.

The Challenge procedure can also be made more efficient by using the ideas in [9]. First, instead of sending random values $a_j$ separately, the client can simply send a random key to a pseudo-random function that will generate those values. Second, a key to a pseudo-random permutation can be sent to select the indices of the challenged blocks $1 \leq i_j \leq n$ ($j = 1, \ldots, C$). The definitions of these pseudo-random families can be put into the public key. See [9] for more details on this challenge procedure. We can now outline our main result (for the proof of security see Section 5.5):

**Theorem 5.4.1.** *Assume the existence of a collision-resistant hash function and that the factoring assumption holds. The dynamic provable data possession scheme presented in this section (DPDP I) has the following properties, where $n$ is the current number of blocks of the file, $f$ is the fraction of tampered blocks, and $C = O(1)$ is the number of blocks challenged in a query:*

1. *The scheme is secure according to Definition 5.2.3 based on existence of a*

*collision-resistant hash function and the factoring assumption;*

2. *The probability of detecting a tampered block is* $1-(1-f)^C$*;*

3. *The expected update time is* $O(\log n)$ *at both the server and the client whp;*

4. *The expected query time at the server, the expected verification time at the client and the expected communication complexity are each* $O(\log n)$ *whp;*

5. *The client space is* $O(1)$ *and the expected server space is* $O(n)$ *whp.*

Note that the above results hold in expectation and with high probability due to the properties of skip lists [129].

## 5.5 Security

In this section we, prove the security of our DPDP scheme. While our proof refers specifically to the DPDP I scheme, it also applies to the DPDP II scheme discussed in the next section. Indeed, the only difference between the two schemes is the authenticated structure used for protecting the integrity of the tags.

We begin with the following lemma, which follows from the two-party authenticated skip list construction (Theorem 1 of [125]) and our discussion in Section 5.3.

**Lemma 5.5.1.** *Assuming the existence of a collision-resistant hash function, the proofs generated using our rank-based authenticated skip list guarantees the integrity of its leaves* $\mathcal{T}(m_i)$ *with non-negligible probability.*

**Theorem 5.5.1** (Security of core DPDP protocol)**.** *The DPDP protocol without tags is secure in the standard model according to Definition 5.2.3 and assuming the existence of a collision-resistant hash function.*

*Proof.* As input, the challenger is given a hash function, which he also passes on to the reductor. The challenger plays the data possession game with the adversary using this hash function, honestly answering every query of the adversary. As the only difference from the real game, the challenger provides the reductor the blocks (together with their ids) whose update proofs have verified, so that the reductor can keep them in its storage. Note that *the extractor does not know the original blocks*, only the reductor does. Also note that the reductor keeps updating the blocks in its storage when the adversary performs updates. Therefore, the reductor always keeps the latest version of each block. This difference is invisible to the adversary, and so he will behave in the same way as he would to an honest challenger. At the end, the adversary replies to the challenge sent by the challenger. The extractor just outputs the blocks contained in the proof sent by the adversary. If this proof verifies, and hence the adversary wins, it must be the case that either all the blocks are intact (and so the extractor outputs the original blocks) or the reductor breaks collision-resistance as follows.

The challenger passes all the blocks (together with their ids) in the proof to the reductor. By Lemma 5.5.1, if we have a skip list proof that verifies, but at least one block that is different from the original block (thus the extractor failed), the reductor can output the original block (the –latest verifying version of the– block he stored that has the same block id) and the block sent in the proof as a collision. Therefore, if the adversary has a non-negligible probability of winning the data possession game, the challenger can either extract (using the extractor) or break the collision-resistance of the hash function (using the reductor) with non-negligible probability. □

Next, we analyze our improved DPDP construction that uses tags. We note that the security of our main scheme relies on neither the RSA assumption nor the knowledge of exponent assumption as in [9] since our tags are simpler. In this case,

we need also the following standard assumption:

**Definition 5.5.1** (Factoring assumption)**.** *For all PPT adversaries $A$ and large-enough number $N = pq$ which is a product of two primes $p$ and $q$, the probability that $A$ can output $p$ or $q$ given $N$ is negligible in the size of $p$ and $q$.*

**Theorem 5.5.2** (Security of DPDP protocol with tags)**.** *The DPDP protocol with tags is secure in the standard model according to Definition 5.2.3, assuming the existence of a collision-resistant hash function and that the factoring assumption holds.*

For the proof, we need the following definitions, facts and lemmas.

**Definition 5.5.2.** *Euler's $\phi$ function for $N = pq$ where $p$, $q$ are primes is defined as $\phi(N) = (p-1)(q-1)$.*

**Definition 5.5.3.** *Carmichael $\lambda$ function for $N = pq$ where $p$, $q$ are primes is defined as $\lambda(N) = \mathsf{lcm}(p-1, q-1)$ where $\mathsf{lcm}(x, y)$ denotes the least common multiple of $x$ and $y$.*

**Fact 1.** $\lambda(N) \mid \phi(N)$.

**Lemma 5.5.2** (Miller's Lemma [112])**.** *Let $L$ be a number divisible by $\lambda(N)$. Then, there exists a PPT algorithm that factors $N$ with non-negligible probability, given $L$ and $N$.*

We can now present the proof of Theorem 5.5.2.

*Proof.* The challenger is given a hash function, and an integer $N = pq$ but not $p$ or $q$. The challenger then samples a high-order element $g$ (a random integer between 1 and $N-1$ will have non-negligible probability of being of high order in $\mathbb{Z}_N^*$, which suffices for the sake of reduction argument—a tighter analysis can also be performed). He interacts with the adversary in the data possession game honestly, using the given hash function, and creates the tags while using $N$ as the modulus and $g$ as the base.

As in the previous proof, our challenger will have two sub-entities: An *extractor* who extracts the challenged blocks from the adversary's proof, and a *reductor* who breaks the collision-resistance of the hash function or factors $N$, if the extractor fails to extract the original blocks. As the only difference from the real game, again, the challenger provides the reductor the blocks whose update proofs have verified, so that the reductor can keep them in its storage (and keep updating them). Note that *the extractor does not know the original blocks*, only the reductor does. This difference is invisible to the adversary, and so he will behave in the same manner he would to an honest challenger. At the end, the adversary replies to the challenge sent by the challenger.

First, consider the case where only one block is challenged. If the adversary wins, and thus the proof verifies, then the challenger can either extract the block correctly (using the extractor), or break the factoring assumption or the collision-resistance of the hash function (using the reductor), as follows.

Call the block sent in the proof by the adversary $x$, and the original challenged block stored at the reductor $b$. The extractor just outputs $x$. If the extractor succeeds in extracting the correct block, then we are done. Now suppose the extractor fails, which means $x \neq b$. The challenger provides the reductor with the block $x$ in the proof, its block id, the hash function, and $g, N$. Then the reductor retrieves the original block $b$ from its storage, and checks if $g^x = g^b \mod N$. If this is the case, the reductor can break the factoring assumption; otherwise, he breaks the collision-resistance of the hash function. If $g^x = g^b \mod N$, this means $x = b \mod \phi(N)$ (where $\phi(N)$ denotes the order of $\mathbb{Z}_N^*$, which is $(p-1)(q-1)$), which means $x - b = k\phi(N)$ for some integer $k \neq 0$ (since the extractor failed to extract the original block). Hence, $x - b$ can be used in Miller's Lemma [112], which leads to factoring $N$. Otherwise $g^x \neq g^b \mod N$. This means, there are two different tags that can

provide a verifying skip list proof. By Lemma 5.5.1, the reductor can break the collision-resistance of the hash function by outputting ($g^x \mod N$) and ($g^b \mod N$).

Now consider challenging $C$ blocks. Let $i_1, i_2, \ldots, i_C$ be the $C$ challenged indices. Recall that each block is not sent individually. Instead, the adversary is supposed to send a linear combination of blocks $M = \sum_{j=1}^{C} a_j m_{i_j}$ for random $a_j$ sent by the challenger. We can easily plug in the extractor at the last paragraph of the proof of Theorem 4.3 in [9]. The idea of the extraction is to reset and challenge with independent $a_j$ and get enough independent linear equations that verifies from the adversary to solve for each $m_{i_j}$ (thus, the extractor is just an algebraic linear solver). In the equation $M = \sum_{j=1}^{C} a_j m_{i_j}$, we have $C$ unknowns. Therefore, we can solve for individual blocks $m_{i_j}$ if we get $C$ verifying linearly independent equations on the same blocks. Therefore, if the adversary can respond to a non-negligible fraction of challenges, since the extractor needs only polynomially-many equations, by rewinding polynomially-many times, the extractor can extract the original blocks. If the extractor fails to extract the original blocks, we can employ the reductor as follows.

With each rewind, if the proof given by the adversary verifies, the challenger passes on the $M$ value and the tags in the proof to the reductor, along with the challenge. Call each original blocks $b_{i_j}$. The reductor first checks to see if there is any tag mismatch: $\mathcal{T}(m_{i_j}) \neq g^{b_{i_j}} \mod N$, for some $1 \leq j \leq C$. If this is the case, the reductor can output $\mathcal{T}(m_{i_j})$ and $g^{b_{i_j}} \mod N$ for that particular $j$ as a collision, using Lemma 5.5.1. If all the tags match the original block, the reductor uses the challenge and the ids of the challenged blocks to compute linear combination $B = \sum_{j=1}^{C} a_j b_{i_j}$ of the original blocks he stored. Since the proof sent by the adversary verified, we have $T = \prod_{j=1}^{C} \mathcal{T}(m_{i_j})^{a_j} \mod N = g^M \mod N$. Since all the tags were matching, we have $\mathcal{T}(m_{i_j}) = g^{b_{i_j}} \mod N$ for all $1 \leq j \leq C$. Replacing the tags in the previous equation, we obtain $T = g^B \mod N$. Now, if $M \neq B$, then it leads to factoring using

Miller's Lemma [112] as before (we have $g^M = g^B \mod N$ with $M \neq B$). Otherwise, if $M = B$ for all the rewinds, then the reductor fails, but this means the extractor was successful.

Therefore, if the adversary can respond to a non-negligible fraction of challenges, since the extractor needs only polynomially-many equations, by rewinding the adversary polynomially-many times, the challenger can either extract the original blocks (using the extractor), or break the collision-resistance of the hash function used or the factoring assumption (using the reductor) with non-negligible probability. This concludes the proof of Theorem 5.5.2. □

Concerning the probability of detection, the client probes $C$ blocks by calling the Challenge procedure. Clearly, if the server tampers with a block other than those probed, the server will not be caught. Assume now that the server tampers with $t$ blocks. If the total number of blocks is $n$, the probability that at least one of the probed blocks matches at least one of the tampered blocks is $1 - ((n-t)/n)^C$, since choosing $C$ of $n-t$ non-tampered blocks has probability $((n-t)/n)^C$.

As mentioned before, error-correcting codes can be applied external to our system to further increase the error-resiliency of the file. We do not take into account such modifications when we consider the probability of detection. Also, depending on its usage, some DPDP systems can tolerate some errors, e.g., movie files, music files, most (unofficial) text files, image files, etc. Especially, in a client-server type of usage (as opposed to a P2P usage), Thus, there are many real scenarios where several flipped bits will not cause real problems. More importantly, the probability of getting caught is so high that no respectable DPDP server will take the risks, considering the fact that we also have a public verifiability protocol that can be used for official arbitration purposes. Redundant storage techniques can be applied to further increase resiliency.

## 5.6 Rank-based RSA Trees

We now describe how we can use ideas from [126] to implement the DPDP II scheme (see Table 5.1), which has a higher probability of detection, maintains logarithmic communication complexity but has increased update time.

In [126], a dynamic authenticated data structure called *RSA tree* is presented that achieves constant expected query time (i.e., time to construct the proof), constant proof size, and $O(n^{\epsilon} \log n)$ expected amortized update time, for a given $0 < \epsilon < 1$. We can add rank information to the RSA tree by explicitly storing ranks at the internal nodes. Using this data structure allows the server to answer $O(\log n)$ challenges with $O(\log n)$ communication cost since the proof for a block tag has $O(1)$ size.

The reason for sending additional challenges is the fact that the probability $p$ of detection increases with number $C$ of challenges, since $p = 1 - (1 - f)^C$, where $f$ is the fraction of tampered blocks. Therefore, by using an RSA tree with ranks to implement DPDP, we obtain the same complexity measures as DPDP I, except for the update time, which increases from $O(\log n)$ to $O(n^{\epsilon} \log n)$ (expected amortized), and achieve an improved probability of detection equal to $1 - (1 - f)^{\Omega(\log n)}$.

We now describe how we can use the tree structure from [126] to support rank information. In [126], an $\epsilon$ is chosen between 0 and 1 and a tree structure[2] is built that has $O(1/\epsilon)$ levels, each node having degree $O(n^{\epsilon})$. However, there is no notion of order in [126]. To introduce a notion of order we assume that the elements lie at the leaves of the tree and we view it as a B-tree with lower bound on the degree $t = 3n^{\epsilon}/4$ and therefore upper bound equal to $2t = 3n^{\epsilon}/2$, which are both viewed as constants. Therefore we can use known B-tree algorithms to do the updates with the difference that we rebuild the tree whenever the number of the blocks of the file increases from $n$ to $2n$ or decreases from $n$ to $n/4$. When we rebuild, we set the new constants for

[2]The use of such a tree is dictated by the specific cryptographic primitive used.

the degree of the tree. By the properties of the B-tree (all leaves lie at the same level), we can prove that it is not possible to change the number of the levels of the tree before a new rebuilt takes place. To see that, suppose our file initially consists of $n$ blocks. Suppose now, for contradiction that the number of the levels of the tree changes before a new rebuilt takes place. Note that a new rebuilt takes place when at least $3n/4$ operations (insertions/deletions) take place. We distinguish two cases:

1. If the number of the levels of the tree increases, that means that the number $b$ of the added blocks is at least $n^{1+\epsilon} - n$. Since there is no rebuilt it should be the case that $b \leq 3n/4$ and therefore that $n^{1+\epsilon} - n \leq 3n/4$, which is a contradiction for large $n$;

2. If the number of the levels of the tree decreases, that means that the number $b$ of the deleted blocks is at least $n - n^{1-\epsilon}$. Since there is no rebuilt it should be the case that $b \leq 3n/4$, and therefore that $n - n^{1-\epsilon} \leq 3n/4$, which is again a contradiction for large $n$.

Therefore before a big change happens in the tree, we can rebuild (by using the same $\epsilon$ and by changing the node degree) the tree and amortize. This is important, because the RSA tree structure works for trees that do not change their depth during updates, since the constant proof complexity comes from the fact that the depth is not a function of the elements in the structure (unlike B-trees), but is always maintained to be a constant.

Using the above provably secure authenticated data structure based on [126] to secure the tags (where security is based on the *strong RSA assumption*), we obtain the following result:

**Theorem 5.6.1.** *Assume the strong RSA assumption and the factoring assumption hold. The dynamic provable data possession scheme presented in this section*

*(DPDP II) has the following properties, where $n$ is the current number of blocks of the file, $f$ is the fraction of tampered blocks, and $\epsilon$ is a given constant such that $0 < \epsilon < 1$:*

1. *The scheme is secure according to Definition 5.2.3;*
2. *The probability of detecting a tampered block is $1 - (1 - f)^{\Omega(\log n)}$;*
3. *The update time is $O(n^{\epsilon} \log n)$ (expected amortized) at the server and $O(1)$ (expected) at the client;*
4. *The expected query time at the server, the expected verification time at the client and the worst-case communication complexity are each $O(\log n)$;*
5. *The client space is $O(1)$ and the server space is $O(n)$.*

Note that sending $O(\log n)$ challenges in [9, 10] or DPDP I would increase the communication complexity from $O(1)$ to $O(\log n)$ and from $O(\log n)$ to $O(\log^2 n)$, respectively.

## 5.7 Extensions and Applications

Our DPDP scheme supports a variety of distributed data outsourcing applications where the data is subject to dynamic updates. In this section, we describe extensions of our basic scheme that employ additional layers of rank-based authenticated dictionaries to store hierarchical, application-specific metadata for use in networked storage and version control.

In the extensions described below, the use case and storage guarantees are the same as before: a client would like to store data on an untrusted server, retaining only $O(1)$ space, with the ability to prove possession and integrity of all application data and

metadata. To the best of our knowledge, these are the first efficient constructions for outsourced storage that provide file system and versioning semantics along with proof of possession. In Section 5.8, we show that such systems are efficient and practical.

### 5.7.1 Variable-sized Blocks

Although our scheme enables updates that insert, modify and delete whole blocks of data without affecting neighboring blocks, some applications or file systems may more naturally wish to perform updates that do not cleanly map to fixed-size block boundaries. For example, an update which added or removed a line in a text file would require modifying each of the blocks in the file after the change, so that data in later blocks could still be accessed easily by byte offset (by calculating the corresponding block index). Under such a naïve scheme, whole-block updates are inefficient, since new tags and proofs must be generated for every block following the updated one. A more complicated solution based solely on our existing constructions could store block-to-byte tables in a "special" lookup block.

We now show how we can augment our hashing scheme to support variable-sized blocks (e.g., when we want to update a byte of a certain block). Recall that our ranking scheme assigns each internal node $u$ a rank $r(u)$ equivalent to the number of bottom-level nodes (data blocks) reachable from the subtree rooted at $u$; these nodes (blocks) are conventionally assigned a rank equal to 1. We support variable-sized blocks by defining the rank of a node at the bottom level to be the size of its associated block (i.e., in bytes). Each internal node, in turn, is assigned a rank equivalent to the amount of bytes reachable from it. Queries and proofs proceed the same as before, except that ranks and intervals associated with the search path refer to byte offsets, not block indices, with updates phrased as, e.g., "insert $m$ bytes at byte offset $i$". Such an update would require changing only the block containing the

data at byte index $i$. Similarly, modifications and deletions affect only those blocks spanned by the range of bytes specified in the update.

### 5.7.2 Directory Hierarchies

We can also extend our DPDP scheme for use in storage systems consisting of multiple files within a directory hierarchy. The key idea is to place the start node of each file's rank-based authenticated structure (from our single-file scheme) at the bottom node of a parent dictionary used to map file names to files. Using key-based authenticated dictionaries [125], we can chain our proofs and update operations through the entire directory hierarchy, where each directory is represented as an authenticated dictionary storing its files and subdirectories. Thus, we can use these authenticated dictionaries in a nested manner, with the start node of the topmost dictionary representing the root of the file system (as depicted in Figure 5.2).

This extension provides added flexibility for multi-user environments. Consider a system administrator who employs an untrusted storage provider. The administrator can keep the authenticated structure's metadata corresponding to the topmost directory, and use it to periodically check the integrity of the whole file system. Each user can keep the label of the start node of the dictionary corresponding to her home directory, and use it to independently check the integrity of her home file system at any time, without need for cooperation from the administrator.

Since the start node of the authenticated structure of the directory hierarchy is the bottom-level node of another authenticated structure at a higher level in the hierarchy, upper levels of the hierarchy must be updated with each update to the lower levels. Still, the proof complexity stays relatively low: For example, for the rank-based authenticated skip list case, if $n$ is the maximum number of leaves in each skip list and the depth of the directory structure is $d$, then proofs on the whole file

system have expected $O(d \log n)$ size and computation time whp.

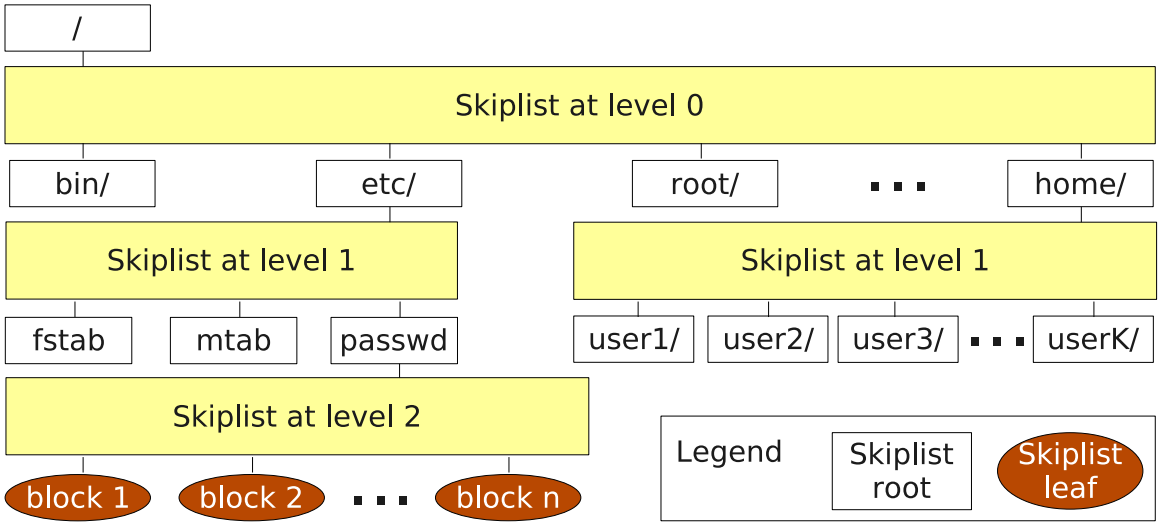


Figure 5.2: A file system skip list with blocks as leaves, directories and files as roots of nested skip lists.

### 5.7.3 Version Control

We can build on our extensions further to efficiently support a versioning system (e.g., a CVS repository, or versioning filesystem). The naïve way to achieve a versioning system is to keep one basis for each version of the file, which requires $O(v)$ client storage and $O(\log n)$ proof complexity, where $v$ is the number of versions of the file. A better system can be supported by adding another additional layer of key-based authenticated dictionaries [125], keyed by revision number, between the dictionaries for each file's directory and its data, chaining proofs as in previous extensions (See Figure 5.3 for an illustration). As before, the client needs only to store the topmost basis; thus we can support a versioning system for a single file with only $O(1)$ storage at the client and $O(\log n + \log v)$ proof complexity, where $v$ is the number of the file versions. For a versioning system spanning multiple directories, let $v$ be the number of versions and $d$ be the depth of the directory hierarchy. The proof complexity for

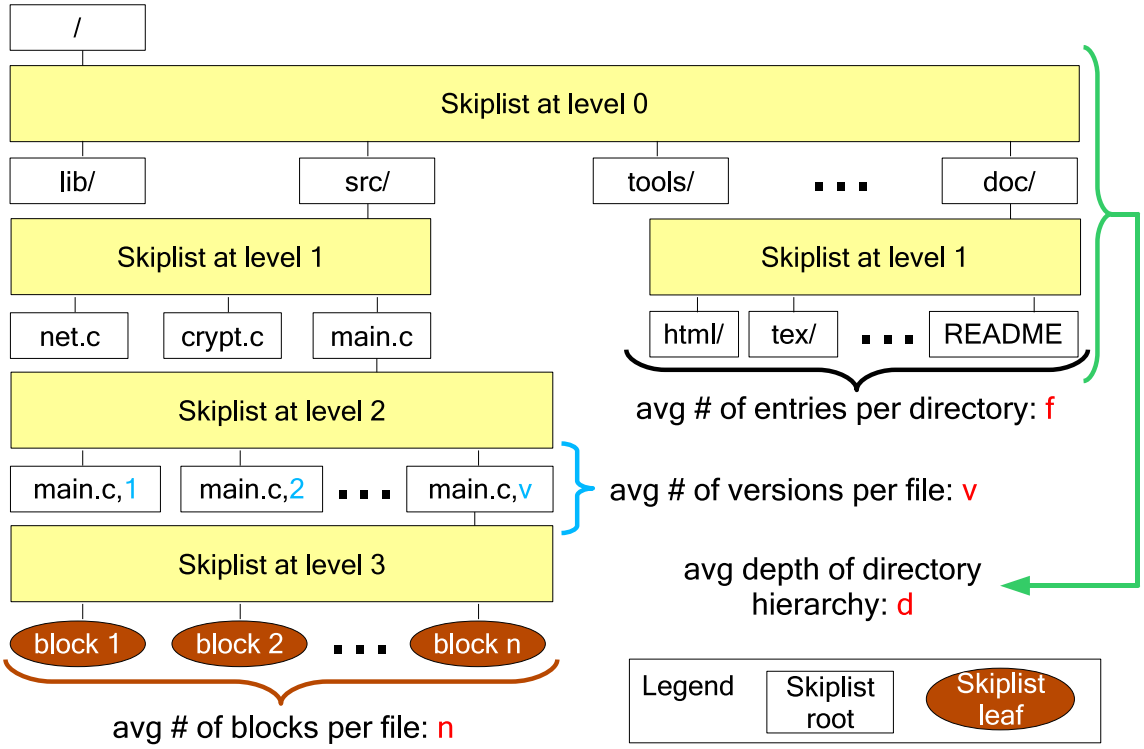


Figure 5.3: A version control file system. Notice the additional level of skiplists for holding versions of a file. To eliminate redundancy at the version level, persistent authenticated skip lists could be used [3]: the complexity of these proofs will then be $O(\log n + \log v + d \log f)$.

the versioning file system has expected size $O(d(\log n + \log v))$.

The server may implement its method of block storage independently from the dictionary structures used to authenticate data; it does not need to physically duplicate each block of data that appears in each new version. However, as described, this extension requires the addition of a new rank-based dictionary representing file data for each new revision added (since this dictionary is placed at the leaf of each file's version dictionary). In order to be more space-efficient, we could use *persistent* authenticated dictionaries [3] along with our rank mechanism. These structures handle updates by adding some new nodes along the update path, while preserving old internal nodes corresponding to previous versions of the structure, thus avoiding

unneeded replication of nodes.

## 5.8 Performance Evaluation

We evaluate the performance of our DPDP I scheme (Section 5.4.2) in terms of communication and computational overhead, in order to determine the *price of dynamism* over static PDP. For ease of comparison, our evaluation uses the same scenario as in PDP [9], where a server wishes to prove possession of a 1GB file. As observed in [9], detecting a 1% fraction of incorrect data with 99% confidence requires challenging a constant number of 460 blocks; we use the same number of challenges for comparison.

Note that even though our scheme requires $O(\log n)$ complexity, as seen in Table 5.4, an update requires $O(\log n)$ hash computations and only one exponentiation. Since hash computations are much faster, this results in real world efficiency. Similarly, proof time at the server side is very efficient, while some more work needs to be done by the client for verification. This is desirable since server will be the party that has more work load and hence needs to be more efficient. We provide actual numbers in the following sections.

| | **Update** | **Proof** | **Verification** |
|---|---|---|---|
| hash computations | $\log n$ | | $C \log n$ |
| multiplications | | $C$ | $C - 1$ |
| additions | | $C - 1$ | |
| exponentiations | 1 | | $C + 1$ |
| skip list lookups | | $C \log n$ | |

Table 5.4: Operations performed during various DPDP actions. $n$ is the number of blocks in the file, and $C$ is the number of challenges (a constant like 460).

In terms of storage, the client needs to store the group (modulus and generator), and the latest skip list root. Using a 1024-bit modulus for the RSA group and 160-bit SHA-1 for the skip list hashes, the client needs to store less than 1KB of data. The server, on the other hand, has a trade-off. The server also needs to store the group

definition that is less than 1KB. It is possible for the server to recompute everything each time it is challenged, but this will slow down the computation at the server side. By storing the tags and the skip list, the server can greatly speed up the computation by paying a storage cost of $\frac{n}{8}$KB for tags and about $\frac{n}{12}$KB for the skip list. For a 1GB file with 65536 blocks, this corresponds to roughly 13MB overhead.

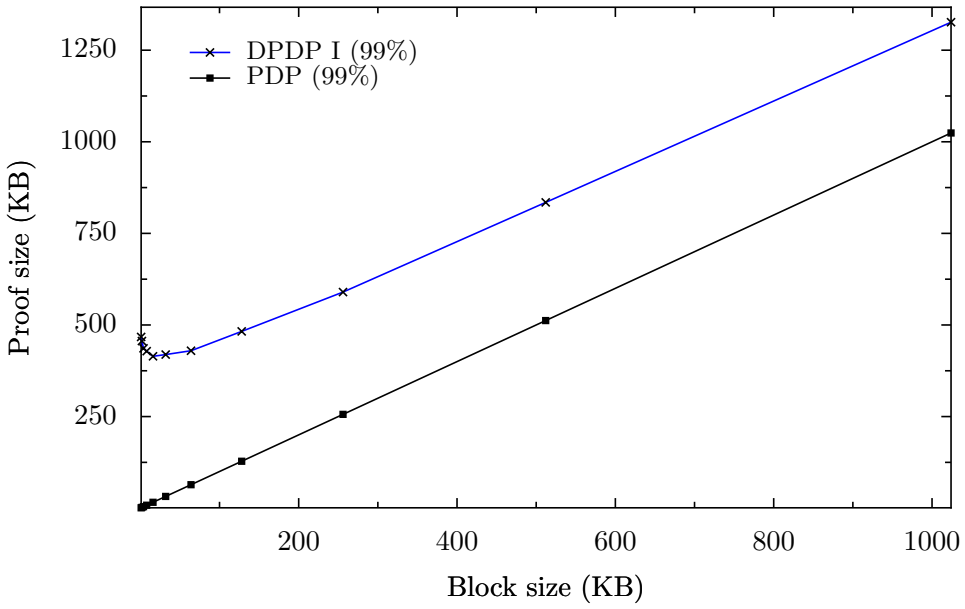


Figure 5.4: Communication cost of dynamism (DPDP I vs. PDP [9]): Size of proofs of possession on a 1GB file, for 99% probability of detecting misbehavior.

### 5.8.1 Communication

The expected size of proofs of possession for a 1GB file under different block sizes is illustrated in Figure 5.4. Here, a DPDP proof consists of responses to 460 authenticated skip list queries, combined with a single verification block $M = \Sigma a_i m_i$, which grows linearly with the block size. The size of this block $M$ is the same as that used by the PDP scheme in [9][3] and is thus represented by the line labeled PDP. The distance

[3]The authors present multiple versions of their scheme. The version without the knowledge of exponent assumption and the random oracle actually sends this $M$; other versions only compute it.

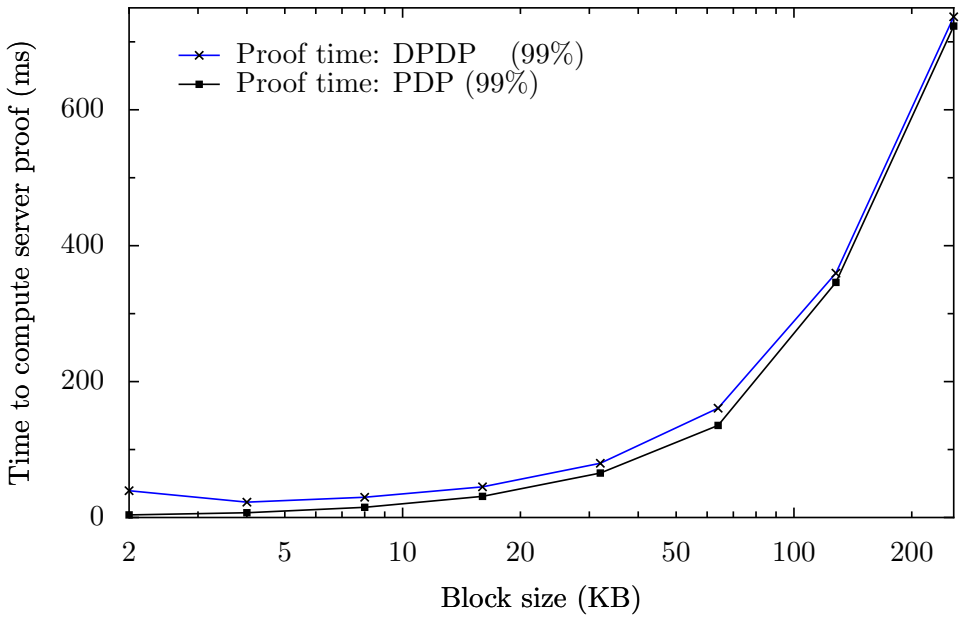


Figure 5.5: Computation cost of dynamism (DPDP I vs. PDP [9]): Computation time required by the server in response to a challenge for a 1GB file, with 99% probability of detecting misbehavior.

between this line and those for our DPDP I scheme represents our communication overhead—the price of dynamism—which comes from the skip list query responses (illustrated in Table 5.2). Each response contains on average $1.5 \log n$ rows, so the total size decreases exponentially (but slowly) with increasing block size, providing near-constant overhead except at very small block sizes.

### 5.8.2 Server Computation

Next, we measure the computational overhead incurred by the server in answering challenges. Figure 5.5 presents the results of these experiments (averaged from 5 trials), which were performed on an AMD Athlon X2 3800+ system with 2GHz CPU and 2GB of RAM. As above, we compute the time required by our scheme for a 1GB file under varying block sizes, providing 99% confidence. As shown, our performance is dominated by computing $M$ and increases linearly with the block size; note that

static PDP [9] must also compute this $M$ in response to the challenge. Thus the computational price of dynamism—time spent traversing the skip list and building proofs—while logarithmic in the number of blocks, is extremely low in practice: even for a 1GB file with a million blocks of size 1KB, computing the proof for 460 challenged blocks (achieving 99% confidence) requires less than 40ms in total (as small as 13ms with larger blocks). We found in other experiments that even when the server is not I/O bound (i.e., when computing $M$ from memory) the computational cost was nearly the same. Note that any outsourced storage system proving the knowledge of the challenged blocks must reach those blocks and therefore pay the I/O cost, and therefore such a small overhead for such a huge file is more than acceptable.

The experiments suggest the choice of block size that minimizes total communication cost and computation overhead for a 1GB file: a block size of 16KB is best for 99% confidence, resulting in a proof size of 415KB, and computational overhead of 30ms. They also show that the price of dynamism is a small amount of overhead compared to the existing PDP scheme.

### 5.8.3 Version Control

Finally, we evaluate an application that suits our scheme's ability to efficiently handle and prove updates to versioned, hierarchical resources. Public CVS repositories offer a useful benchmark to assess the performance of the version control system we describe in Section 5.7. Using CVS repositories for the Rsync [134], Samba [134] and Tcl [121] projects, we retrieved the sequence of updates from the RCS source of each file in each repository's main branch. RCS updates come in two types: "insert $m$ lines at line $n$" or "delete $m$ lines starting at line $n$". Note that other partially-dynamic schemes (i.e., Scalable PDP [10]) cannot handle these types of updates. For this evaluation,

we consider a scenario where queries and proofs descend a search path through hierarchical authenticated dictionaries corresponding (in order) to the directory structure, history of versions for each file, and finally to the source-controlled lines of each file. We use variable-sized data blocks, but for simplicity, assume a naïve scheme where each line of a file is assigned its own block; a smarter block-allocation scheme that collects contiguous lines during updates would yield fewer blocks, resulting in less overhead.

| | **Rsync** | **Samba** | **Tcl** |
|---|---|---|---|
| dates of activity | 1996-2007 | 1996-2004 | 1998-2008 |
| # of files | 371 | 1538 | 1757 |
| # of commits | 11413 | 27534 | 24054 |
| # of updates | 159027 | 275254 | 367105 |
| Total lines | 238052 | 589829 | 1212729 |
| Total KBytes | 8331 KB | 18525 KB | 44585 KB |
| Avg. # updates/commit | 13.9 | 10 | 15.3 |
| Avg. # commits/file | 30.7 | 17.9 | 13.7 |
| Avg. # entries/directory | 12.8 | 7 | 19.8 |
| Proof size, 99% | 425 KB | 395 KB | 426 KB |
| **Proof size per commit** | **13 KB** | **9 KB** | **15 KB** |
| **Proof time per commit** | **1.2ms** | **0.9ms** | **1.3ms** |

Table 5.5: Authenticated CVS server characteristics.

Table 5.5 presents performance characteristics of three public CVS repositories under our scheme; while we have not implemented an authenticated CVS system, we report the server overhead required for proofs of possession for each repository. Here, "commits" refer to individual CVS checkins, each of which establish a new version, adding a new leaf to the version dictionary for that file; "updates" describe the number of inserts or deletes required for each commit. Total statistics sum the number of lines (blocks) and kilobytes required to store all inserted lines across all versions, even after they have been removed from the file by later deletions.

We use these figures to evaluate the performance of a proof of possession under

the DPDP I scheme: as described in Section 5.7, the cost of authenticating different versions of files within a directory hierarchy requires time and space complexity corresponding to the depth of the skip list hierarchy, and the width of each skip list encountered during the Prove procedure.

As in the previous evaluation, "Proof size, 99%" in Table 5.5 refers to the size of a response to 460 challenges over an entire repository (all directories, files, and versions). This figure shows that clients of an untrusted CVS server—even those storing none of the versioned resources locally—can query the server to prove possession of the repository using just a small fraction (1% to 5%) of the bandwidth required to download the entire repository. "Proof size and time *per commit*" refer to a proof sent by the server to prove that a single commit (made up of, on average, about a dozen updates) was performed successfully, representing the typical use case. These commit proofs are very small (9KB to 15KB) and fast to compute (around 1ms), rendering them practical even though they are required for each commit. Our experiments show that our DPDP scheme is efficient and practical for use in distributed applications.

## 5.9 Future Work

### 5.9.1 Other DPDP Constructions

Following ideas from [126], we can modify our scheme in Section 5.6 to implement DPDP III and DPDP IV schemes (see Table 5.6), which are optimized for challenge-intensive or update-intensive workloads, respectively. Both DPDP III and DPDP IV schemes will achieve the same probability of detection as our DPDP I scheme. The reasons they are presented as future work is that we have not analyzed their efficiency carefully, and hence the following table should be taken only as a good guess of what will happen if ideas in [126] will be incorporated to our DPDP scheme keeping the

same tags and the challenge structure.

| **Scheme** | DPDP III | DPDP IV |
|---|---|---|
| **Update time (server)** | $O(n^{\epsilon})$ | $O(1)$ |
| **Challenge time (server)** | $O(1)$ | $O(n^{\epsilon})$ |
| **Client computation** | $O(1)$ | $O(1)$ |
| **Communication** | $O(1)$ | $O(1)$ |
| **Model** | Standard | Standard |
| **Append blocks** | ✓ | ✓ |
| **Modify blocks** | ✓ | ✓ |
| **Insert blocks** | ✓ | ✓ |
| **Delete blocks** | ✓ | ✓ |
| **Prob. of detection** | $1-(1-f)^C$ | $1-(1-f)^C$ |

Table 5.6: Two new DPDP schemes that will incorporate tags and proof techniques described in this chapter, together with ideas from [126]. As before, we denote with $n$ the number of the blocks of the file, with $f$ the fraction of the corrupted blocks, and with $C$ being the number of challenged blocks (typically a constant independent of $n$). In all constructions, the storage space is $O(1)$ at the client and $O(n)$ at the server.

### 5.9.2 On Impossibility of Dynamic Proof of Retrievability Schemes

Another very important line of future work is to bridge the gap between proof of retrievability (POR) line of work [90, 139, 65] and the provable data possession (PDP) line of work [9, 10, 70] (e.g., this chapter). The main difference between the two lines of work can be summarized as follows: POR employs erasure codes (better than error-correcting codes) on top of the file, whereas PDP does not. Thus, the resulting encoded file and block size in POR is bigger, and it is not suitable for **efficient** dynamic updates.

Consider a POR scheme that employs a 2-erasure code (e.g., [139]). This means that the encoded file consists of 2 times the number of blocks in the original file (i.e., $2 * n$) (or each block in the encoded file is 2 times the size of an original block).

Thus, retrieving $n$ blocks of the encoded file (or $n/2$ double-sized encoded blocks) will provide enough information to reconstruct the whole original file. This can be generalized to p-erasure codes that requires $p * n$ encoded blocks (or $n/2$ encoded blocks that are sized $1 + p$ times the original blocks) to reconstruct the original file.

Both POR and PDP works as follows: To verify integrity of a file composed of $n$ blocks, the client challenges the server on $C$ random blocks. If the server has modified (or deleted) a single block (which is the hardest-to-catch damage the server can do), the probability that this block is challenged is $1 - ((n-1)/n)^C$. Note that catching the cheating server when a faulty block is challenged has overwhelming probability. The advantage of using erasure codes in POR is that, even though blocks have larger size, it is enough to make sure the server has not modified (or deleted) more than $n/2$ encoded blocks. As long as the server has modified less than half of the encoded blocks, the reconstruction procedure will work. Therefore, all we need in a POR system is the ability to catch a cheating server who has modified more than half of the blocks. When $C$ blocks are challenged, the probability that no modified block is challenged becomes more than $1 - ((n - n/2)/n)^C = 1 - (1/2)^C = 1 - neg(C)$, which is an overwhelming probability.

This provides a great opportunity of trading extra storage for better security guarantees for static files. The problem of POR is with dynamic files. For the erasure code to work properly in this dynamic setting, modifying a single block must change many blocks in the encoded file. Note that, when an update request is sent to the server, the server must be given a list of encoded blocks to update. If the number of such blocks is small (e.g., $\log n$), then the server can delete those blocks, therefore effectively discarding the original updated blocks, without having a high probability of getting caught. If an update requires changing $t$ encoded blocks and the server deletes those blocks, the probability of catching the server is $1 - ((n-t)/n)^C$. For

$t = \log n$ this probability becomes $1 - ((n - \log n)/n)^C$. To be able to obtain a very high probability, it must be the case that $t = \Omega(n^\delta)$ for some $\delta > 0$, which renders updates inefficient.

This problem is not directly related to the erasure code itself. The erasure code might be changing a smaller number of blocks (e.g., $\log n$), but the underlying dynamic POR (DPOR) scheme might choose to update some $n^\delta$ blocks to hide this information. This can be done if the DPOR scheme keeps the blocks encrypted, and randomly picks some unchanged blocks to re-encrypt and send. Note that the number of randomly-picked unchanged blocks must also be high in order to hide the actual modified blocks when multiple updates are performed over time.

In the future, I am planning to formalize this impossibility proof for efficient DPOR, and provide a construction (most probably based on our DPDP construction in this chapter, together with erasure code ideas from Compact POR [139]) that matches this lower bound. The proof and the construction will feature a parametrized trade-off between update time and detection probability rather than a clear-cut number.

# Chapter 6

# Official Arbitration in the Cloud

## 6.1 Introduction

Consider a case where two parties would like to agree on a message $M$, and then present this agreement as evidence to a trusted authority (called the judge) in case of a dispute. Then, based on messages sent by the parties, the judge can perform an official arbitration. This happens in the cloud in many cases, when two parties agree on a contract. For example, consider a storage outsourcing system such as Amazon S3, Google Documents, or Microsoft Azure. In this scenario, a client outsources storage of her files to a server, but does not necessarily trust the server. In Chapter 5, we provided 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. Unfortunately, such a system is practically useless unless the client or the server can prove anything to the judge in case of a dispute about the data being intact. We call this *official arbitration*, which is a very desirable property in real world applications. In the context of official arbitration, there needs to be an official agreement between the client and the server on this metadata $M$ that can be presented to the judge.

In the case that the message parties would like to agree on does not change over

time, the agreement protocol is equivalent to a signature fair exchange protocol (see Chapter 2 and [7]) on this message. In case of a dispute, those signatures may be presented to the judge for official arbitration purposes. Note that for a fair exchange to be meaningful, there must be a verification algorithm that can verify that the message in the signature is the correct message. This verification algorithm was discussed in detail in Section 2.3. In the case of outsourced storage, this verification algorithm is the one normally run by the client as defined in the provable data possession model [9] in Chapter 5 and [9].

When there is a dispute between the client and the server, the judge rules based on the contract (signatures of the client and the server on the message), and the proof send by the server that must be accepted by the verification algorithm associated with the message in the signatures. If the server corrupts the client's data in the outsourced storage scenario, the client needs to be able to prove to the arbiter that the server corrupted her data. On the other hand, we do not want to enable a malicious client to be able to frame an honest server.

In general, the hard case is when the message in the contract keeps changing over time. In this case, we need both a *dynamic agreement protocol*, and a *dynamic official arbitration protocol*. In a dynamic outsourced storage scenario (dynamic provable data possession –DPDP–) as in Chapter 5, the metadata changes with every update to the file. We need a solution that includes conflict resolution by a trusted judge with possible punishments.

In this chapter we start by defining static and dynamic agreement protocols, and then we define static and dynamic official arbitration protocols. Next, we show how the agreement protocols can be obtained in a naïve way. Then we show a barebone version of our dynamic agreement and dynamic official arbitration protocol that is much more efficient. We then apply this to the dynamic outsourced storage case, and

extend our protocol to include electronic payments (e.g., for automated punishments by the judge).

The main idea behind the agreement protocol is that the client and the server perform a fair exchange of signatures on some message that enables a verifier to check the proof sent by the server. Any time the judge needs to verify that the server keeps the data intact, he is presented by the message signed by both parties. Then, using this message and the proof sent by the server, he runs the official agreement algorithm and declares the judgement. This works perfectly for static storage scenarios in which this fair exchange is performed only once.

For a dynamic protocol, this message may keep changing each time the protocol is run. Hence, one may need to perform a fair exchange of signatures with each run of the underlying protocol. Unfortunately, fair exchange is an expensive operation, due to the fact that all currently known fair exchange protocols require use of an expensive primitive called verifiable escrow. In our barebone protocol, we show how to accomplish the fair exchange of the message without using a verifiable escrow in each exchange. While doing that we make use of the fact that each protocol run can be associated with a monotonic counter (e.g., update serial number).

Once an efficient barebone version of our protocol is ready, we go further and show how this can be applied to the dynamic outsourced storage scenario, and how electronic payments can be incorporated into our official arbitration protocol. We will show electronic payments in the form of electronic cash (e-cash) [50], but other forms of payments such as electronic checks [49] can also be easily used.

We use a model where the client pays the server for the storage initially (e.g., paying for the space) and then performs unlimited updates. It is very straightforward to change our scheme to limit the number of updates or time to store the file per payment: Such a limitation just needs to be included in the contract between the

server and the client, and checked by the judge at dispute resolution time.

Finally, we show how these electronic payments can also be used to automate penalties during dispute resolution. For example, we show how a cheating server can be automatically punished by the judge using electronic payments in our public verifiability protocol. These payments can be used through the operation expenses of the judge.

### 6.1.1 Previous Work

We are not aware of any general message agreement protocol that is optimized for dynamic messages (other than the naïve method of performing a fair exchange with every update). In terms of official arbitration of outsourced storage, [9] and [90] present public verifiability capability of their outsourced storage protocols for static files. Unfortunately, their public verifiability protocols cannot be used for official arbitration purposes, since the client can easily frame the server (by publishing an incorrect public key).

Recently, Wang et al. [148] came up with a dynamic outsourced storage protocol which provides public verifiability for official arbitration purposes. Unfortunately, in their scheme, the trusted authority must be involved in every update, and store the latest metadata for every client-server contract. In contrast, in our scheme, the arbiter is involved only when there is a dispute between the client and the server, and does not need to keep any permanent storage. This is similar to the distinction between the regular and *optimistic* fair exchange. If the reader would like, (s)he can think of our protocols as optimistic dynamic agreement and optimistic dynamic official arbitration protocols.

### 6.1.2 Contributions

Our contributions are as follows:

- We provide the first 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).
- We provide the first efficient optimistic method for official arbitration and agreement protocol in a dynamic setting.
- We provide the first efficient optimistic official arbitration protocol for dynamic provable data possession (DPDP). The efficiency of our protocol renders official arbitration practical by reducing the computational overhead from **7 hours** to **8.35 minutes**, and communication overhead from **610 MB** to **2 MB** for real workloads. Our protocol's overhead corresponds to **0.1 KB** and **24 ms** per update for real CVS repository workloads. Furthermore, the storage requirement is constant, and is less than 84 bytes at both the client and the server.

## 6.2 Agreement and Official Arbitration

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 in Section 2.3. It is enough if one party can generate a proof that is verified

using the verification function associated with the fair exchange protocol. Henceforth, we will call the party who can generate such a proof *the server*, and the other party *the client*. 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, the arbiter gets involved only in case of a dispute.

**Definition 6.2.1** ((Optimistic) Agreement Protocol)**.** *An (optimistic) agreement protocol for a message $M$ is an (optimistic) fair exchange protocol at the end of which the client obtains the signature of the server on the message ($sign_S(M)$) and the server obtains the signature of the client on the message ($sign_C(M)$), or neither party obtains anything.*

We will not repeat the definition of an (optimistic) fair exchange protocol here, but instead redirect the reader to Chapter 2. Since fair exchange cannot be performed without a trusted arbiter [122], 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 proof must also prove that this message is the latest version of the message.

**Definition 6.2.2** ((Optimistic) Dynamic Agreement Protocol)**.** *An (optimistic) dynamic agreement protocol for a list of messages $M_i$ is an (optimistic) fair exchange protocol at the end of which the client obtains the signature of the server on the message and the counter ($sign_S(M_i, i)$) and the server obtains the signature of the client on the message and the counter ($sign_C(M_i, i)$) at the end of round $i$, or neither party obtains anything at that round.*

The definition implies that an (optimistic) dynamic agreement protocol can be

constructed by performing an (optimistic) fair exchange protocol on signatures on the $i^{th}$ message and the counter $i$ sequentially.

An official arbitration protocol (e.g., for an outsourced storage system) involves a client, a server, and a trusted third party called *the judge*. The distinction between the client and the server is based on their capability of proving. The server is the party who can respond to a challenge with a proof. If both parties have this capability, they can play either role. An official arbitration protocol is secure if it guarantees with high probability that a cheating server (e.g., who corrupts the client's data) will be caught, yet a cheating client cannot frame an honest server.

**Definition 6.2.3** (Official Arbitration Protocol)**.** *An official arbitration protocol involves a client $C$, a server $S$, and a judge $J$. The judge, upon receipt of signatures $sign_C(M_C)$ sent by the server and $sign_S(M_S)$ sent by the client, and a proof $\pi_M$ sent by the server, rules that either the client is cheating, or the server is cheating, or otherwise.*

**Definition 6.2.4** (Secure Official Arbitration Protocol)**.** *An official arbitration protocol is secure if the judge rules that the client is cheating if and only if the client is, and that the server is cheating if and only if the server is. The server is considered cheating if the proof $\pi_M$ sent by the server does not verify according to the message $M_S$ signed by himself. The client is considered cheating if $M_S \neq M_C$ while the signatures are correct, and the proof $\pi_M$ verifies with $M_S$.*

An agreement protocol can be used as a backbone to an official arbitration protocol. Once the client and the server agree on message $M$, if there is a dispute between them, they can go to the judge, and officially resolve their dispute according to the official arbitration protocol.

**Definition 6.2.5** (Dynamic Official Arbitration Protocol)**.** *A dynamic official arbitration protocol involves a client $C$, a server $S$, and a judge $J$. The judge, upon receipt*

*of signatures $sign_C(M_{Ci}, i)$ sent by the server and $sign_S(M_{Si}, i)$ sent by the client (for some or all $i : 1..N$), and a proof $\pi_i$ sent by the server (for some or all $i : 1..N$), rules that either the client is cheating, or the server is cheating, or otherwise.*

**Definition 6.2.6** (Secure Dynamic Official Arbitration Protocol)**.** *A dynamic official arbitration protocol is secure if the judge rules that the client is cheating if and only if the client is, and that the server is cheating if and only if the server is. The server is considered cheating if the proof(s) $\pi_i$ sent by the server does not verify according to the messages $M_{Si}$ that are all signed by himself. The client is considered cheating if $\exists M_{Si} \neq M_{Ci}$ for some $i : 1..N$ with all verifying signatures, and all proof(s) $\pi_i$ verify with $M_{Si}$.*

A dynamic agreement protocol is well-suited as a backbone for a dynamic official arbitration protocol. Once the client and the server agree on the latest version of the message, possibly together with the help of the arbiter, they can contact the judge in case of a dispute. In the DPDP scenario, once the client and the server agree on the metadata regarding the latest version of the file, the judge can challenge the server for a proof using the underlying DPDP system. Depending on the verification result, the judge can decide if the server corrupted the client's data, or if the client is trying to frame the server, and rule accordingly.

Our official arbitration protocol in this chapter is applicable on top of any DPDP protocol as long as the client has no secret keys. Below, when we say metadata $M$, we mean all public values required to verify the proof from the server. In particular, it includes the public key of the client.

## 6.3 Efficient Dynamic Agreement and Official Arbitration Protocol

In this section, we will describe an efficient and scalable dynamic agreement protocol that accomplishes agreement of any dynamic data that has a verification algorithm as defined in Section 2.3 optimistically. In particular, we will show that our protocol can be applied to provide dynamic official arbitration on top of any dynamic provable data possession (DPDP) protocol that does not have a secret key, where the verification algorithm is built upon the challenge-response phase of the underlying DPDP protocol. The verification follows the Challenge, Prove, Verify protocols (as defined in Chapter 5), where the Prove protocol is run by the server and Challenge, Verify protocols are run by the client or the judge/arbiter.

The main idea is to keep a counter at both the client side and the server side, and increment that counter with every update. Then, use this counter to sign the message $M$ and exchange signatures. Once there is a dispute between the client and the server, the arbiter/judge can check the signatures to make sure both parties agree on the same message, and then perform verification (e.g., check for the proof of possession of the data).

A naïve way of implementing such a dynamic agreement protocol would be performing an optimistic fair exchange of signatures with every update. This would definitely result in a complete and secure protocol, but will be an overkill in terms of performance. Regular fair exchange is a costly primitive, and thus performing it with every update may cause a performance bottleneck. Below, we present a protocol that is very efficient since it makes use of only simple digital signatures with every update. Then, we show possible dispute resolutions of the protocol, and its security.

At the beginning of the protocol, we assume that both parties know the initial

message $M$, and a counter $ctr$ which is set to 0. For the DPDP case, we assume the client have already uploaded her file to the server. The protocol then begins with an optimistic fair exchange of signatures, where the client obtains the server's signature on the message and counter $(M, ctr)$, and the server obtains the client's signature on the counter $ctr$. Note that the distinction between the server and the client is that the "server" is the party who can provide a proof that will be verified by the arbiter, whereas the "client" does not necessarily have that capability. Therefore, it is enough for the client to just sign the counter $ctr$, and the message $M$ will be signed only by the server.

After this initial fair exchange, the client and the server do not need to perform any more fair exchanges. With each update, both parties increment their counter, and then the client sends her signature on the updated counter to the server, and the server responds with his signature on the new message $M'$ that resulted after the update operation (e.g., of the underlying DPDP protocol), and the updated counter. Note that the server must also prove that the new message $M'$ in the signature passes the verification check. This protocol is depicted on the left side of Figure 6.1.

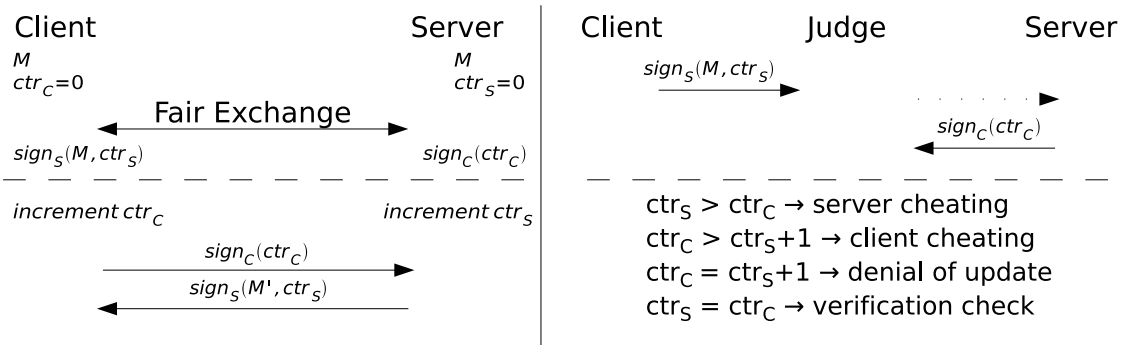


Figure 6.1: Left side shows our dynamic agreement protocol, in which the part above the dashed line is the one time setup and the part below it is performed with every update. Right side shows the arbitration procedure in case of a dispute.

During an update, if the client refuses to send her signature on the updated counter, the server does not perform the update. If the client sends her signature, but

does not receive a response from the server, then she contacts the judge. She presents the judge the most recent signature she received from the server ($sign_S(M, ctr_S)$). The judge then contacts the server, and requests the most recent signature of the client it has. The server responds with $sign_C(ctr_C)$. The judge decides as follows:

- $ctr_S > ctr_C \Rightarrow$ The server is cheating by trying to replay an old signature from the client. The judge punishes the server in this case.

- $ctr_C > ctr_S + 1 \Rightarrow$ The client is cheating by trying to replay an old signature from the server. The judge punishes the client in this case.

- $ctr_C = ctr_S + 1 \Rightarrow$ There are three possibilities in this case. (1) The server did not perform the update the client requested. In this case, the arbiter requests the update on the client's behalf and sends the updated signature of the server to the client. Note that the server must prove that the new message in the signature passes the verification check. If the server refuses to update, then he is punished. (2) The server performed the update but the signature message got lost in the network. The arbiter deals with this case the same as the previous case. (3) Everything went well but the client is trying to overload the judge/arbiter. This can be limited by making the client pay for such requests after some number of free resolutions. Normally, a client should stop working with a particular server if he denies service several times.

- $ctr_S = ctr_C \Rightarrow$ The client and the server agree on the latest message $M$. In this case, the judge challenges the server and verifies the proof sent by the server in response to the challenge. If the verification fails (e.g., the judge realizes that the server corrupted the client's data), he rules for appropriate punishment.

**Theorem 6.3.1.** *The protocol above is a secure dynamic official arbitration protocol.*

*Proof Sketch.* It is obvious that since the client sends her signature first, it is always the case that $ctr_C \geq ctr_S$. Furthermore, since each update increments the counter, the most they can differ by is one: $ctr_C - ctr_S \leq 1$. If these two conditions do not hold, one paty must be cheating and the judge catches in the first two cases above. The other two cases are already explained in text. Furthermore, when the judge runs the Challenge, Prove, Verify protocols for verification purposes, due to the security of the underlying DPDP protocol, our protocol provides secure dynamic official arbitration. □

## 6.4 Payment-Extended Dynamic Official Arbitration Protocol

We now extend our barebone protocol to include possible payments. 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 setup, but it can be repeated after some time or some number of updates. (2) The server pays the client in case of failure to provide service (e.g., data corruption). The server may send a warranty check to the client, who can take the check to the judge and get paid in such a case. (3) The judge/arbiter gets paid for his work by the cheating party. We are assuming the judge has (official) authoritative power over both parties.

At first, in our payment protocol (see Figure 6.2), 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 public key $pk_S$. For example, the contract can specify that the server will keep the client's files intact, will perform updates as requested by the client, and will not perform denial of service. If the client is not happy with the contract, she aborts the

protocol.

If the client is happy with the contract, she can go ahead and send her file for storage at the server. The server then computes the message (e.g., DPDP metadata), and the proof that the initial message is correct (e.g., the metadata corresponds to the file), together with her signature on the proof. As in our barebone protocol, all signatures in our payment protocol will also include a counter kept at both parties, and initialized to 0. Note that in case the message is randomized and cannot be computer by both parties in the same manner, a pseudorandom seed used by the randomized algorithm can be included within the signature.

Now, 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. She then prepares a verifiable escrow of her payment $v = VE_{Arb}(payment; pk_C, pk_S, contract_P)$, labeled using the public key of the client $pk_C$ and the server $pk_S$, and the payment contract. The payment contract specifies the original contract, the message $M$, the signatures to be exchanged, the *receipt* to be given by the server in return for the payment, an optional warranty clause specifying the verifiable escrow of a warranty check we will show below, a timeout for the fair exchange, and possibly some additional details. If there is anything wrong with the verifiable escrow (it does not verify or the label is incorrect) then the server aborts. Note that if the payment is made using e-cash [50], then again the anonymity of the client is preserved.

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 check $w = VE_{Arb}(warranty; pk_C, pk_S, contract_W)$. The warranty contract in the label describes that the warranty check should be decrypted if the server fails to observe the obligations in the contract (e.g., corrupts the client's data).

At this point, if the protocol is not aborted by any party yet, the parties start exchanging the first signatures of the barebone protocol. Namely, the client sends her signature on the counter (initialized to 0), and the server responds with her signature on the message $M$, and the counter (initialized to 0).

Once the signature exchange is done, the server sends the *receipt* which includes the original contract, with terms describing any limits on the number of updates or time, 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. 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–), then the client sends the payment to the server, ending the setup phase.

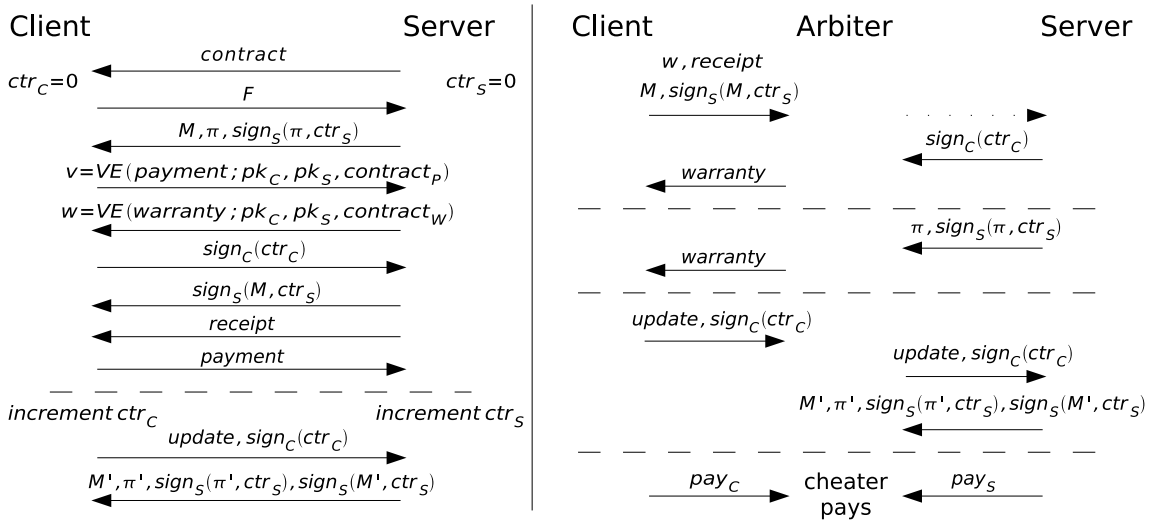


Figure 6.2: Left side above the dashed line is the setup phase of the payment-extended dynamic official arbitration protocol, whereas below the dashed line shows a regular update phase. Right side shows dispute resolution by the judge.

**Theorem 6.4.1.** *The payment protocol above is a fair exchange protocol that guarantees with high probability that the server obtains the client's signature and payment, and the client obtains the server's signature and receipt.*

*Proof Sketch.* The protocol is very similar to the protocol in Chapter 2 and hence we are going to just sketch the proof, mentioning the novel parts. As in that protocol, the verifiable escrow $v$ will be decrypted by the judge/arbiter only if the server provides all necessary material described in the label. These are the signature of the server on the message $M$, and the counter (set to 0), and the *receipt* as detailed above. Note that there is no point for the server to contact the arbiter before he obtains the client's signature, since the arbiter will not be able to provide it, and then she will be caught cheating at the first 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. The client can contact the arbiter after the timeout and obtain the server's signature, and the receipt. □

At the end, the client only needs to keep the message $M$ (e.g., metadata for the DPDP protocol), the signature $sign_S(M, ctr_S)$, the *receipt*, and possibly the warranty $w$. 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 protocol, any third party can verify the server's (dis)honesty (but cannot punish unless he is the trusted judge). Since such a dispute can happen even when no updates were performed, we provide this protocol before the update protocol.

Now that the initial handshake between the client and the server has been completed, we need a protocol to deal with further updates on the message (and on the file). *Each party needs to store only the latest signature received from the other party; no growing history is necessary.* Hence, with every update, both parties need to increment their update counters $ctr$, exchange signatures and keep only these last signatures received. As discussed before, using a fair exchange protocol [7, 6, 5] we can easily achieve this fair exchange of signatures at every update (as long as the message has an associated verification procedure as defined in Chapter 2). But even

the most efficient signature fair exchange protocol due to Asokan, Shoup, and Waidner (ASW) [7] is very inefficient when compared to our new protocol described below (see Section 6.5).

ASW fair exchange is slow due to the use of verifiable escrows, which can be tens of kilobytes in size and take seconds to compute. We use this primitive in our payment protocol, but payment is a one-time operation (or only once in a while if the contract limits time or number of updates) as opposed to a large number of updates that will continuously be performed. Our new protocol completely removes the use of verifiable escrow during updates and uses a very efficient primitive called digital signature [84]. Hence, public verifiability (even in the presence of updates) comes at no cost (only about 60 bytes and a few milliseconds more per update using DSS [119], which is easily dominated by the cost of transferring the updated blocks in DPDP).

In our protocol, with each update command, the counters are incremented. First the client sends the update command together with her signature $sign_C(ctr_C)$ to the server. If the signature does not verify, or the update counter $ctr_C$ is different than the one the server expects, the server ignores this message. Otherwise, he performs the update and computes the new message $M'$ and the associated proof $\pi'$. The server then sends $\pi', sign_S(\pi', ctr_S), sign_S(M', ctr_S)$ to the client. The client checks the proof, updates the message, and also verifies the signatures. If all the checks pass the update is complete, otherwise the client contacts the judge for resolution as shown below.

### 6.4.1 Dispute Resolution

If the update protocol above fails, the client contacts the judge. The client provides the *receipt* as a proof that there is an (storage) agreement between the server and the client. The receipt also includes public keys of the parties. The client provides the

latest signature she has received from the server, namely $sign_S(M, ctr_S)$. In addition, the client can send the latest message $M$, and the warranty escrow $w$.

The judge then contacts the server who provided the receipt, and asks for the latest signature sent by the client. The server responds with $sign_C(ctr_C)$. Then, the judge decides as in the barebone protocol. If the server is found cheating, then the arbiter decrypts $w$ and sends the warranty check to the client.

If the judge suspects that the server corrupted the client's data, he asks for a proof (of data possession) from the server. If this proof does not verify, again, the warranty check will be given to the client. If the proof verifies though, the judge may perform more checks but after some number of free checks, the judge may ask for payment by the client.

In case the judge suspects that the server denied performing the update, then the client provides the judge with the update command she wishes to perform, together with her signature on the counter for that update, just as in a regular update protocol. The judge then requests this update on behalf of the client, possibly in disguise. The server needs to respond properly as in the update protocol. Again, if such a request between the same two participants (identified by their public keys or the receipt) has been repeated many times, the judge may choose to charge the participants. Therefore, a client should stop doing business with a server that fails to comply with the update protocol several times.

A more general denial of service claim (e.g., the client claims that the server is not sending over her data) can be handled in a very similar manner, by the judge obtaining the file from the server and handing it over to the client. Ideally, if the server cannot distinguish the judge's request from a regular user's request, it will be much easier to prevent denial of service; he should never deny a request in order not to be penalized. Again, the judge may get paid after some number of resolutions. In

general, the judge/arbiter can pay for his costs by getting paid by the cheating party. One way to ensure that is to get the payment together with the resolution request, and cash it if a party is found guilty.

### 6.4.2 Analysis

**Theorem 6.4.2.** *The protocol above is a secure dynamic official arbitration protocol.*

*Proof.* Follows from Theorems 6.3.1 and 6.4.1 since the latter proves that the initial exchange performed between the client and the server is a fair exchange, and the former proves that the remaining interactions still provide a secure dynamic official arbitration protocol. □

We now argue that our protocol protects the **privacy** of the client. In our protocols, there are only two pieces of information that can identify the client: the public key of the client and the payment. If the client chooses a new public key $pk_C$ for each contract, and if the payments are made using e-cash, none of these can be used to identify the client. The privacy of the client can be preserved even when the client contacts the judge using anonymous routing techniques like onion routing [61]. Furthermore, the client can always store encrypted blocks at the untrusted server, hence keeping even the file itself private. We assume that the message itself is not enough for identifying the client, since this is definitely the case in DPDP where the message is the metadata, which is the root of a skip list.

Since in a real scenario we expect servers to be well-known entities, our focus will be on client anonymity. Note that the server warranty can be made using electronic checks since we are assuming the server is not anonymous. Yet, the server can still be somewhat anonymous. Just as in the DPDP protocol the client needs a way to reach the server, the judge also needs that to request the signature from the server.

Thus, our official arbitration protocol with its resolution can be applied on top of any DPDP protocol, even on top of the **peer-to-peer** ones.

Lastly, our dynamic official arbitration and dynamic agreement protocol can be of independent interest in fair multi-exchange scenarios where the next exchange is defined within the current one (e.g., in our case, the update counter *ctr* serves that purpose). Throughout this chapter we presented a concrete example of such a case based on the DPDP scenario. Yet, our protocols can be applied as a dynamic agreement protocol for any message that can somehow be verified, and keeps getting updated. In general, our protocols are applicable on top of any dynamic provable data structure, including authenticated skip lists [124], authenticated hash tables [48], and many other authenticated data structures [67].

## 6.5 Performance Evaluation

As we discussed before, fair exchange of signatures is a costly operation. The cost of a fair exchange is more than the cost of a verifiable escrow. In our implementation of the most efficient verifiable escrow known to us [45], each verifiable escrow takes about 1 second and 25 KB. On the other hand, each DSS signature [119] takes about 1 ms and 40 bytes.

To provide a real usage scenario and numbers, we consider the CVS repositories depicted in Table 5.5 in Section 5.8.3. Using the naïve method, which is the only existing method that we can compare our method to, every commit would necessitate a fair exchange. For roughly $25,000$ commits in Table 5.5, 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. On the other hand, using our scheme, including the initial fair exchange, providing public verifiability for all $25,000$ commits will

require **51 seconds** and **2 MB**. The improvement from 7 hours to 51 seconds, and from 610 MB to 2 MB makes our approach the first ever practical, efficient, and officially usable public verifiability protocol for dynamic provable data possession. **Per update/commit**, our overhead is only **0.1 KB** and **2 ms**.

Note that our protocol requires **no growing history**. Only a counter and the latest signature on the last message needs to be kept at each party, and the client also keeps the previous to last. In any case, the storage overhead for both parties is less than **84 bytes**.

# Chapter 7

# Practicality of the Cloud

## 7.1 Introduction

There is often a large gap between protocols that are developed and proposed in theory and protocols that are actually implemented and used in practice. In general, cryptographic protocols are fairly complicated in design, and as such their implementation requires great care. Unfortunately, one cannot expect all good cryptographers to be good programmers, or vice versa. As a result, many useful cryptographic protocols go unimplemented, despite the fact that they may be efficient enough to be used in practice.

We hope to bridge this gap by providing a library, CashLib, that builds upon our language [68] to implement some state-of-the-art protocols such as electronic cash (e-cash), blind signatures, verifiable encryption, and fair exchange. Our library implements many of the protocols required by a cryptosystem built from discrete-logarithm-based zero-knowledge proofs of knowledge, and provides these primitives to the programmer as language features and library operations.

The design and implementation of our library were motivated by collaborations

with systems researchers interested in employing zero-knowledge protocols in high-throughput applications, such as a peer-to-peer file sharing system [16] in which peers pay each other to download files (or file blocks). The performance concerns arising from this work, and the complexity of the protocols required, have motivated our library's focus on performance and ease of use for both the cryptographers designing the protocols and the systems programmers charged with putting them into practice.

Having the fair exchange mechanism described in Chapter 2, and the overall plan for using it in P2P file sharing [16], our group started implementing its own file sharing system, backwards compatible with the existing BitTorrent framework. To this end, we developed a cryptographic library that contains efficient zero-knowledge proofs, electronic cash, and fair exchange [68]. The pseudocodes for the algorithms used in our library can be found in Appendix A. To the best of our knowledge, such a comprehensive and detailed pseudocode compilation on zero-knowledge proofs, blind signatures, e-cash, verifiable encryption and fair exchange was not done before.

Our group is already using this cryptographic library in our BitTorrent client. The results of this effort will soon be available [69], proving the real-world usability, efficiency and scalability of the fair exchange protocols in this thesis. We are constantly working together as a group to ensure the cryptographic protocols we have are easily usable and that they satisfy the actual needs of our BitTorrent client.

Our goal is to free the programmer from having to worry about the implementation of cryptographic primitives, efficient mathematical operations, generating and processing messages, etc. Our library makes performance optimizations based on analysis of the protocol description itself. As we will show, this provides greater opportunities for performance improvement than those available to lower-level libraries for cryptography and multiple-precision arithmetic (Gnu Multi-Precision library).

This chapter describes the design and implementation of our library. We also describe experiences and performance optimization techniques that we undertook while implementing the library, and provide benchmarks to show the efficiency provided by our approach and implementation.

### 7.1.1 Related work

Idemix [88, 35, 26] project, under development at IBM Research, is similar to our work as it is also focused on the development of a library for blind signatures and zero-knowledge proofs. Our focus is on performing optimizations, rather than performing the translation to code. Having a centralized framework that performs execution enables for optimizations on the system as a whole. Furthermore, while Idemix is designed mainly for use with anonymous credential systems, to the best of our knowledge, we provide the only existing implementation of e-cash, verifiable encryption, and optimistic fair exchange.

Another system, FairPlayMP [21], provides a framework for secure multi-party computation (FairPlay [106] being its two-party version), a cryptographic primitive that allows multiple parties to jointly compute a function on private inputs while revealing nothing but the resulting value. Having an implementation of a generic multi-party computation is certainly a useful tool, but the generic circuit-based techniques used in FairPlayMP can be quite slow. For many applications, there are specialized schemes that are much more efficient. In contrast, we limit our focus to discrete-logarithm-based zero-knowledge proofs, allowing us to make use of specialized efficient algorithms, as well as optimize our implementation for practical everyday use.

There are also other special-purpose libraries. The Advanced Crypto Software Collection [25] is a collection of cryptographic libraries that implement some advanced

cryptographic primitives such as attribute-based encryption and forward-secure signatures. Many of these libraries use at their core the Pairing-Based Crypto library [104] which implements arithmetic operations over various types of elliptic curves.

### 7.1.2 Contributions

We have implemented a cryptographic library, in C++, to provide higher-level cryptographic protocols including blind signatures, e-cash, verifiable encryption, and optimistic fair exchange. The library performs optimizations such as pre-computation of expected exponentiations, preventing redundancy in proofs, and caching. Further details on optimizations we made on existing protocols are provided in Section 7.3. We provide some benchmark numbers for our implementations of these primitives and argue that they are usable in real scenarios.

We plan to make the source code for our library freely available online. We hope that our efforts will encourage cryptographers and programmers to use (and extend) our library to implement their protocols. We welcome contribution by our fellow researchers in this effort.

## 7.2 Cryptographic Background

We assume the reader is familiar with the basic properties and definitions of hash functions (in particular, universal one-way hash functions [118] and Merkle hashes [109]), symmetric and asymmetric encryption schemes, digital signature schemes [84], pseudo-random functions [80], commitment schemes [128, 76, 58], and zero-knowledge proofs of knowledge [83, 82, 17]. Further necessary cryptographic background to understand this chapter is given below, and more detailed descriptions of the specific protocols we use are in Appendix A.

**Sigma proofs** Sigma proofs [59] are three-round honest-verifier zero-knowledge proofs of knowledge. When used together with the Fiat-Shamir heuristic [74], they can be used as non-interactive zero-knowledge proofs of knowledge for *any* verifier, secure in the random oracle model [19].

We currently support four main types of relations:

- Proving knowledge of the opening of a commitment [135]. We can prove openings of Pedersen or Fujisaki-Okamoto commitments [128, 76, 58], in prime-order or special RSA groups respectively. In both cases we allow for commitments to multiple values.

- Proving equality of the openings of different commitment. Given any number of commitments, containing any number of bases and exponents, we can prove the equality of any subset of the exponents in the commitments. It is also possible to prove that different subsets of exponents match within different commitments.

- Proving that a committed value is the product of two other committed values [58, 32]. As shown in our sample program, we can prove that a value $x$ contained within a commitment is the product of two other values $y, z$ contained within two other commitments; i.e., $x = y \cdot z$. As a special case, we can also prove that $x = y^2$.

- Proving that a committed value is contained within a public range [32, 103]. First, one can prove that a committed value $x$ is non-negative, so $x \geq 0$. Then for a range of the form $lo \leq x < hi$, we can see that it suffices to prove that $(x - lo)(hi - x) \geq 0$ to guarantee that $x$ is contained within the range.

**Blind signatures** Blind signatures, introduced by Chaum [50], enable a signature issuer to sign a message without knowing the contents of the message. In general,

this is done by the signature recipient picking a random value and using it to blind the message; this blinded value is then sent to the issuer, who signs it to produce a partial signature. The recipient, upon receipt of this partial signature, un-blinds it to obtain a signature on the original message.

Some blind signatures, such as the scheme we employ due to Camenisch and Lysyanskaya (CL) [43], also provide protocols that allow the owner of a signature to prove that she has a signature on a committed value, while revealing neither the value nor the signature. A pseudocode of the CL protocols for obtaining and proving possession of a signature can be found in Appendix A.6.

**Electronic cash** Electronic cash (e-cash), also introduced by Chaum [50], can be thought of as the electronic equivalent of cash; i.e., an electronic currency that preserves users' anonymity, as opposed to electronic checks [49] or credit cards. An e-cash system provides protocols for a user to withdraw money from the bank and spend that money, as well as a way for merchants to deposit money with the bank. We are furthermore interested in *offline* e-cash systems where the bank need not be active in every transaction. Such an e-cash system must also provide means to detect double-spenders; i.e., users who try to spend the same coin twice.

We implement endorsed e-cash [44] (which is an extension of compact e-cash [40]), because it enables efficient fair exchange. Endorsed e-cash splits a coin into an unendorsed coin (denoted $coin'$) and endorsement (denoted $end$). One can think of $coin'$ as an encrypted coin and $end$ as the decryption key. Just as no one can figure out the key or the message given the ciphertext alone, given only the unendorsed part $coin'$, no other party (except the owner) can come up with a valid endorsement $end$ to obtain the original coin. One coin can correspond to multiple $coin', end$ pairs (consider them as two encryptions using different randomness). Endorsed e-cash has the ability to catch double-spenders, however, so that if one uses two different $coin', end$ pairs to

try to spend the same coin twice, she will be caught and her identity will be revealed (so she can be punished accordingly). Note that if a party tries to deposit the same coin twice (using the same $coin', end$ pair), the operation can easily be denied by checking against a list of past transactions. Lastly, only matching $coin', end$ pairs can be linked; two separate coins, even from the same user, are always unlinkable.

Because the unendorsed coin does not contain the endorsement (hence the name), we can see that the user can safely give $coin'$ to the merchant without revealing any private information. The user and the merchant can then engage in a fair exchange in which the merchant obtains the endorsement $end$ on $coin'$ and the user obtains the item she wanted to buy. During this fair exchange, the user will verifiably encrypt the endorsement $end$. In order to check that a coin is valid, a merchant can check the verifiable escrow against values contained in the unendorsed coin $coin'$, as well as check the validity of $coin'$ itself.

During the withdrawal phase of endorsed e-cash, a user contacts the bank. Before withdrawing, the user will have registered with the bank by storing a commitment. In order to prove her identity, then, the user will provide a proof that she knows the opening of the registered commitment.

Once the bank has verified this proof, the user and the bank will run a protocol to obtain a CL signature (using the pseudocodes in Appendix A.7) on the user's identity and two pseudo-random function seeds. These private values and the signature on them define a wallet that contains $W$ coins (where $W$ is a system-wide public parameter).

When a user wishes to spend the $J$-th coin in her wallet, she prepares two serial numbers using the Dodis-Yampolskiy pseudo-random function [64] on the seeds from the wallet and the index $J$. She then blinds these serial numbers and uses the randomness from this process as the endorsement $end$ for the coin. The unendorsed

coin $coin'$ will include these serial numbers as well as a proof that they are formed correctly (i.e., using the wallet seeds and the value $J$), a commitment to the user's wallet, a randomized CL signature, a proof that the randomized CL signature is a valid signature from the bank, and a proof that the coin index is proper, so that $1 \leq J \leq W$.

**Verifiable escrow** In the cryptographic context, an *escrow* can be considered a ciphertext under the public key of some trusted third party (called here the arbiter). A *verifiable* escrow [7, 45, 37] means that the recipient of the escrow can verify that the contents of the ciphertext satisfy some relation (in particular, the recipient can be assured that the ciphertext contains the expected content). A label attached to such a ciphertext defines the conditions under which the arbiter should decrypt and give away the encrypted secret [143]. The label is public and is integrated with the ciphertext in such a way that it cannot be modified, thereby guaranteeing the creator of the escrow that the secret will not be unduly revealed. In the rest of this chapter, we will use $E_{Arb}(a; b)$ to denote an escrow of the secret $a$ under the arbiter's public key, using the contract $b$ as a label. Similarly, $VE_{Arb}(a; b)$ will denote a verifiable escrow.

Our implementation of verifiable encryption is based on the construction of Camenisch and Shoup [45], due to its efficiency. The main use of verifiable encryption in e-cash is to allow a user to verifiably encrypt the opening of a commitment under the public key of the arbiter. A recipient of such a verifiable escrow can verify that the encrypted values correspond to the opening of the commitment and can also check that the label associated with the escrow was formed correctly. More details of the Camenisch-Shoup verifiable encryption scheme can be found in Appendix A.8.

**Optimistic fair exchange** Fair exchange is a two-party protocol between users; call them Alice and Bob. In a typical fair exchange, Alice has something that Bob wants, and Bob has something that Alice wants. A fair exchange protocol guarantees that at the end either both of them obtain what they want, or neither of them do; these protocols are necessary for any electronic transaction in which an e-coin is exchanged with a digital item (file). Unfortunately, it is impossible to achieve fair exchange protocols without the use of a trusted third party [122] (the arbiter) that ensures that Alice cannot take advantage of Bob, and vice versa. In *optimistic* fair exchange, the TTP gets involved only in the case of a dispute between the two parties. Descriptions of the fair exchange protocols we implement in our library can be found in Chapter 2 and [16] (and we have some modifications described in Section 7.3).

## 7.3 Implementation of Cashlib

Using the primitives described in the previous section, we wrote a cryptographic library designed for optimistic fair exchange protocols. The library was written in C++ and consists of approximately 17000 lines of code. In the process of implementing some of the protocols we describe below, we found ways to optimize them for both efficiency and usability, and so we describe those optimizations as well.

Our framework currently allows for three different types of groups: prime-order groups, special RSA groups (i.e., groups of order $n = pq$, where $p$ and $q$ are safe primes), and Paillier groups [123] (i.e., groups of the form $Z^*_{n^2}$, where $n = pq$). We consider security levels of 80, 112, and 128 by default, although we also allow for the security level to be chosen by the user.

### 7.3.1 Modifications to Endorsed E-cash

A description of endorsed e-cash can be found in [44]. The version used in our library, however, contains a number of optimizations. Just as with real cash, we now allow for different coin denominations. Each coin denomination corresponds to a different bank public key, so once the user requests a certain denomination, the wallet is then signed using the corresponding public key. A coin generated from such a wallet will only verify when the same public key of the bank is used, and thus the merchant can check for himself the denomination of the coin.

Similarly, we modified endorsed e-cash to have a wallet size that is per-wallet, instead of being system-wide. This wallet size is signed by the bank during withdrawal, and presented during spending. To prevent linkability through wallet sizes, we allow user to pick a wallet size only among a pre-defined set of wallet sizes. With millions of users at a bank, when each wallet size is used by hundreds of thousands of users, linkability will not constitute a problem.

We also randomize the user's spending order rather than having them perform a range proof that the coin index was contained within the proper range. As the random spending order does not reveal how many coins are left in the wallet, the user's privacy is still protected even though the index is publicly available. Furthermore, because range proofs are slow and require a fair amount of space (see Figure 7.1 for a reminder), this optimization resulted in coins that were 17% smaller and 23% faster to generate and verify.

Finally, endorsed e-cash requires a random value contributed by both the merchant and the user. Since e-coin transactions should be done over a secure channel, in practice we expect that SSL connections will be used between the user and the merchant. One useful feature of an SSL connection is that it already provides both parties with shared randomness, and thus this randomness can be used in our library

to eliminate the need for a redundant message.

### 7.3.2 Modifications to Buying and Bartering

In our library, we implement the two most efficient optimistic fair exchange protocols known to date. Belenkiy et al. [16] provide a *buy* protocol for exchanging a coin with a file, while Chapter 2 provides a *barter* protocol for exchanging two files or blocks. The two protocols serve different purposes (buy vs. barter) and so we have implemented both.

Two of the main usage scenarios of fair exchange protocols are e-commerce and peer-to-peer file sharing [16]. In e-commerce, one needs to employ a buy protocol to ensure that both the user and the merchant are protected; the user receives her item while the merchant receives his payment. In a peer-to-peer file sharing scenario, peers exchange files or blocks of files. In this setting, it is more beneficial to barter for the blocks than to buy them one at a time; for an exchange of $n$ blocks, buying all the blocks requires $O(n)$ verifiable escrow operations (which, as discussed in Section 7.3.3, are quite costly), whereas bartering for the blocks requires only one such operation, regardless of the number of blocks exchanged.

Although the solution might seem to be to barter all the time and never buy, Belenkiy et al. suggest that both protocols are useful in a peer-to-peer file sharing scenario. Peers who have nothing to offer but would still like to download can offer to buy the files, while peers who would only like to upload and have no interest in downloading can act as the merchant and earn e-cash. Due to the resource considerations mentioned above, however, bartering should be preferred if possible.

Because peers do not always know beforehand if they want to buy or barter for a file, we have modified the buy protocol to match up with the barter protocol in the first two messages. We further modified both protocols to let them exchange multiple

| Program type | Time ($ms$) | | Size (bytes) | Cache size (MB) |
|---|---|---|---|---|
| | Prover | Verifier | | |
| DLR proof | 3.42 | 1.37 | 511 | 0 |
| Multiplication proof | 2.66 | 2.07 | 848 | 33.5 |
| Range proof | 51.35 | 25.45 | 5455 | 33.5 |
| CL recipient proof | 154.09 | 86.44 | 19189 | 134.2 |
| CL issuer proof | 8.88 | 1.95 | 1097 | 0 |
| CL possession proof | 167.45 | 89.69 | 19979 | 134.2 |
| Verifiable encryption | 584.14 | 153.25 | 24501 | 190.2 |
| Coin | 176.29 | 95.04 | 22526 | 223.7 |

Figure 7.1: Time (in milliseconds) and size (in bytes) required for each of our proofs, averaged over twenty runs. Timings are considered from both the prover and verifier sides, and are considered with caching for fixed-based exponentiations; the size of the cache is also measured (in megabytes). The numbers for CL proofs were obtained using a CL signature on three private values and one public value, and the numbers for VE were obtained using a verifiable escrow on three values (as would be done in e-cash).

blocks at once, so that one block of the fair exchange protocol might correspond to multiple blocks of the underlying file.

We have also implemented the trusted third parties (the bank and the arbiter) necessary for e-cash and fair exchange, and we provide performance benchmarks for the bank in Figure 7.2.

### 7.3.3 Performance of Primitives

Here we give some benchmarks for primitives in our library, both in terms of communication and computational complexity. These numbers were collected on a MacBook with a 2.4GHz Intel Core 2 Duo processor and 4GB of RAM running OS X 10.6; we expect that these numbers will therefore reflect those of an average user.

Looking back at the pseudocodes for our various proofs and seeing which primitives are involved in each (see Appendix A), the numbers in Figure 7.1 make sense. For example, the marked difference between the time required to generate a CL issuer

proof and a CL possession proof can be attributed to the fact that a CL issuer proof requires proving only one discrete log relation, while a CL possession proof on three private values requires three range proofs, in addition to five more discrete log relations.

Figure 7.1 also shows that verifiable encryption is by far the biggest bottleneck, requiring three times as much computation time as any other step. As seen in the program in Appendix A, there is one range proof performed for each value contained in the verifiable escrow. In order to perform a range proof, the value contained in the range must be decomposed as a sum of four squares. Because the values used in our verifiable encryption program are much larger than the ones used in CL signatures, this decomposition often takes considerably more time (up to 600ms) for verifiable encryption than it does for CL signatures. Furthermore, since the values being verifiably encrypted are different each time, caching the decomposition of the values won't do us any good.

One final observation on computation time is that the time required to prove possession of a CL signature completely dominates the time required to prove the validity of a coin, as demonstrated by the fact that the numbers for the two proofs are nearly identical. This suggests that the only way to get significantly faster numbers for the coin, as well as for verifiable encryption, would be to come up more efficient techniques for range proofs (which has in fact been the subject of some recent research [36]).

In terms of proof sizes, the range proofs are quite a lot larger than the proofs for discrete logarithms or multiplication. This is to be expected, as the translations of range proofs into their discrete logarithm representation (as shown in Appendix A) requires 11 discrete log equations, whereas a single DLR proof requires only 1 and a multiplication proof requires 2.

| Operation | Time ($ms$) | Size (bytes) |
|---|---|---|
| Withdraw (user) | 192.74 | 20093 |
| Withdraw (bank) | 114.96 | 1167 |
| Deposit (bank) | 95.44 | 22526 |
| Buying a block (buyer) | 778.81 | 47286 |
| Buying a block (seller) | 254.75 | 203 |
| Barter setup message | 709.97 | 46934 |
| Checking setup message | 249.75 | n/a |
| Barter after setup (initiator) | 21.54 | 1280 |
| Barter after setup (responder) | 1.22 | 204 |

Figure 7.2: Average time required and network overhead, in milliseconds and bytes respectively, for each stage in our e-cash implementation. The timings were averaged over twenty runs, and caching and compression optimizations of the library are used.

### 7.3.4 Performance of High Level Protocols

In Figure 7.2, we can see the computation time and size complexity for the steps described above, as well as computation and communication overhead for the withdraw and deposit protocols involving the bank. The numbers in the table were computed on the same computer as those in Section 7.3.3, using security parameters that provide a security level of 80 bits.

If we look back at the results in Figure 7.1, the numbers in Figure 7.2 should not be surprising. As mentioned before, bartering is quite a lot more efficient than buying, both in terms of computation and communication overhead. We can see that the setup message for both buying and bartering takes about 700ms to generate and approximately 46KB of space. In contrast, the rest of the barter protocol takes very little time; on the order of milliseconds for both parties (and about 1.5KB of total overhead). Therefore, bartering really will be more efficient as long as more than one exchange is expected to take place.

## 7.4 Conclusions and Future Work

Implementing cryptographic systems is a hard task, due in large part to the complexity of the protocols involved. We built a library that provides state-of-the-art optimistic fair exchange protocols based on electronic cash, based on widely-used discrete-logarithm-based zero-knowledge proofs of knowledge. Our library is already in use by our collaborators [34], and soon we are planning to release our library for use by other researchers.

Finally, in terms of extending the library, in order to improve a bank's efficiency, it might also possible to speed up coin verification time by supporting batch verification techniques [41, 73] for CL signatures; we leave this as one of many interesting open problems.

In Chapter 4, we proposed a solution in which the bank outsources some of its computation to untrusted contractors; these contractors can then be rewarded and fined as necessary. Using this framework, one might outsource the job of coin verification, thus taking some of the burden off the bank and allowing the system to scale more efficiently. Necessary e-cash primitives are already implemented in our library, and hence realizing this framework might as well be a near-term future work.

Issues of decentralization also arise with the arbiter for fair exchange. In this case, one might employ secret sharing techniques [140, 27] to distribute the trust placed on the arbiter (see Chapter 3).

# Chapter 8

# Conclusion and Future of the Cloud

We have shown that providing security and privacy in many cloud systems is possible at virtually no cost: There are very efficient and scalable privacy-preserving cryptographic protocols that provide necessary security features to such systems. Considering many failed attempts of trying to incorporate security to a system after its design (e.g., incorporating IPSec to the Internet), the next generation cloud systems must be designed with security and privacy in mind. And now that we have the enabling cryptography, I see no reason not to, since security and privacy features benefit both the user and the service provider (see Brands for a very nice advocacy [33]).

In general, in this thesis we are interested in a broad definition of fairness in the cloud, as in being fair to users. The goal is to design protocols that ensure this efficiently through cryptographic means and mechanism design. For example, our fair exchange protocol for peer-to-peer file sharing ensures that peers are fair to each other with respect to exchanging data. Our work on outsourced computation discusses how the boss can fairly reward honest contractors, as well as punish malicious ones limiting their damage to the system. Our outsourced storage solution makes sure that official

arbitration between the client and the server is fair in that the client cannot frame an honest server yet a malicious server will be caught cheating. We also considered other aspects of fairness such as trust and incentives and how that fits into the secure cloud paradigm. Finally, we showed how to render all these practical by developing a cryptographic library and language.

In the future, I am planning to extend my work on this broad definition of fairness into further areas of the cloud. For example, two-party computation, which is a very important cloud computing idea, suffers from lack of fairness: ensuring either both parties obtain the result of the computation or neither does is not a straightforward task. It requires novel fairness techniques, possibly similar to the fair exchange protocol we discovered. Furthermore, even when two peers exchange data fairly, they may end up exchanging data that is not considered fair; this may be due to one peer having some external knowledge about the system that provides him some advantage in trading a less valuable data in exchange for a more useful data. For example, in our peer-to-peer file sharing scenario, this may correspond to one peer trading a common data item in exchange for a rare one. Oblivious fair exchange techniques may result in both parties exchanging data that they want, but without low-level control on the data, possibly ensuring that peers exchange some random data they want from each other. Novel techniques are required in many cases including oblivious data access or private information retrieval in the cloud to be fair to the server in terms of efficiency and to the client in terms of privacy.

Another aspect of this thesis work that I would like to continue in the future is privacy. Our protocols are designed to respect privacy of the participants. Privacy is a big concern in the cloud, especially now that all the digital communication can easily be logged, backed up and mined. For example, even fairness might suffer in real life if the solution does not respect the privacy of the participants. When

the participants and the arbiter are not anonymous, bribery might be a natural disaster, and trust issues may arise. Receipt-freeness, meaning that a participant cannot prove his actions or messages, is a useful property in such a setting, just as in voting protocols. Furthermore, cloud may be utilized as a helper for privacy, considering techniques such as anonymous routing that anonymizes network packets sent between parties and joint coin flipping that lets others to contribute randomness.

For a privacy-preserving economy, e-cash offers a good solution. The economics of solutions using e-cash leads to interesting game-theoretic problems. For example, when analyzing the problem of bootstrapping new users, Sybil (multi-identity) attacks become an issue in the peer-to-peer world. Analyzing economies created by the use of e-cash in peer-to-peer file sharing protocols and outsourcing presents novel challenges. In our outsourced computation scenario, we assumed the boss can reward or fine the contractors. Whether or not the boss can keep a balance between fines and rewards is an open problem. If the boss is the e-cash bank, it is possible to create more money, but this may lead to devaluation. Future collaborations with economics researchers would be useful in understanding such effects.

I think any system needs to be analyzed as a whole. This is exactly what I did during my Ph.D.: coming up with a novel fair exchange protocol, and analyzing how possible bottlenecks can be avoided by outsourcing or distributing the work of the centralized, trusted components. In terms of practical use of the solutions, I always favor seeing them in action. Thus, we built a language that eases implementation of cryptographic protocols, and a library that is being used by our collaborators for systems research. In the future, I am planning to continue addressing different security aspects of the cloud as a whole. Apart from research, my side goal would be to ensure wide-spread and easy use of cryptography.

# Bibliography


[1] M. Abd-El-Malek, G.R. Ganger, G.R. Goodson, M.K. Reiter, and J.J. Wylie. Fault-scalable byzantine fault-tolerant services. In *SOSP*, 2005.

[2] A.S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J.P. Martin, and C. Porth. Bar fault tolerance for cooperative services. In *ACM SOSP*, 2005.

[3] A. Anagnostopoulos, M.T. Goodrich, and R. Tamassia. Persistent Authenticated Dictionaries and Their Applications. *ISC*, pages 379–393, 2001.

[4] N. Asokan, PA Janson, M. Steiner, and M. Waidner. The state of the art in electronic payment systems. *IEEE Computer*, 30(9):28–35, 1997.

[5] Nadarajah Asokan, Matthias Schunter, and Michael Waidner. Optimistic protocols for fair exchange. In *ACM CCS*, 1997.

[6] Nadarajah Asokan, Victor Shoup, and Michael Waidner. Optimistic fair exchange of digital signatures. In *EUROCRYPT*, 1998.

[7] Nadarajah Asokan, Victor Shoup, and Michael Waidner. Optimistic fair exchange of digital signatures. *IEEE Selected Areas in Communications*, 18:591–610, 2000.

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

[9] Giuseppe Ateniese, Randal Burns, Reza Curtmola, Joseph Herring, Lea Kissner, Zachary Peterson, and Dawn Song. Provable data possession at untrusted stores. In *ACM CCS*, 2007.

[10] Giuseppe Ateniese, Roberto Di Pietro, Luigi V. Mancini, and Gene Tsudik. Scalable and efficient provable data possession. In *SecureComm*, 2008. http://eprint.iacr.org/2008/114.

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

[12] M. Backes, A. Datta, A. Derek, J.C. Mitchell, and M. Turuani. Compositional analysis of contract-signing protocols. *Theoretical Computer Science*, 367(1-2):33–56, 2006.

[13] Endre Bangerter, Jan Camenisch, and Anna Lysyanskaya. A cryptographic framework for the controlled release of certified data. *SPW*, 2004.

[14] Feng Bao, Robert Deng, and Wenbo Mao. Efficient and practical fair exchange protocols with off-line TTP. In *IEEE Security and Privacy*, 1998.

[15] Mira Belenkiy, Melissa Chase, Chris Erway, John Jannotti, Alptekin Küpçü, and Anna Lysyanskaya. Incentivizing outsourced computation. In *NetEcon*, 2008. http://www.cs.brown.edu/research/brownie/ioc-netecon08.pdf.

[16] Mira Belenkiy, Melissa Chase, Chris Erway, John Jannotti, Alptekin Küpçü, Anna Lysyanskaya, and Eric Rachlin. Making p2p accountable without losing privacy. In *ACM WPES*, 2007. http://www.cs.brown.edu/research/brownie/p2p-ecash-wpes07.pdf.

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

[18] M. Bellare and P. Rogaway. Collision-resistant hashing: Towards making uowhfs practical. In *CRYPTO*, 1997.

[19] Mihir Bellare and Philip Rogaway. Random oracles are practical: a paradigm for designing efficient protocols. *ACM CCS*, pages 62–73, 1993.

[20] Mihir Bellare and Philip Rogaway. Optimal asymmetric encryption. In *EUROCRYPT*, 1994.

[21] A. Ben-David, N. Nisan, and B. Pinkas. Fairplaymp: a system for secure multiparty computation. In *ACM CCS*, 2008.

[22] Michael Ben-Or, Ran Canetti, and Oded Goldreich. Asynchronous secure computation. In *STOC*, pages 52–61, 1993.

[23] Michael Ben-Or, Oded Goldreich, Silvio Micali, and Ronald L. Rivest. A fair protocol for signing contracts. *IEEE Transactions on Information Theory*, 36(1):40–46, 1990.

[24] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *STOC*, pages 1–10, 1988.

[25] John Bethencourt and Brent Waters. Advanced crypto software collection. http://acsc.cs.utexas.edu.

[26] P. Bichsel, C. Binding, J. Camenisch, T. Gross, T. Heydt-Benjamin, D. Sommer, and G. Zaverucha. Cryptographic protocols of the identity mixer library. *IBM Research Report*, 2009.

[27] George Robert Blakley. Safeguarding cryptographic keys. In *NCC*, 1979.

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

[29] Boinc. `http://boinc.berkeley.edu`.

[30] D. Boneh and M. Naor. Timed commitments. In *CRYPTO*, 2000.

[31] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In *ASIACRYPT*, 2001.

[32] Fabrice Boudot. Efficient proofs that a committed number lies in an interval. *EUROCRYPT*, pages 431–444, 2000.

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

[34] Brownie points project at brown university. http://cs.brown.edu/research/brownie.

[35] J. Camenisch and E. Van Herreweghen. Design and implementation of the idemix anonymous credential system. In *ACM CCS*, 2002. http://www.zurich.ibm.com/ jca/papers/camvan02.pdf.

[36] Jan Camenisch, Rafik Chaabouni, and Abhi Shelat. Efficient protocols for set membership and range proofs. In *ASIACRYPT*, 2008.

[37] Jan Camenisch and Ivan Damg$r_a$rd. Verifiable encryption, group encryption, and their applications to group signatures and signature sharing schemes. In *ASIACRYPT*, 2000.

[38] Jan Camenisch and Jens Groth. Group signatures: Better efficiency and new theoretical aspects. *SCN*, 2004.

[39] Jan Camenisch, Susan Hohenberger, Markulf Kohlweiss, Anna Lysyanskaya, and Mira Meyerovich. How to win the clonewars: efficient periodic n-times anonymous authentication. In *ACM CCS*, 2006.

[40] Jan Camenisch, Susan Hohenberger, and Anna Lysysanskaya. Compact e-cash. In *EUROCRYPT*, 2005.

[41] Jan Camenisch, Susan Hohenberger, and Michael Ostergaard Pedersen. Batch verification of short signatures. In *EUROCRYPT*, 2007.

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

[43] Jan Camenisch and Anna Lysyanskaya. A signature scheme with efficient protocols. *SCN*, 2576:268–289, 2002.

[44] Jan Camenisch, Anna Lysyanskaya, and Mira Meyerovich. Endorsed e-cash. In *IEEE Security and Privacy*, 2007.

[45] Jan Camenisch and Victor Shoup. Practical verifiable encryption and decryption of discrete logarithms. In *CRYPTO*, 2003.

[46] Ran Canetti, Oded Goldreich, and Shai Halevi. The random oracle methodology, revisited. *Journal of ACM*, 51(4):557–594, 2004.

[47] Ran Canetti and Tal Rabin. Fast asynchronous byzantine agreement with optimal resilience. In *STOC*, pages 42–51, 1993.

[48] Roberto Tamassia Charalampos Papamanthou and Nikos Triandopoulos. Authenticated hash tables. In *CCS*, 2008.

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

[50] David Chaum. Blind signatures for untraceable payments. In *CRYPTO*, 1982.

[51] Dwaine E. Clarke, Srinivas Devadas, Marten van Dijk, Blaise Gassend, and G. Edward Suh. Incremental multiset hash functions and their application to memory integrity checking. In *ASIACRYPT*, pages 188–207, 2003.

[52] Bram Cohen. Incentives build robustness in bittorrent. In *WEPS*, 2003.

[53] Larry Cohen. Testimony of larry cohen, president of communications workers of america, May 2007.

[54] S. Coull, M. Green, and S. Hohenberger. Controlling access to an oblivious database using stateful anonymous credentials. In *PKC*, 2009.

[55] R. Cramer and I. Damgard. Zero-knowledge proofs for finite field arithmetic; or: Can zero-knowledge be for free? In *CRYPTO*, 1998.

[56] Ronald Cramer and Victor Shoup. A practical public key cryptosystem provably secure against adaptive chosen ciphertext attack. In *CRYPTO*, 1998.

[57] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES–the Advanced Encryption Standard*. Springer, 2002.

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

[59] Ivan Damgard. On sigma protocols. http://www.daimi.au.dk/ ivan/Sigma.pdf.

[60] Ivan Damgard. Efficient concurrent zero-knowledge in the auxiliary string model. *EUROCRYPT*, pages 418–430, 2000.

[61] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: the second-generation onion router. In *USENIX Security*, 2004.

[62] Distributed.net. `http://www.distributed.net`.

[63] Y. Dodis, P.J. Lee, and D.H. Yum. Optimistic fair exchange in a multi-user setting. *LNCS*, 4450:118, 2007.

[64] Y. Dodis and A. Yampolskiy. A verifiable random function with short proofs and keys. In *PKC*, 2005.

[65] Yevgeniy Dodis, Salil Vadhan, and Daniel Wichs. Proofs of retrievability via hardness amplification. In *TCC*, 2009.

[66] Danny Dolev, Cynthia Dwork, and Moni Naor. Nonmalleable cryptography. *SIAM Journal on Computing*, 30(2):391–437, April 2000.

[67] Cynthia Dwork, Moni Naor, Guy N. Rothblum, and Vinod Vaikuntanathan. How efficient can memory checking be?, 2009. TCC.

[68] Chris Erway, Theodora Hinkle, Alptekin Küpçü, Anna Lysyanskaya, and Sarah Meiklejohn. Zkpdl: Enabling efficient implementation of zero-knowledge proofs and electronic cash. In *USENIX Security*, 2010.

[69] Chris Erway, Alptekin Küpçü, Sarah Meiklejohn, Theodora Hinkle, John Jannotti, and Anna Lysyanskaya. Fairtrader: Fair, fungible file sharing. *under submission*, 2009.

[70] Chris Erway, Alptekin Küpçü, Charalampos Papamanthou, and Roberto Tamassia. Dynamic provable data possession. In *ACM CCS*, 2009. http://eprint.iacr.org/2008/432.

[71] Chris Erway, Alptekin Küpçü, Charalampos Papamanthou, and Roberto Tamassia. Publicly verifiable dynamic provable data possession. *work in progress*, 2009.

[72] P. Wyckoff F. Monrose and A. Rubin. Distributed execution with remote audit. In *NDSS*, 1999.

[73] Anna Lisa Ferrara, Matthew Green, Susan Hohenberger, and Michael Ostergaard Pedersen. Practical short signature batch verification. In *CT-RSA*, 2009.

[74] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. *CRYPTO*, 86:186–194, 1986.

[75] E. Fujisaki, T. Okamoto, D. Pointcheval, and J. Stern. Rsa-oaep is secure under the rsa assumption. *Journal of Cryptology*, 17(2):81–104, 2004.

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

[77] Eiichiro Fujisaki and Tatsuaki Okamoto. A practical and provably secure scheme for publicly verifiable secret sharing and its applications. In *EUROCRYPT*, volume 1403 of *LNCS*, pages 32–46, 1998.

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

[79] Décio Luiz Gazzoni and Paulo Sérgio Licciardi Messeder Barreto. Demonstrating data possession and uncheatable data transfer. Cryptology ePrint Archive, Report 2006/150, 2006.

[80] O. Goldreich, S. Goldwasser, and S. Micali. How to construct random functions. *Journal of ACM*, 33(4):807, 1986.

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

[82] 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.

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

[84] Shafi Goldwasser, Silvio Micali, and Ronald Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281–308, April 1988.

[85] P. Golle and I. Mironov. Uncheatable distributed computations. In *CT-RSA*, 2001.

[86] M. T. Goodrich, R. Tamassia, and A. Schwerin. Implementation of an authenticated dictionary with skip lists and commutative hashing. In *DISCEX II*, pages 68–82, 2001.

[87] Michael T. Goodrich, Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. Athos: Efficient authentication of outsourced file systems. In *ISC*, pages 80–96, 2008.

[88] Identity mixer project. http://idemix.wordpress.com.

[89] Ru Iosup, Pawe Garbacki, Johan Pouwelse, and Dick Epema. Correlating topology and path characteristics of overlay networks and the internet. In *GP2PC*, 2006.

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

[91] Burt Kaliski. Twirl and rsa key size. Technical report, RSA Laboratories, 2003. http://www.rsa.com/rsalabs/node.asp?id=2004.

[92] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable secure file sharing on untrusted storage. *FAST*, pages 29–42, 2003.

[93] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. Chapman & Hall/CRC, 2007.

[94] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishan Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. Oceanstore: an architecture for global-scale persistent storage. *SIGPLAN Not.*, 35(11):190–201, 2000.

[95] Alptekin Küpçü and Anna Lysyanskaya. Optimistic fair exchange with multiple arbiters. *PODC brief announcement, full version under submission*, 2009. http://eprint.iacr.org/2009/069.

[96] Alptekin Küpçü and Anna Lysyanskaya. Usable optimistic fair exchange. In *CT-RSA*, 2010. http://eprint.iacr.org/2008/431.

[97] RSA Laboratories. Faq: What key size should be used? http://www.rsa.com/rsalabs/node.asp?id=2264.

[98] K. Lai, L. Rasmusson, E. Adar, L. Zhang, and B.A. Huberman. Tycoon: An implementation of a distributed, market-based resource allocation system. *Multiagent and Grid Systems*, 1(3):169–182, 2005.

[99] Feifei Li, Marios Hadjieleftheriou, George Kollios, and Leonid Reyzin. Dynamic authenticated index structures for outsourced databases. In *SIGMOD*, pages 121–132, 2006.

[100] H.C. Li, A. Clement, E.L. Wong, J. Napper, I. Roy, L. Alvisi, and M. Dahlin. BAR Gossip. *Proceedings of the 7th Symposium on Operating System Design and Implementation (OSDI)*, 2006.

[101] J. Li, M. Krohn, D. Mazieres, and D. Shasha. Secure Untrusted Data Repository (SUNDR). *OSDI*, pages 121–136, 2004.

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

[103] H. Lipmaa. On diophantine complexity and statistical zero-knowledge arguments. *ASIACRYPT*, 2894:398–415, 2003.

[104] Ben Lynn. PBC (pairing-based cryptography) library. http://crypto.stanford.edu/pbc.

[105] Umesh Maheshwari, Radek Vingralek, and William Shapiro. How to build a trusted database system on untrusted storage. In *OSDI*, pages 10–26, Berkeley, CA, USA, 2000. USENIX Association.

[106] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplaya secure two-party computation system. In *USENIX Security*, 2004.

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

[108] A.J. Menezes, P.C. Van Oorschot, and S.A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

[109] R.C. Merkle. A digital signature based on a conventional encryption function. *LNCS*, 293:369–378, 1987.

[110] Silvio Micali. Simultaneous electronic transactions with visible trusted parties. US Patent 5,553,145, 1996.

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

[112] G.L. Miller. Riemann's hypothesis and tests for primality. In *STOC*, pages 234–239, 1975.

[113] S. Mitsunari, R. Sakai, and M. Kasahara. A new traitor tracing. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 85(2):481–484, 2002.

[114] David Molnar. The seti@home problem. *ACM Crossroads*, Sep 2000. http://www.acm.org/crossroads/columns/onpatrol/september2000.html.

[115] A. Muthitacharoen, R. Morris, T.M. Gil, and B. Chen. Ivy: A Read/Write Peer-to-Peer File System. *OSDI*, pages 31–44, 2002.

[116] Moni Naor and Kobbi Nissim. Certificate revocation and certificate update. In *USENIX Security*, pages 17–17, 1998.

[117] Moni Naor and Guy N. Rothblum. The complexity of online memory checking. In *FOCS*, pages 573–584, 2005.

[118] Moni Naor and Moti Yung. Universal one-way hash functions and their cryptographic applications. In *STOC*, 1989.

[119] NIST. Digital signature standard (dss). *FIPS PUB 186-2*, 2000.

[120] A. Oprea, M.K. Reiter, and K. Yang. Space-Efficient Block Storage Integrity. *NDSS*, 2005.

[121] John Ousterhout. Tcl/tk. http://www.tcl.tk/.

[122] 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.

[123] Pascal Paillier. Public-key cryptosystems based on composite residuosity classes. In *EUROCRYPT*, 1999.

[124] Charalampos Papamanthou and Roberto Tamassia. Time and space efficient algorithms for two-party authenticated data. In *ICICS*, 2007.

[125] Charalampos Papamanthou and Roberto Tamassia. Time and space efficient algorithms for two-party authenticated data structures. In *ICICS*, pages 1–15, 2007.

[126] Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. Authenticated hash tables. In *ACM CCS*, pages 437–448, 2008.

[127] Patch-free-processing. `http://web.archive.org/web/20070207064618/http://home.hccnet.nl/a.alfred/p-free-p1pfp.html`.

[128] Torben Pryds Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *CRYPTO*, 1991.

[129] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.

[130] M.O. Rabin and J.O. Shallit. Randomized algorithms in number theory. *Communications on Pure and Applied Mathematics*, 39:239–256, 1986.

[131] Anatol Rapoport. Prisoner's dilemma - recollections and observations. *Game Theory as a Theory of Conflict Resolution*, 1974.

[132] O. Regev and N. Nisan. The POPCORN market. Online markets for computational resources. *Decision Support Systems*, 28(1-2):177–189, 2000.

[133] Rosetta@home. `http://boinc.bakerlab.org/rosetta/`.

[134] Samba. Samba.org CVS repository. http://cvs.samba.org/cgi-bin/cvsweb/.

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

[136] T. Schwarz and E.L. Miller. Store, Forget, and Check: Using Algebraic Signatures to Check Remotely Administered Storage. *ICDCS*, page 12, 2006.

[137] F. Sebe, A. Martinez-Balleste, Y. Deswarte, J. Domingo-Ferre, and J.-J. Quisquater. Time-bounded remote file integrity checking. Technical Report 04429, LAAS, July 2004.

[138] Seti@home. `http://setiathome.berkeley.edu`.

[139] Hovav Shacham and Brent Waters. Compact proofs of retrievability. In *ASIACRYPT*, 2008.

[140] Adi Shamir. How to share a secret. *ACM Communications*, 22(11):612–613, November 1979.

[141] V. Shmatikov and J.C. Mitchell. Finite-state analysis of two contract signing protocols. *Theoretical Computer Science*, 283(2):419–450, 2002.

[142] Victor Shoup. Lower bounds for discrete logarithms and related problems. *LNCS*, 1233:256, 1997.

[143] Victor Shoup and Rosario Gennaro. Securing threshold cryptosystems against chosen ciphertext attack. In *EUROCRYPT*, 1998.

[144] Roberto Tamassia. Authenticated data structures. In *ESA*, pages 2–5, 2003.

[145] Roberto Tamassia and Nikos Triandopoulos. Computational bounds on hierarchical data processing with applications to information security. In *ICALP*, pages 153–165, 2005.

[146] truXoft Calibrating BOINC Core Client. `http://boinc.truxoft.com/core-cal.htm`.

[147] C.A. Waldspurger, T. Hogg, B.A. Huberman, J.O. Kephart, and W.S. Stornetta. Spawn: A Distributed Computational Economy. *IEEE Transactions on Software Engineering*, 18(2):103–117, 1992.

[148] Qian Wang, Cong Wang, Jin Li, Kui Ren, and Wenjing Lou. Enabling public verifiability and data dynamics for storage security. In *ESORICS*, 2009.

# Appendix A

# Algorithms Used

## A.1 Security Parameters

In this chapter, we will use a unified security parameter *sec*, which corresponds to the conjecture that any attack that breaks the system must perform roughly $2^{sec}$ steps. We also define the following security parameters for ease of notation:

*RSALength* defines the bit-length of an RSA modulus.

*stat* defines the statistical security parameter. That is to say, two statistically indistinguishable distributions will be $2^{-stat}$ close to each other.

*hashLength* defines the length of the output of a hash function used for collision resistance or as a random oracle.

*orderLength* defines the bit-length of the order of the prime-order group we will use. *primeLength* defines the bit-length of the modulus of $Z^*_{primeModulus}$. The prime-order group will be a subgroup of $Z^*_{primeModulus}$ of order *primeOrder*.

To get $sec = 80$, which is considered the bare minimum requirement for security, we need to have $RSALength = 1024$, $stat = sec = 80$, $hashLength = 2 * sec = 160$, $orderLength = 2 * sec = 160$, $primeLength = 1024$. We will use these numbers for performance evaluation. The reasoning for hash functions and prime order is

related to the birthday bound, and states the best known algorithms break them in $2^{hashLength/2}$ and $2^{orderLength/2}$ time respectively [108]. We will use SHA-1 as our hash function.

To get a longer-lasting security, we should use $sec = 128$, which corresponds to $RSALength = 2048$, $stat = sec = 128$, $hashLength = 2 * sec = 256$, $orderLength = 2 * sec = 256$, $primeLength = 1024$. For symmetric encryption, we should use AES-128 as the block cipher and SHA-256 as the hash function whenever necessary (note that the RSA numbers correspond to $sec = 112$ [91, 97]).

Finally, if we want a really paranoid level of security (e.g., for financial uses), we can set $RSALength = 4096$, $primeLength = 2048$, and use SHA-512 and AES-256. The only issue is that we will sacrifice some amount of efficiency.

## A.2 Assumptions

The suggested security parameters above are set to satisfy the following assumptions (by conjecture). Those assumptions are computational: they work against computationally bounded —probabilistic polynomial time (PPT)— adversaries. They are also probabilistic. A negligible probability denotes a probability that is a negligible function of the security parameter (e.g., the key-length of an encryption scheme). A negligible function of $n$ is a function which is smaller than any inverse polynomial over $n$ with $n > N$ for sufficiently large $N$ (e.g., $neg(n) = 2^{-n}$). A non-negligible probability is a probability that is not negligible.

Strong RSA Assumption (see [18]): Given an RSA modulus $n$, and a random value $x \in Z_n^*$, no PPT adversary can find $z > 1$ and $y \in Z_n^*$ such that $y^z = x \mod n$ with non-negligible probability.

Discrete Logarithm Assumption (see [142]): Given a group $G$ with modulus $n$, a random generator $g$, and a random element $x \in G$, no PPT adversary can find

a value $y$ such that $g^y = x \mod n$ with non-negligible probability.

## A.3 Setup

We will concentrate on two main types of groups that are widely-used: a special RSA group and a prime order group. These groups are usually generated by a trusted third party, but we will describe methods that can be used if for some reason the setup is untrusted.

Let $QR_n$ denote the quadratic residues modulo $n$.

**Algorithm A.3.1**: Setup for generating a special RSA group.

**Input**: security parameter *RSALength*, number of generators $m$

**Pre-conditions**: *RSALength* must be at least 1024. $m$ must be at least 1.

**Output**: modulus $n$, primes $p, q, p', q'$, generators $g_1, \ldots, g_m, h$, exponents $a_1, \ldots, a_m$

**Post-conditions**: The length of $n$ must be *RSALength*. $n = pq$, $p = 2p' + 1$, $q = 2q' + 1$. $p, q, p', q'$ are primes. $p = 3 \mod 4$, $q = 3 \mod 4$. $|p| = |q| = RSALength/2$

`Special RSA Group Setup`

1. Choose primes $p, q$ of length $RSALength/2$ each such that $p = 2p' + 1$ and $q = 2q' + 1$ where $p', q'$ are primes (so $p, q$ are safe primes, $p', q'$ are Sophie Germain primes, and $n$ is a special RSA modulus). Note that $p$ and $q$ need to be Blum integers (i.e. $p = 3 \mod 4$ and $q = 3 \mod 4$), and that this will indeed be the case for primes chosen as above.
2. Compute $n = pq$.
3. Choose $h \leftarrow QR_n$. To do this, pick a random residue, square it, and check that $h^{(p-1)/2} = 1 \mod p$ and $h^{(q-1)/2} = 1 \mod q$. Alternatively, pick $h_p \leftarrow Z_p^*$ and $h_q \leftarrow Z_q^*$, and set $h = h_p^{q-1} * h_q^{p-1} \mod n$.
4. **for** $i$ : $1..m$ **do**
5. Choose $a_i \leftarrow \{0,1\}^{RSALength+stat}$ and set $g_i = h^{a_i} \mod n$
6. Output modulus $n$, primes $p, q, p', q'$, generators $g_1, \ldots, g_m, h$, and exponents $a_1, \ldots, a_m$.

**Algorithm A.3.2**: Setup for generating a prime-order group.

**Input**: security parameters *primeLength*, *orderLength*, number of generators $m$

**Pre-conditions**: *primeLength* must be at least 512. *orderLength* must be at least 160 or $2 * stat$, whichever is larger.[a] $m$ must be at least 1.

**Output**: modulus *primeModulus*, order *primeOrder*, generators $g_1, \ldots, g_m, h$, exponents $a_1, \ldots, a_m$

**Post-conditions**: The length of *primeModulus* must be *primeLength*. The length of *primeOrder* must be *orderLength*.

`Prime Order Group Setup`

1 Pick a prime order *primeOrder* of length *orderLength* and a prime modulus *primeModulus* of length *primeLength* where *primeModulus* $- 1$ is divisible by *primeOrder*. This results in the order *primeOrder* subgroup of $Z^*_{primeModulus}$.

2 Pick generator $h$ for the subgroup with order *primeOrder*. To do this, pick $h' \leftarrow Z^*_{primeModulus}$ and set $h = h'^{(primeModulus-1)/primeOrder}$ mod *primeModulus*.

3 **for** $i$ *:* $1..m$ **do**

4 Pick generators $g_i$ each with order *primeOrder* (same method as above)

5 Output modulus *primeModulus*, order *primeOrder*, generators $g_1, \ldots, g_m, h$, exponents $a_1, \ldots, a_m$

[a] The best algorithms to solve Discrete Logarithm problem in prime-order groups have running time that depends on the size of either the order of the subgroup or the modulus of the group [142, 108]

Definitions of Groups:

Throughout this chapter, the "definition of the RSA group" means the modulus $n$ and possibly the bases $g_1, \ldots, g_m, h$ if they are not explicitly defined.

Similarly, when we use the "definition of the prime-order group", it means the modulus *primeModulus* and the order *primeOrder*, and possibly the bases $g_1, \ldots, g_m, h$ if they are not explicitly defined.

Bases will generally be explicitly defined in the algorithms.

Group Operations:

A group operation is an operation on a group element. We will use multiplicative notation when denoting the group operation and inverse of an element. All group operations must be done modulo the modulus of the group. Therefore, all group operations will be done modulo $n$ for the RSA group, and modulo *primeModulus* for the prime-order group. A simple example of this is $g * h \mod n$.

Operations on Exponents:

An operation on exponents means the multiplication, addition, or inversion of exponents. In RSA groups, all such operations must be performed over integers, as the order of the group is not assumed to be known. In prime-order groups, all such operations will be done modulo *primeOrder*, as this value is always public. An example of such an operation is $g^{a*b \mod primeOrder} \mod primeModulus$.

Group Secrets:

When we emphasize that some party has not generated a group itself, we mean that that party may not know the secrets associated with the group generation. For an RSA group, the secrets will be $p, q$; the factorization of the modulus $n$. Furthermore, in both an RSA group and a prime-order group, the relative discrete logarithms of the generators are secrets.

## A.4 Commitment Schemes

### A.4.1 Fujisaki-Okamoto Commitment Scheme

The Fujisaki-Okamoto commitment scheme [76, 58] is a statistically hiding, computationally binding commitment scheme. The Committer commits to something and sends the resulting commitment to the Verifier. At some later time, the Committer opens the commitment and the Verifier needs to verify that the opening matches the commitment sent before.

SETUP:

This commitment scheme uses a special RSA group. In the case that an untrusted party (e.g., the Verifier) generates the RSA group, he needs to prove to the Committer that each $g_i$ is in the group generated by $h$, so that the commitment is statistically hiding. This can be done by proving in zero knowledge the knowledge of $a_i$ such that $g_i = h^{a_i} \mod n$. The committer may not generate or know $p, q, p', q', a_1, \ldots, a_m$, as otherwise the scheme will not provide any meaningful binding property.

ASSUMPTIONS:

The security of the scheme relies on the Strong RSA assumption.

HIDING:

The hiding property relies on the fact that $g_i, h$ all generate the same group, so that when randomization is used the resulting commitment is a random group element.

BINDING:

The binding property relies on the assumption that the Committer cannot find two different openings (using the same bases) that result in the same commitment. This follows from the Strong RSA assumption.

WARNINGS:

This commitment scheme requires at least two generators: $g_1, h$, otherwise it does

not work.

**Algorithm A.4.1**: Commitment procedure of the Fujisaki-Okamoto commitment scheme. This procedure is run by the Committer.

**Input**: Definition of the RSA group, number of secrets $k$, bases $g_1, \ldots, g_k, h$, secrets $x_1, \ldots, x_k$

**Pre-conditions**: The RSA group must be generated by a trusted third party or it must be proven that each $g_i, h$ generates $QR_n$.

**Output**: commitment $C$, opening *open*

`Commit`

1 Pick a random number $r$ from $\{0,1\}^{RSALength+stat}$.

2 Create the commitment $C = h^r \prod_{i=1}^{k} g_i^{x_i} \mod n$.

3 Output commitment $C$ and opening $open = x_1, \ldots, x_k, r$.

COMMENTS:

The random number $r$ actually needs to be relatively prime to $\phi(n)$, but since this will be the case with high probability we omit the check.

To open a commitment, the Committer just sends the opening *open* to the Verifier. After that, the Verifier needs to verify the opening of the commitment. This is done as follows:

**Algorithm A.4.2**: Verification procedure of the Fujisaki-Okamoto commitment scheme. This procedure is run by the Verifier after receiving the opening *open* for a commitment $C$.

**Input**: Definition of the RSA group, number of secrets $k$, bases $g_1, \ldots, g_k, h$, commitment $C$, opening $open = x_1, \ldots, x_k, r$

**Pre-conditions**: The RSA group may NOT be generated by the Committer.

**Output**: accept or reject

```
Verify
1  if C = h^r ∏_{i=1}^{k} g_i^{x_i}  mod n then
2      Output accept
3  else
4      Output reject
```

## A.4.2 Pedersen Commitment Scheme

The Pedersen commitment scheme [128] is a statistically hiding, computationally binding commitment scheme. It allows for commitments to values between 1 and $primeOrder - 1$.

SETUP:

This commitment scheme uses a prime-order group. If an untrusted party (e.g., the Verifier) generates the prime-order group, then the participants (both the Committer and the Verifier) need to check that both *primeModulus* and *primeOrder* are primes (such that *primeOrder* divides $primeModulus - 1$) and that $g_i, h$ have order *primeOrder* (which is equivalent to saying that $g_i \neq 1 \mod primeModulus$ and $g_i^{primeOrder} = 1 \mod primeModulus$). It is important that the Committer does not know the relative discrete logarithms of the bases, or otherwise the commitment is

no longer binding.

ASSUMPTIONS:

The security of the scheme relies on the Discrete Logarithm assumption.

HIDING:

The hiding property relies on the fact that $g_i, h$ all generate the group with prime order *primeOrder*, so that when randomization is used the resulting commitment is a random element of the prime-order group.

BINDING:

The binding property relies on the Committer not being able to find two different openings, using the same bases, that result in the same commitment. This follows directly from the Discrete Logarithm assumption.

---

**Algorithm A.4.3**: Commitment procedure of the Pedersen commitment scheme. This procedure is run by the Committer.

---

**Input**: Definition of the prime-order group, number of secrets $k$, bases $g_1, \ldots, g_k, h$, secrets $x_1, \ldots, x_k$

**Pre-conditions**: Group and the bases must adhere to guidelines in the SETUP in this section. Each $x_i$ must be between 1 and $primeOrder - 1$.

**Output**: commitment $C$, opening *open*

**Post-conditions**: $C$ should be sent to the Verifier.

`Commit`

1. Pick a random number $r$ from $Z^*_{primeOrder}$.
2. Create the commitment $C = h^r \prod_{i=1}^{k} g_i^{x_i} \mod primeModulus$.
3. Output commitment $C$ and opening $open = x_1, \ldots, x_k, r$.

---

To open a commitment, the Committer simply sends *open* to the Verifier. Upon

receiving the opening, the Verifier needs to verify its validity. This is done as follows:

**Algorithm A.4.4**: Verification procedure of the Pedersen commitment scheme. This procedure is run by the Verifier after receiving the opening *open* for a commitment $C$.

**Input**: Definition of the prime-order group, number of secrets $k$, bases $g_1, \ldots, g_k, h$, commitment $C$, opening $open = x_1, \ldots, x_k, r$

**Pre-conditions**: The group may not be generated by the Committer. Each $x_i, r$ must be between 1 and $primeOrder - 1$.

**Output**: accept or reject

`Verify`

1 **if** $C = h^r \prod_{i=1}^{k} g_i^{x_i} \mod primeModulus$ **then**
2 Output accept
3 **else**
4 Output reject

COMMENTS:

If only one generator exists (namely $g_1$), then the commitment is only computationally hiding based on the Discrete Logarithm assumption, although it still works.

## A.5 Honest-Verifier Zero Knowledge Sigma Proofs

In these Σ-protocols (sigma proofs) [55, 59, 135], both the Prover and the Verifier know the definition of the group used, as well as some common information (typically a commitment) $C$. In all honest-verifier zero knowledge Σ-proofs, the Prover first sends a randomized proof $R$. The Verifier then replies with a challenge $c$ (which is always generated in the same way). The Prover then responds to that challenge with

some value $A$. Finally, the Verifier verifies the whole proof.

**Algorithm A.5.1**: Randomized Proof round of a $\Sigma$-protocol. This procedure is run by the Prover.

**Input**: Definition of the group, number of secrets $k$, bases $g_1, \ldots, g_k, h$,

**Output**: randomized proof $R$, its opening $openR$

`Randomize Proof`

1 Will be defined separately for each protocol.

**Algorithm A.5.2**: Challenge round of a 3-round $\Sigma$-protocol. This procedure is run by the Verifier upon receipt of randomized proof $R$ from the Prover.

**Input**: Definition of the group, number of secrets $k$, bases $g_1, \ldots, g_k, h$, commitment $C$, randomized proof $R$

**Pre-conditions**: In many protocols we use, the group may not be generated by the Prover. See the specific protocol specification for more on this.

**Output**: challenge $c$

**Post-conditions**: $C, R$ and $c$ should be kept for use in the verification procedure.

`Challenge`

1 Pick a random number $c$ from the domain of challenges $D_C$.

2 Output challenge $c$

Domain of Challenges $D_C$:

If an RSA group is used, the domain of challenges is $D_C = \{0,1\}^{stat}$ (with the exception that the challenge must not be 0). If a prime-order group is used, the domain of challenges is $D_C = Z^*_{primeOrder}$ (actually $D_C = \{0,1\}^{stat}$ will also work). When using non-interactive proofs, $D_C = \{0,1\}^{2*stat}$ will be used, which will be the same as $D_C = Z^*_{primeOrder}$. This means that in practice it will make sense to simply

use $D_C = \{0,1\}^{2*stat} - \{0\}$ for both groups.

Domain of Randomness $D_R$:

If an RSA group is used, the domain of randomness is $D_R = \{0,1\}^{RSALength+stat}$ (except again the randomness must not be 0). If a prime-order group is used, the domain of randomness is $D_R = Z^*_{primeOrder}$. Note that, unlike the challenge domains, these domains are very different for both groups, and that the prime-order group is more efficient in terms of both computation and communication.

Non-Interactive Version:

If non-interactive proofs will be employed using the Fiat-Shamir heuristic [74], the challenge will be computed as $c = hash$(definition of the group and bases used $||$ $k$ $||$ $C$ $||$ $R$). This computation can be carried out by the Prover and the Verifier separately, and then the Verifier can verify the proof as before. It is important to note that if the Fiat-Shamir heuristic is used, the resulting protocol is secure only in the Random Oracle model [19, 46].

Full Zero Knowledge:

Techniques for converting honest-verifier zero-knowledge proofs to full zero-knowledge proofs (i.e., using trapdoor commitments as in [60]) should be applied for the interactive versions. Non-interactive versions using the Fiat-Shamir heuristic do not require this conversion, since the Random Oracle model essentially forces the Verifier to be honest. Since all the proofs we will use are non-interactive, we do not provide full

zero-knowledge protocols here.

**Algorithm A.5.3**: Response round of a Σ-protocol. This procedure is run by the Prover.

**Input**: Definition of the group, number of secrets $k$, bases $g_1, \ldots, g_k, h$, opening of $C$: $openC$, randomized proof $R$ and its opening $openR$, challenge $c$

**Output**: response $A$

`Respond`

1 Will be defined separately for each protocol.

**Algorithm A.5.4**: Verification of a Σ-protocol. This procedure is run by the Verifier upon receipt of response $A$ from the Prover.

**Input**: Definition of the group, number of secrets $k$, bases $g_1, \ldots, g_k, h$, commitment $C$, randomized proof $R$, challenge $c$, response $A$

**Output**: accept or reject

`Verify`

1 Will be defined separately for each protocol.

Non-Interactive Verification:

When verifying non-interactive proofs, the verification procedure is called with challenge $c = hash$(definition of the group and bases used $||$ $k$ $||$ $C$ $||$ $R$), and the rest proceeds the same as the interactive version.

Randomization:

Here is a tiny procedure that we will refer to when we would like to create random

numbers.

**Algorithm A.5.5**: Procedure to generate a random group element with associated random exponents. Call this procedure **Randomize**.

**Input**: Definition of the group, number of random elements $k$, number of fixed elements $l$, bases $g_1, \ldots, g_{k+l-1}, h$, fixed elements $x_1, \ldots, x_l$ if any.

**Pre-conditions**: $l < k$, $k \geq 1$, $l \geq 0$

**Output**: random element $R$, random exponenets $openR$

`Randomize`

1. Pick $k-1$ random numbers $s_i$ from the domain of randomness $D_R$.
2. Pick another random number $t$ from the domain of randomness $D_R$.
3. Compute $R = [\prod_{i=1}^{l} g_i^{x_i}][\prod_{i=1}^{k-1} g_{l+i}^{s_i}]h^t$ using group operations.
4. Output the random group element $R$ and random exponents $openR = s_1, \ldots, s_{k-1}, t$.

### A.5.1 Proof of Knowledge of Discrete Logarithm Representation

This is the protocol used for proving knowledge of the discrete logarithm representation of a number using some well-defined bases in an honest verifier zero knowledge way [135]. The Verifier knows the number and the bases. We call this protocol **PoKoDLR**. It has two versions:

The RSA group version uses Fujisaki-Okamoto commitments. Therefore it requires the same setup and the same assumptions. Everything said for the Fujisaki-Okamoto commitments also applies here.

The prime-order group version uses Pedersen commitments. Therefore it requires the same setup and the same assumptions. Everything said for the Pedersen commitments also applies here.

| Version | Commitments Used | Assumption |
|---|---|---|
| RSA group | Fujisaki-Okamoto | Strong RSA |
| Prime-order group | Pedersen | Discrete Log |

Table A.1: PoKoDLR protocol security summary.

All operations will be done in respective groups.

ASSUMPTIONS:

RSA group version makes the Strong RSA assumption.

Prime-order group version makes the Discrete Logarithm assumption.

HONEST-VERIFIER ZERO KNOWLEDGE:

RSA group version is honest verifier zero knowledge provided that the Fujisaki-Okamoto commitment is hiding. This means that the RSA group must be generated by a trusted third party or it must be proven to the Prover that each $g_i, h$ generates $QR_n$.

Prime-order group version is honest verifier zero knowledge provided that the Pedersen commitment is hiding. This means that the prime-order group must be generated by a trusted third party or the Prover must verify that each $g_i, h$ has order $primeOrder$.

SOUNDNESS:

In RSA group version, the extraction works under the Strong RSA assumption, which requires that the RSA group may NOT be generated by the Prover.

In prime-order group version, the extraction works without any assumption.

**Algorithm A.5.6**: Randomized Proof round of PoKoDLR protocol. This procedure is run by the Prover.

**Input**: Definition of the group, number of secrets $k$, bases $g_1, \ldots, g_k, h$

**Pre-conditions**: The group must be generated by a trusted third party, or it must be proven to the Prover that each $g_i, h$ generates $QR_n$ in a special RSA group, or the Prover must verify that each $g_i, h$ has order *primeOrder* in a prime-order group.

**Output**: randomized proof $R$, its opening $openR$

`Randomize Proof`

1 **for** $i$ *:* $1..k$ **do**

2 Pick a random number $s_i$ from domain of randomness $D_R$.

3 Create a random element $R$ using Randomize(group definition, $k+1$, $g_1, \ldots, g_k, h$) as in Algorithm 16. Let the random exponents returned by the Randomize procedure be $openR = s_1, \ldots, s_k, t$.

4 Output randomized proof $R$ and its opening $openR$.

**Algorithm A.5.7**: Response round of PoKoDLR protocol. This procedure is run by the Prover.

**Input**: Definition of the group, number of secrets $k$, bases $g_1, \ldots, g_k, h$, opening $open = x_1, \ldots, x_k, r$, randomized proof $R$ and its opening $openR = s_1, \ldots, s_k, t$, challenge $c$

**Pre-conditions**: The group must be generated by a trusted third party, or it must be proven to the Prover that each $g_i, h$ generates $QR_n$ in a special RSA group, or the Prover must verify that each $g_i, h$ has order *primeOrder* in a prime-order group.

**Output**: response $A$

`Respond`

1 **for** $i$ *:* $1..k$ **do**

2 Compute $a_i = s_i + cx_i$

3 Compute $b = t + cr$

4 [ In prime-order group version, let $a_i = a_i \mod primeOrder$ and $b = b \mod primeOrder$ ]

5 Output response $A = a_1, \ldots, a_k, b$

**Algorithm A.5.8**: Verification of PoKoDLR protocol. This procedure is run by the Verifier upon receipt of response $A$ from the Prover.

**Input**: Definition of the group, number of secrets $k$, bases $g_1, \ldots, g_k, h$, commitment $C$, randomized proof $R$, challenge $c$, response $A = a_1, \ldots, a_k, b$

**Pre-conditions**: The RSA group may NOT be generated by the Prover

**Output**: accept or reject

`Verify`

1 **if** $RC^c = h^b \prod_{i=1}^{k} g_i^{a_i}$ *(* mod $n$ *for RSA group version,* mod *primeModulus for Prime-order group version)* **then**

2 Output accept

3 **else**

4 Output reject

WARNINGS:

In prime-order group version of this protocol, it's important for both parties to make sure that each $x_i, r, s_i, t$ is between 1 and $primeOrder - 1$. Prover can only prove knowledge of $x_i$ that is in that range.

### A.5.2 Proof of Equality of Discrete Logarithm Representation

This is the protocol for proving knowledge of the equality of discrete logarithm representations of some numbers using some well-defined bases in an honest verifier zero knowledge way. The Verifier knows the bases. We call this protocol **PoEoDLR**. it has two versions:

The first version uses Fujisaki-Okamoto commitments. Therefore it requires the

same setup and the same assumptions. Everything said for the Fujisaki-Okamoto commitments also applies there.

The second version uses Pederson commitments. Therefore it requires the same setup and the same assumptions. Everything said for the Pederson commitments also applies there.

All operations will be done in respective groups.

Assumptions:

RSA group version makes the Strong RSA assumption.

Prime-order group version makes the Discrete Logarithm assumption.

Honest-Verifier Zero Knowledge:

RSA group version is honest verifier zero knowledge provided that the Fujisaki-Okamoto commitment is hiding. This means that the RSA group must be generated by a trusted third party or it must be proven to the Prover that each $g_i, h$ generates $QR_n$.

Prime-order group version is honest verifier zero knowledge provided that the Pedersen commitment is hiding. This means that the prime-order group must be generated by a trusted third party or the Prover must verify that each $g_i, h$ has order $primeOrder$.

Soundness:

In RSA group version, the extraction works under the Strong RSA assumption, which requires that the RSA group may NOT be generated by the Prover.

In prime-order group version, the extraction works without any assumption.

**Algorithm A.5.9**: Randomized Proof round of PoEoDLR protocol. This procedure is run by the Prover.

**Input**: Definition of the group, bases $g_1, ..., g_k, h$, number of common secrets $k$, number other secrets $l$

**Output**: randomized proof set $R$, its opening set $openR$

`Randomize Proof`

1 **for** $i$ *:* $1..k$ **do**

2 Pick a random number $s_i$ from domain of randomness $D_R$.

3 **for** $i$ *:* $1..l$ **do**

4 Create the randomized proof $R_i$ to using Randomize(group definition, 1 (number of random elements), $k$ (number of fixed elements), $g_1, ...g_k, h$ (bases), $s_1, \ldots, s_k$ (fixed elements)). Let the opening returned by the Commit procedure be $openR_i = s_1, \ldots, s_k, t_i$.

5 Output randomized proof set $R$ (set of $R_i$'s) and opening set $openR$ (set of $openR_i$'s).

**Algorithm A.5.10**: Response round of PoEoDLR protocol. This procedure is run by the Prover.

**Input**: Definition of the group, f the group, bases $g_1, ..., g_k, h$, number of common secrets $k$, number other secrets $l$, common secrets $x_1, \ldots, x_k$ and other secrets $r_1, \ldots, r_l$, openings of randomized proofs $openR_1 \ldots openR_l$ where $openR_j = s_1, \ldots, s_k, t_j$, challenge $c$

**Output**: set of responses $A = a_1, \ldots, a_m$

`Respond`

1 **for** $i$ *:* $1..k$ **do**

2 Compute $a_i = s_i + cx_i$

3 **for** $j$ *:* $1..l$ **do**

4 Compute $b_j = t_j + cr_j$

5 If no RSA group is involved, let $a_i = a_i \mod primeModulus$ and $b_i = b_i \mod primeModulus$

6 Output response $A = a_1, \ldots, a_k, b_1, \ldots, b_l$

**Algorithm A.5.11**: Verification of PoEoDLR protocol. This procedure is run by the Verifier upon receipt of response $A$ from the Prover.

**Input**: Definition of the group, number of common secrets $k$, number of other secrets $l$, bases $g_1, \ldots, g_k, h$, set of commitments $C = C_1, \ldots C_l$, set of randomized proofs $R = R_1, \ldots R_l$, challenge $c$, response $A = a_1, \ldots, a_k, b_1, \ldots, b_l$

**Output**: accept or reject

`Verify`

1 **if** $R_1 C_1^c = h^{b_1} \prod_{i=1}^{k} g_i^{a_i}$ *AND* $\ldots$ *AND* $R_l C_l^c = h^{b_l} \prod_{i=1}^{k} g_i^{a_i}$ **then**

2 Output accept

3 **else**

4 Output reject

### A.5.3 Proof that a Committed Value $x$ is of the Form $x = y*z$

This is the protocol used for proving that a discrete logarithm representation is a product of two discrete logarithm representations in an honest verifier zero knowledge way [58]. In particular, this protocol can be used to prove that a committed number is a square, as in [32].

Suppose the Prover knows secrets $x, y, z$ such that $x = y * z$ and wants to prove this to an honest verifier. The Verifier knows commitments to each of these numbers: $C_x$ is a commitment to $x$, $C_y$ is to $y$ and $C_z$ is to $z$. We call this protocol **Mult**. It uses Fujisaki-Okamoto commitments. Therefore it requires the same setup and the same assumptions. Everything said for the Fujisaki-Okamoto commitments also applies here.

Assumptions:

The security of the scheme relies on Strong RSA assumption.

HONEST-VERIFIER ZERO KNOWLEDGE:

The protocol is honest verifier zero knowledge provided that the Fujisaki-Okamoto commitment is hiding. This means that the RSA group must be generated by a trusted third party or it must be proven to the Prover that each $g_i, h$ generates $QR_n$.

SOUNDNESS:

The extraction works under the Strong RSA assumption, which requires that the RSA group may NOT be generated by the Prover.

---

**Algorithm A.5.12**: Randomized Proof round of a Mult protocol for $x = yz$. This procedure is run by the Prover.

---

**Input**: Definition of the RSA group and bases: $n, g_1, h$

**Output**: randomized proof $R$, its opening $openR$

`Randomize Proof`

1. Pick a random number $s$ from $D_R$.
2. Create a random element $R_1$ using Randomize(group definition, 1, 1, $g_1, h$, $s$) as in Algorithm 16 where $s$ is a fixed element. Let the random exponent returned by the Randomize procedure be $openR_1 = t_1$.
3. Create a random element $R_2$ using Randomize(group definition, 1, 1, $C_y, h$, $s$) as in Algorithm 16 where $s$ is again a fixed element. Note the use of $C_y$ as one of the bases in the procedure (it will be used with the fixed element). Let the random exponent returned by the Randomize procedure be $openR_2 = t_2$.
4. Output randomized proof $R = R_1, R_2$ and opening $openR = s, t_1, t_2$.

---

**Algorithm A.5.13**: Response round of a Mult protocol for $x = yz$. This procedure is run by the Prover.

**Input**: Definition of the RSA group and bases: $n, g_1, h$, openings to commitments $C_x, C_y, C_z$ as $openX = x, r_x$, $openY = y, r_y$, $openZ = z, r_z$, randomized proof $R = R_1, R_2$ and its opening $openR = s, t_1, t_2$, challenge $c$

**Output**: response $A$

`Respond`

1 Compute $a = s + cz$, $b_1 = t_1 + cr_z$, $b_2 = t_2 + c(r_x - zr_y)$

2 Output response $A = a, b_1, b_2$

**Algorithm A.5.14**: Verification of a Mult protocol for $x = yz$. This procedure is run by the Verifier upon receipt of response $A$ from the Prover.

**Input**: Definition of the RSA group and bases: $n, g_1, h$, commitments $C_x, C_y, C_z$, randomized proof $R = R_1, R_2$, challenge $c$, response $A = a, b_1, b_2$

**Output**: accept or reject

`Verify`

1 **if** $R_1 C_z^c = g_1^a h^{b_1} \mod n$ AND $R_2 C_x^c = C_y^a h^{b_2} \mod n$ **then**

2 Output accept

3 **else**

4 Output reject

PROVING KNOWLEDGE OF $x, y$ SUCH THAT $x = y^2$:

Using the proof above, it is very easy to prove knowledge of $x, y$ such that $x = y^2$. Just set $z = y$, which also means getting rid of $C_z$, or in other words, setting $C_z = C_y$ and hence $openC_z = openC_y$.

### A.5.4 Proof that a Committed Value $x$ is Non-Negative $x \geq 0$

We would like to prove that a secret value $x$ is greater than or equal to 0 [32, 103]. The protocol uses a special RSA group. The verifier knows a commitment $C_x$ to $x$, along with the RSA group definition. The prover additionally knows the secret $x$.

As shown by Lagrange, any non-negative integer can be represented as a sum of 4 squares [130, 103] (e.g., $x = v_1^2 + v_2^2 + v_3^2 + v_4^2$), whereas negative integers cannot (obviously, square numbers must sum up to a non-negative number). These numbers $(v_i)$ can be computed efficiently [103, 130]. Let $W_i$ be $v_i^2$.

We have seen that using the Mult algorithm, we can prove that a committed number is actually a square. Therefore, the basic idea is to prove that each $W_i$ is a square (by committing to them so that the verifier does not learn $W_i$ or $v_i$), and then their sum is equal to $x$.

We call this protocol **Non-Negative**. It uses Fujisaki-Okamoto commitments. Therefore it requires the same setup and the same assumptions. Everything said for the Fujisaki-Okamoto commitments also applies here.

Assumptions:

The security of the scheme relies on Strong RSA assumption.

Honest-Verifier Zero Knowledge:

The protocol is honest verifier zero knowledge provided that the Fujisaki-Okamoto commitment is hiding. This means that the RSA group must be generated by a trusted third party or it must be proven to the Prover that each $g_i, h$ generates $QR_n$.

Soundness:

The extraction works under the Strong RSA assumption, which requires that the RSA group may NOT be generated by the Prover.

Below we provide the pseudocode for proving that a given value $y$ is non-negative

($y \geq 0$). We call this protocol **Non-Negative**.

In the Non-Negative protocol, the Prover is given a commitment $C_y$ and its opening $open_y = y, r_y$, and the group definition. The Verifier is given $C_y$ and the group definition.

1. The Prover computes $v_1, v_2, v_3, v_4$ using the four-squares algorithm [103, 130]. Let $W_i = v_i^2$.

2. The Prover computes commitments $C_i$ to $W_i$ with their openings $open_i = W_i, r_i$, with the only rule that the last randomness is set to be $r_4 = r_y - (r_1 + r_2 + r_3)$.

3. The Prover proves that each $C_i$ is a commitment to a square, using the special case of the Mult protocol.

4. The Verifier, in addition to checking that each $C_i$ is a commitment to a square, checks if $C_y = \Pi_{i=1}^{4} C_i$. If all checks pass, the Verifier accepts, otherwise he rejects.

### A.5.5 Proof that a Committed Value $x$ lies within an Interval $[lo, hi]$

We would like to prove that a secret value $x$ lies within a publicly known interval $[lo, hi]$, as in [32, 103, 40]. The protocol uses a special RSA group. The verifier knows a commitment $C_x$ to $x$, and $lo, hi$, along with the RSA group definition. The prover additionally knows the secret $x$.

Observe that $x$ is in the interval $[lo, hi]$ iff $(x - lo) * (hi - x) \geq 0$. This is true for any proper range ($hi \geq lo$), since no value $x$ can be smaller than $lo$ and greater than $hi$ at the same time. Further observe that a commitment to $x - lo$ can easily be obtained by computing $D = C_x g_1^{-lo}$, and a commitment to $hi - x$ can easily

be obtained by computing $E = g_1^{hi} C_x^{-1}$. Once the verifier computes $D, E$ himself, notice that the prover can use the Mult algorithm to prove that commitment $C_y$ to $y = (x - lo) * (hi - x)$ is the multiplication of $x - lo$ and $hi - x$ committed as $D, E$. Then, the prover can prove that $y \geq 0$. We call this protocol **Range**.

The Verifier knows a commitment $C_x$ to $x$, the interval (meaning $lo, hi$), and the RSA group definition, and can compute $D$ and $E$. The Prover, in addition to these, also knows the opening $open_x = x, r_x$ to $C_x$.

1. The Prover sets $y = (x - lo) * (hi - x)$ and computes $C_y = g^y h^r$ and proves that $C_y$ contains multiplication of the values in $D$ and $E$ using the Mult protocol.

2. The Prover runs the Non-Negative protocol for $y$ using $C_y$ as the commitment.

## A.6 CL Signatures

Camenisch-Lysyanskaya (CL) signatures [42, 43, 38, 13] is an example of blind signatures, where the signer may not know the values she signed. Besides, useful protocols such as proving existence of a CL signature on a secret value is also necessary. The version of CL signatures we will present here uses special RSA groups.

SETUP:

The key generation of CL signatures is exactly as generating a special RSA group as in Algorithm 6:

- We have the following generators: $f, g_1, \ldots, g_m, h$, and the key owner (signer) needs to provide a non-interactive proof that all these generators generate the same group (namely, $QR_n$). This can be done by proving the knowledge of $a_i$ such that $g_i = h^{a_i} \mod n$ and $a$ such that $f = h^a \mod n$.

- The public key (i.e., verification key) is $CLPK = n, f, g_1, \ldots, g_m, h$.

- The secret key (i.e., signing key) is $CLSK = p, q$.

ASSUMPTIONS:

The security of the scheme relies on Strong RSA assumption.

PARAMETERS:

This scheme uses the following (security) parameters: *RSALength* for the length of the RSA modulus, $m$ for the number of bases (maximum number of messages that can be signed at once), $l_x$ for the length of a message and $D_x$ for the domain of messages (each message $x_i$ is in $D_x = \{0,1\}^{l_x}$). We can also use messages in the domain $D_x = [-(2^{l_x}-1), 2^{l_x}-1]$. Further define $l_e = l_x + 2$ and $l_v = RSALength + l_x + 2 * stat$.

**Algorithm A.6.1**: Signing procedure for a CL signature. This is the signing procedure to sign a public message (not a blind signature yet). This procedure is run by the Signer.

**Input**: CL signature public key $CLPK = n, f, g_1, \ldots, g_m, h$, secret key $CLSK = p, q$, messages to be signed $x_1, \ldots, x_k$

**Pre-conditions**: Each message $x_i$ needs to be in $D_x$.

**Output**: signature $\sigma = A, e, v$

`Sign`

1. Pick a random **prime** number $e$ of length $l_e$.
2. Pick a random number $v$ of length $l_v$.
3. Compute the value $A$ such that $A^e = fh^v \Pi_{i=1}^k g_i^{x_i} \mod n$. This can be done with the knowledge of the $CLSK = p, q$ by setting $A = [fh^v \Pi_{i=1}^k g_i^{x_i}]^{1/e}$
4. Output the signature $A, e, v$.

**Algorithm A.6.2**: Verification procedure for a CL signature. This is the verification procedure to verify a public message (not a blind signature yet). This procedure is run by the Verifier.

**Input**: CL signature public key $CLPK = n, f, g_1, \ldots, g_m, h$, messages $x_1, \ldots, x_k$, signature $\sigma = A, e, v$

**Pre-conditions**: Each message $x_i$ needs to be in $D_x$. $e$ needs to be of length $l_e$, $v$ needs to be of length $l_v$.

**Output**: accept or reject

`Verify`

1 Check the pre-conditions (Length checks are important !!).
2 **if** $A^e = fh^v \Pi_{i=1}^{k} g_i^{x_i} \mod n$ **then**
3 Output accept
4 **else**
5 Output reject

### A.6.1 Obtaining a Blind CL Signature

Here, we present the protocol to obtain the CL signature on messages that are committed by the Recipient. Without loss of generality, let the first $l$ out of $k$ messages be the committed messages, and the rest be public messages.

Hence, both the Recipient and the Issuer knows the Fujisaki-Okamoto commitments $C_1, \ldots, C_l$ to messages $x_1, \ldots, x_l$ (note that another version can just use one commitment to all $x_1, \ldots, x_l$ under different bases. The extension is very straightforward, and hence not shown). The Issuer knows the public (or issuer-chosen) messages

$x_{l+1}, \ldots, x_k$. The Recipient knows all the messages $x_1, \ldots, x_k$.

**Algorithm A.6.3**: This procedure is run by the Recipient of a blind CL signature to initiate the signature issuing process.

**Input**: CL signature public key $CLPK = n, f, g_1, \ldots, g_m, h$, messages $x_1, \ldots, x_k$, commitments $C_1, \ldots, C_l$ and their openings $openC_1, \ldots, openC_l$ which include $r_1, \ldots, r_l$ (if there was only one commitment to those messages, there's just on $r$)

**Pre-conditions**: Each message $x_i$ needs to be in $D_x$.

`Receive`

1. Choose a random $v'$ of length $RSALength + stat$.
2. Compute $C = h^{v'} \Pi_{i=1}^{l} g_i^{x_i}$.
3. Prove using PoEoDLR protocol that the discrete logarithm representation of each secret $x_i$ in $C$ corresponds to those in $C_1, \ldots, C_l$ and also prove that each secret $x_i$ is in $D_x$ (note that $D_x = [-(2^{l_x} - 1), 2^{l_x} - 1]$ and we use a Range proof).

**Algorithm A.6.4**: This procedure is run by the Issuer of a blind CL signature to issue a blind CL signature.

**Input**: CL signature public key $CLPK = n, f, g_1, \ldots, g_m, h$, secret key $CLSK = p, q$, messages $x_{l+1}, \ldots, x_k$, commitments $C_1, \ldots, C_l$

**Pre-conditions**: Each message $x_i$ needs to be in $D_x$.

**Output**: partial signature $\sigma' = A, e, v''$

`Issue`

1. Pick a random **prime** number $e$ of length $l_e$.
2. Choose a random $v''$ of length $RSALength + l_x + stat$.
3. Compute $A = [fCh^{v''}\Pi_{i=l+1}^{k} g_i^{x_i})]^{1/e}$. Let us explain this step in detail. The given commitment is re-randomized using $v''$. Then, public messages are added to the signature in the $\Pi$ clause. Finally, the signature is computed.
4. Send $A, e, v''$ to the Recipient and prove using PoKoDLR protocol the knowledge of $1/e$ in the equation above.

**Algorithm A.6.5**: This procedure is run by the Recipient of a blind CL signature to construct a CL signature upon receipt of the partial signature.

**Input**: CL signature public key $CLPK = n, f, g_1, \ldots, g_m, h$, partial signature $\sigma' = A, e, v''$

**Pre-conditions**: Each message $x_i$ needs to be in $D_x$. $e$ must be a prime of length $l_e$, $v''$ needs to be of length $l_v$.

**Output**: signature $\sigma = A, e, v$

`Construct`

1. Check the range (and optionally primality) for $e$
2. Set $v = v' + v''$
3. Output signature $\sigma = \{A, e, v\}$

### A.6.2 Proving a CL Signature

Now that the Recipient has a blind CL signature, he wants to prove that fact to a verifier, without revealing the signature. Without loss of generality, let the first $l$ out of $k$ messages be the committed messages, and the rest be public messages.

Hence, both the Recipient and the Verifier knows the Fujisaki-Okamoto commitments $C_1, \ldots, C_l$ to messages $x_1, \ldots, x_l$ (as before, there can be just one commitment to all those messages). The Verifier knows the public (or issuer-chosen) messages $x_{l+1}, \ldots, x_k$. The Recipient knows all the messages $x_1, \ldots, x_k$ and the signature $\sigma = A, e, v$. Of course, both parties know the CL signature public key $CLPK = n, f, g_1, \ldots, g_m, h$.

- The Recipient choses a random number $r$ from $\{0,1\}^{RSALength+stat}$.
- The Recipient computes $A' = Ah^r$ and sends $A'$ to the Verifier. Set $v' = v + r * e$.
- The Verifier and the Recipient both separately computes the public value $D = \Pi_{i=l+1}^{k} g_i^{x_i}$ using public messages $x_{l+1}, \ldots, x_k$.
- The Recipient computes the commitment $C = h^{r_C} \Pi_{i=1}^{l} g_i^{x_i}$ for secret messages (using a random $r_C$ from $\{0,1\}^{RSALength+stat}$), and proves to the Verifier using PoEoDLR protocol that the discrete logarithm representation of each secret $x_i$ in $C$ corresponds to those in $C_1, \ldots, C_l$ and also proves that each secret $x_i$ is in $D_x$ (note that $D_x = [-(2^{l_x} - 1), 2^{l_x} - 1]$, and we use a Range proof), and of course the knowledge of $r_C$.
- Lastly, the Recipient proves using PoKoDLR protocol the knowledge of $e, v'$ such that $A'^e h^{r_C} = f h^{v'} CD$ (actually, prove that $fCD = A'^e * h^{(r_C - v')}$ or equivalently $fCD = A'^e * (1/h)^{v' - r_C}$).

## A.7 E-cash

In this section, we will provide the most efficient e-cash construction until now. We will be using the regular (non-bilinear) and offline versions of Compact E-Cash [40] and Endorsed E-Cash [44]. We require that all connections between a user and the Bank must be over an authenticated and secure channel (i.e., an SSL connection using the Bank's certificate).

### A.7.1 Compact E-cash

Compact e-cash [40] enables a user to withdraw a wallet containing many coins at once. But, the coins need to be spent one by one. Some extensions addressing this issue will be discussed later. Using offline compact e-cash, the Bank can find the public keys of double-spenders. Double deposits can easily be detected using serial numbers. The use of CL signatures assure that the serial numbers are not known to the Bank during the withdrawal process, and hence the anonymity of an honest user is guaranteed.

Compact e-cash works with a prime-order group, and utilizes CL signatures, so uses a special RSA group for the purposes of CL signatures. It's important that the $orderLength \leq l_x$, the CL signature message length.

Below, we will present a slightly modified version of compact e-cash. The modifications keep the scheme secure, while improving its efficiency greatly.

Assumptions:

Compact e-cash works in the Random Oracle model and makes Discrete Logarithm and Strong RSA assumptions, and also the following assumptions:

q-Decisional Diffie-Hellman Inversion Assumption (see [40, 64]): Given a group $G$ with modulus $n$ and prime order $p$, a random generator $g$, and a q-tuple $(g^x, \ldots, g^{x^q})$ for a random $x \in Z_p$, and a value $R$, no PPT adversary can decide

whether or not $R = g^{1/x} \mod n$ with non-negligible probability.

### Register

Every user must register with the Bank her public key. This is a very simple protocol, where the User just picks a random secret key $1 < sk_u < primeOrder$ and sends her public key $pk_u = g^{sk_u} \mod primeModulus$ to the Bank. The Bank registers the User's public key in a database, and associates it to an account.

Later on, whenever a user contacts the Bank and needs to prove her identity, the User proves knowledge of $sk_u$ that corresponds to $pk_u$ using the PoKoDLR protocol using only one base and Pedersen commitments.

### Withdraw

To withdraw a coin, a user contacts the Bank. Before this, the User must have been registered with the Bank. First, the User proves her identity to the Bank. Then, the

User and the Bank execute the following randomization procedure:

**Algorithm A.7.1**: Randomization procedure between the User and the Bank. This procedure needs to be executed for each wallet, before the withdrawal.

**Input**: Both parties know the definition of the prime-order group. The User knows his secret key $sk_u$ registered with the bank.

**Pre-conditions**: The group must be generated by a trusted third party, or the Prover must verify that each $g_i, h$ has order *primeOrder* in a prime-order group.

**Output**: The User's output is $s, t, A$, the Bank's output is $A$.

`Randomize Together`

1. The User picks secrets $s'$ and $t$ from $Z^*_{primeOrder}$.
2. The User creates a Pedersen commitment to $sk_u, s', t$. Call this commitment $A'$. The User sends $A'$ to the Bank.
3. The Bank picks a random number $r'$ from $Z^*_{primeOrder}$. The Bank sends $r'$ to the User.
4. The User sets $s = s' + r'$. The User and the Bank independently compute $A = A' * g^{r'}$.
5. The User's output is $s, t, A$, the Bank's output is $A$.

After the Randomize Together protocol, the User gets a blind CL signature from the Bank. The User starts with the commitment $A$, which he needs to prove that the secrets in $A$ correspond to the the secrets he committed to in the CL signature protocol, and the first secret corresponds to his secret key. Furthermore, the User also sends the Bank a wallet size $W$, which must be picked from a choice of wallet sizes (e.g., 1,10,100,1000,10000) and must be at most the User's current account balance. After checking the User's balance, the Bank signs $sk_u, s, t, W$ using blind CL signatures (where $W$ is public input), and decrements the User's account balance. At the end

of the Withdraw protocol, the User's wallet is composed of $sk_u, s, t, W, \sigma(sk_u, s, t, W)$ where $\sigma(sk_u, s, t, W)$ denotes the Bank's CL signature on $sk_u, s, t, W$, and a data structure to keep track of spent and remaining coins in the wallet (even though this can be a simple counter, we would like to spend coins with random indices, and so we need to keep a shuffled list of coin indices).

**Spend**

This is the protocol between a user and a merchant. To prevent man-in-the-middle attacks, before the exchange of the money, the User and the Merchant perform a secure key exchange protocol without setup. This can be a simple Diffie-Hellman key exchange over an RSA group using fresh randomly generated keys for both parties. Let the session secret derived from the key exchange be $ses = hash(\text{session key})$.

At the beginning, the Merchant picks a random $\mathsf{info}$ and sends this to the User. Both the Merchant and the User computes $R = hash(ses, \mathsf{info})$. Note that only the User and the Merchant can compute this value. Furthermore, the Merchant needs to use a different $\mathsf{info}$ for every transaction. This is to prevent man-in-the-middle attacks.

Next, the User picks the next unused coin index (i.e., the next random index in

the shuffled list of coin indices). Call this index $J$.

**Algorithm A.7.2**: Spend-Earn procedure between the User and the Merchant for Compact E-cash.

**Input**: Both parties know the definition of the prime-order group, and the Bank's CL signature public key $CLPK$. The User knows her wallet $sk_u, s, t, W, \sigma(sk_u, s, t, W)$, and the index $J$. Both parties know $R = hash(\mathsf{info}, ses)$.

**Pre-conditions**: The User must have withdrawn a wallet from the Bank, and the wallet must contain an unused coin. Furthermore, $R$ must be computed as directed above.

**Output**: The Merchant outputs $coin = B, C, D, W, S, T, J, \pi_{CL}, \pi_{ST}, \mathsf{info}, ses$.

`Spend Compact E-cash`

1. The User creates Pedersen commitments $B$ to $sk_u$, $C$ to $s$, and $D$ to $t$. The User prepares a non-interactive proof of knowledge of a CL signature on these values, where $W$ is a public value. Call this proof $\pi_{CL}$.
2. The User computes $S = g^{1/(s+J)}$ and $T = pk_u * g^{R/(t+J)}$.
3. The User prepares a non-interactive proof that $S, T$ are formed correctly. This is done by proving the knowledge of $s, t, sk_u, r_B, \alpha, r_1, \beta, r_2$ such that $B = g^{sk_u} * h^{r_B}$, $g = (g^J * C)^\alpha h^{r_1}$ (the Prover knows $r_1 = -r_C/(s+J)$ mod $primeOrder$), $g = (g^J * D)^\beta h^{r_2}$ (the Prover knows $r_2 = -r_D/(t+J)$ mod $primeOrder$), $S = g^\alpha$, $T = g^{sk_u} * (g^R)^\beta$. Call this proof $\pi_{ST}$.
4. The User sends $B, C, D, W, S, T, J, \pi_{CL}, \pi_{ST}$ to the Merchant.
5. The Merchant checks if the coin index is correct: $0 \leq J < W$, the proof of knowledge of the CL signature $\pi_{CL}$ verifies, and $S, T$ are correct under the proof $\pi_{ST}$. If all these are satisfied, the Merchant stores $coin = B, C, D, W, S, T, J, \pi_{CL}, \pi_{ST}, \mathsf{info}, ses$.

### Deposit

Depositing the coin is very easy. The Merchant contacts the Bank over a secure and one-way authenticated channel (i.e., SSL), proves his identity to the Bank, and then sends the coin to the Bank ($coin = B, C, D, W, S, T, J, \pi_{CL}, \pi_{ST}, \mathsf{info}, ses$). The Bank performs the exact check on line 5 of the Spend protocol in Algorithm 32. This verifies that the coin is formed correctly.

The bank then performs double-spending checks. First, the bank checks if $S$ appears in the list of deposited coins. If it does not exist in the deposited coins database, the Bank credits the Merchant's account, and adds the coin to the database. If $S$ indeed does exist, it exists together with an $R'$. If $R = R'$, then the Merchant is trying to deposit twice, and thus the Bank rejects the deposit. Otherwise, $R \neq R'$, and so the User double-spent the coin. The Bank runs the following Identify Double-Spender algorithm to find the public key (and hence the account) of the double-spender. An appropriate punishment can be performed afterwards.

**Algorithm A.7.3**: Identify Double-Spender procedure run by the Bank.

**Input**: Two coins $coin_1, coin_2$ with the same serial number $S$ but different $R_1, T_1$ and $R_2, T_2$.

**Output**: the public key of the double-spender $pk_u$, and proof of double-spending $\pi_{DS} = coin_1, coin_2$.

`Identify Double-Spender`

1 Verify the coins, if not already verified.

2 The Bank computes $pk_u = (T_2^{R_1}/T_1^{R_2})^{(R_1-R_2)^{-1}} \mod primeModulus$.

3 The Bank outputs the public key of the double-spender $pk_u$, and proof of double-spending $\pi_{DS} = coin_1, coin_2$.

Given the proof of double-spending, anyone can run the Identify Double-Spender protocol and be assured that the user with public key $pk_u$ has really double-spent.

This is because the coins require the knowledge of the secret key of the user.

### A.7.2 Endorsed E-cash

The Withdraw, Deposit, and Identify Double-Spenders protocols of endorsed e-cash [44] are the same as the compact e-cash. It makes different assumptions, though.

ASSUMPTIONS:

Endorsed e-cash works in the Random Oracle model and makes Discrete Logarithm and Strong RSA assumptions, and also the following assumptions:

COMPUTATIONAL DIFFIE-HELLMAN ASSUMPTION: Given a group $G$ with modulus $n$ and prime order $p$, a random generator $g$, and two values $g^x \mod n$ and $g^y \mod n$ for two random values $x, y \in Z_p$, no PPT adversary can compute $g^{xy} \mod n$ with non-negligible probability.

Q-DIFFIE-HELLMAN INVERSION ASSUMPTION (see [113]): Given a group $G$ with modulus $n$ and prime order $p$, a random generator $g$, and values $(g, g^x, \ldots, g^{x^q})$ for a random $x \in Z_p$, no PPT adversary can compute $g^{1/x} \mod n$ with non-negligible probability.

#### Spend

The only different part of the endorsed e-cash is how to spend a coin. The spend algorithm below is run by the user, who, on input a valid set of values for $(\mathsf{info}, ses)$,

outputs the unendorsed coin and its endorsment.

**Algorithm A.7.4**: The algorithm whereby a user generates an unendorsed coin together with a valid endorsement

**Input**: The User, as well as the Merchant who will ultimately receive the coin, know the definition of the prime-order group, and the Bank's CL signature public key $CLPK$. The User knows her wallet $sk_u, s, t, W, \sigma(sk_u, s, t, W)$, and the index $J$. Both parties know $R = hash(\mathsf{info}, ses)$.

**Pre-conditions**: The User must have withdrawn a wallet from the Bank, and the wallet must contain an unused coin. Furthermore, $R$ must be computed as directed above.

**Output**: The unendorsed coin is $uecoin = (B, C, D, W, S, T, J, \pi_{CL}, \pi_{ST})$. The corresponding endorsement is $endorsement = (x_1, x_2, r_y)$

`Spend Endorsed E-cash`

1 The User creates Pedersen commitments $B$ to $sk_u$, $C$ to $s$, and $D$ to $t$. The User prepares a non-interactive proof of knowledge of a CL signature on these values, where $W$ is a public value. Call this proof $\pi_{CL}$.

2 The User picks random $x_1, x_2, r_y$ from $Z^*_{primeOrder}$. The User sets $y = g_1^{x_1} * g_2^{x_2} * f^{r_y}$; i.e., $y$ is a Pedersen commitment to $(x_1, x_2, x_3)$. (The bases used for $y$ are also generators of the prime-order group, but they are different generators than used for $B, C, D, S, T$.)

3 The User computes $S = g^{1/(s+J)} * g^{x_1}$ and $T = pk_u * g^{R/(t+J)} * g^{x_2}$.

4 The User prepares a non-interactive proof that $S, T, E$ are formed correctly. This is done by proving the knowledge of $s, t, sk_u, r_B, \alpha, r_1, \beta, r_2, x_1, x_2, r_y$ such that $y = g_1^{x_1} * g_2^{x_2} * f^{r_y}$, $B = g^{sk_u} * h^{r_B}$, $g = (g^J * C)^\alpha h^{r_1}$ (the Prover knows $r_1 = -r_C/(s+J) \mod primeOrder$), $g = (g^J * D)^\beta h^{r_2}$ (the Prover knows $r_2 = -r_D/(t+J) \mod primeOrder$), $S = g^\alpha * g^{x_1}$, $T = g^{sk_u} * (g^R)^\beta * g^{x_2}$. Call this proof $\pi_{ST}$.

5 The unendorsed coin is $uecoin = (B, C, D, W, S, T, J, \pi_{CL}, \pi_{ST}, y)$.

6 The endorsement is $(x_1, x_2, r_y)$.

The Deposit protocol consists of doing Algorithms 35 and 36 together. Notice that, the resulting coin is the same as in the compact e-cash, except that the proofs $\pi_{CL}, \pi_{ST}$ are slightly different, and that we should use $S' = S/g^{x_1}, T' = T/g^{x_2}$ for double-spending detection and identification purposes.

The Withdraw protocol stays the same since no extra information needs to be stored in the wallet for the endorsed e-cash. The usefulness of endorsed e-cash will be clear once we see Verifiable Encryption and Fair Exchange protocols.

### A.7.3 E-Cash FAQ

Q: Why do we run Algorithm 31 before the withdrawal to randomize $s$?

A: If multiple users use the same $s$, then their coins will have identical $S$ values. This means, to identify a double-spender, the Bank needs to check all 2-combinations of those coins. Therefore, malicious users might mount a denial-of-service attack on the Bank this way.

Q: Why does the user pick the wallet size $W$ from only a limited list of wallet sizes?

A: The problem occurs since for efficiency we send both $W$ and the coin index $J$ in clear when spending. Consider a wallet size of 1 million coins. Only very few people will have such a wallet, and once anyone sees a coin index that is very large, the spender can be identified as one of the rich guys. A fully flexible and secure version would commit to both $W$ and $J$, and change the proofs accordingly. A range proof that proves a value is in a committed range (as opposed to a public range) must to be used [54].

Q: Why do we need Endorsed E-Cash?

A: The values $x_1, x_2, r_y$ can be easily used in verifiable encryption to prove that $y$, and hence the coin is formed correctly. Then, it is very easy and fast to fairly exchange

$x_1, x_2, r_y$ with the material being bought.

Q: Why do we need the randomness $R$ in the coin?

A: We use $R = hash(ses, \mathsf{info})$ for mainly two reasons. Use of *ses* ensures that no one other than the user and merchant knows that value, and thus prevents man-in-the-middle attacks. The use of info ensures that the merchant uses different $R$ values for each transaction, which means all the coins he receives will be honored, and so the Bank can catch double-spenders.

Note that other types of contracts can also be used. What is required from a contract is that it is different for each transaction. This benefits the merchant and the Bank, but not the User who wants to cheat. Therefore, the merchant needs to input enough randomness (in the form of info above) into the contract. The other use of the contract prevents man-in-the-middle attacks. Consider the case above where *ses* is not used and only info is used. Then, an attacker can just forward the coin he gets from the User to the merchant. One possible problem of this is that the man-in-the-middle gets the service in effect "for free". This may not present a problem in all the scenarios, but we choose to be on the safe side. With *ses* in use, the connections of the attacker to the User and the Merchant will be distinguished.

Q: We can identify the public keys of double-spenders, but can we do more?

A: Another version of e-cash is available where the Bank gets enough information to trace all the coins spent from a wallet of a double-spender. The idea is that during withdrawal, the Bank learns a verifiable encryption of the wallet secret under some other secret. Later on, when double-spending occurs, the Bank learns the secret encryption key similar to the way he learns the user's public key. Using the wallet decrypted secret, the Bank can create a blacklist for wallets/coins that the merchants should not accept. This way, further double-spending will be prevented online. Unfortunately, this version reveals all purchases of the double-spender. The

double-spending may be accidental (faulty software, stolen card, etc. ), and so the privacy of an innocent user may be lost this way. There are "glitch protection" methods [39] to let some number of accidents happen, but this means all the adversarial users will double-spend some allowed number of coins every time. Therefore, blacklisting only "future" coins is an important open problem.

## A.8 Verifiable Encryption

We will be using Caminisch-Shoup verifiable encryption [45]. This will later be the verifiable escrow method used in fair exchange.

ASSUMPTIONS:

The security of the scheme relies on Strong RSA assumption, Paillier's Decision Composite Residuosity assumption [123], and existence of a collision-resistant family of hash functions.

PARAMETERS:

The parameter $m$ denotes the number of messages that can be verifiably encrypted at once. For Endorsed E-Cash, we need $m = 3$.

SETUP:

The key generation of CS verifiable encryption uses two runs of special RSA group generation as in Algorithm 6. The important differences are clearly identified below:

- Let $N$ be the modulus returned by the first special RSA group generation. The generators will be picked in a completely different way. In fact, the group that will be in use will be a subgroup of $Z^*_{N^2}$ instead of $Z^*_N$.
- Pick a random number $f'$ from $Z^*_{N^2}$. Compute $f = f'^{2N} \mod N^2$.
- Pick random numbers $x_1, \ldots, x_m, y, z$ from the interval $(0, N^2/4)$. Compute $a_i = f^{x_i} \mod N^2$ for $1 \leq i \leq m$, $d = f^y \mod N^2$, $e = f^z \mod N^2$.

- Compute $b = (1+N) \mod N^2$. Notice that $b$ is an element of order $N$ in $Z^*_{N^2}$.

- Let the second special RSA group generation return a group with modulus $n$ and generators $g_1, \ldots, g_m, h$.

- Also, pick a key $hk$ as the key for a keyed hash function. For simplicity in presentation below, we will omit the key in hashes, but it will be used for every hash calculation.

- The public key (i.e., encryption and verification key) is composed of all public parts of the groups, and the hash function key: $VEPK = N, a_1, \ldots, a_m, b, d, e, f, n, g_1, \ldots, g_m, h, hk$.

- The secret key (i.e., decryption key) is the secret parts of the groups $VESK = P, Q, x_1, \ldots, x_m, y, z, p, q$ such that $N = PQ$ and $n = pq$.

The absolute value algorithm below will be used in the verifiable encryption

scheme a lot.

**Algorithm A.8.1**: Absolute Value procedure for Camenisch-Shoup verifiable encryption scheme.

**Input**: $x$ in range $(0, N^2)$

**Pre-conditions**: $x$ needs to be in range $(0, N^2)$

**Output**: $abs(x)$

```
abs
1  if x > N^2/2 then
2      Output (N^2 - x)  mod N^2
3  else
4      Output x  mod N^2
```

### A.8.1 Encrypt

The encryption procedure creates a ciphertext. To turn it into verifiable encryption, we will need some specialized non-interactive zero knowledge proofs. Just as we need for fair exchange, this is a labeled encryption scheme [143]. The *label* is also known

as *tag*, or *contract*.

**Algorithm A.8.2**: Encryption procedure for Camenisch-Shoup verifiable encryption scheme. This procedure is run by the Encryptor. This is a sub-procedure; it's not verifiable yet.

**Input**: Verifiable encryption public key

$VEPK = N, a_1, \ldots, a_m, b, d, e, f, n, g_1, \ldots, g_m, h, hk$, messages to be verifiably encrypted $x_1, \ldots, x_m$, a label $L$

**Output**: Ciphertext $u_1, \ldots, u_m, v, w$.

`Encrypt`

1 Pick a random number $r$ from the interval $(0, N/4)$.

2 **for** $i$ *:* $1..m$ **do**

3 Compute $u_i = b^{x_i} * a_i^r \mod N^2$.

4 Compute $v = f^r \mod N^2$

5 Compute $w = abs([d * e^{hash(u_1||\ldots||u_m||v||L)}]^r \mod N^2)$

6 Output $u_1, \ldots, u_m, v, w$.

## A.8.2 Verifiably Encrypt

We will now provide a conversion from regular encryption to verifiable encryption for Camenisch-Shoup verifiable encryption scheme. This will involve non-interactive zero knowledge proofs.

This verification can be used to prove correct encryption of discrete logarithms. When we consider Endorsed E-Cash, the Prover will verifiably encrypt $x_1, x_2, r_y$ such that $y = g_1^{x_1} * g_2^{x_2} * f^{r_y}$ in the group chosen by the Endorsed E-Cash setup. Take a note that it is a different group than the ones used in verifiable encryption here. $X$

below will be $y$ of the Endorsed E-Cash.

**Algorithm A.8.3**: Verifiable Encryption procedure for Camenisch-Shoup verifiable encryption scheme. This procedure is run by the Encryptor/Prover.

**Input**: Verifiable encryption public key

$VEPK = N, a_1, \ldots, a_m, b, d, e, f, n, g_1, \ldots, g_m, h, hk$, commitment $X$ to all messages $x_1, \ldots, x_m$ (or a commitment $X_i$ to each one of them), a label $L$

**Output**: Ciphertext and its proof $u_1, \ldots, u_m, v, w, X', \pi_{VE}$.

`Verifiably Encrypt`

1. Get the ciphertext $u_1, \ldots, u_m, v, w$ from the Encrypt function in Algorithm 38, along with the randomness $r$ used in the encryption.
2. Pick a random number $s$ from the interval $(0, N/4)$.
3. Compute a Fujisaki-Okamoto commitment $X'$ to $x_1, \ldots, x_m$ using the randomness $s$ in the group defined by $n, g_1, \ldots, g_m, h$ (the second RSA group generated for verifiable encryption purposes).
4. Prove knowledge of $r, s, x_1, \ldots, x_m$ such that $v^2 = f^{2r} \mod N^2$, $w^2 = [d * e^{hash(u_1||\ldots||u_m||v||L)}]^{2r} \mod N^2$, and that each $u_i^2 = b^{2x_i} * a_i^{2r} \mod N^2$, and that each $x_i$ corresponds to the one in $X'$ and $X$, and that the range of each $x_i$ is $(-N/2, N/2)$ (using a Range proof). Call this proof $\pi_{VE}$.
5. Output $u_1, \ldots, u_m, v, w, X', \pi_{VE}$.

Any party with the verifiable encryption public key can verify the encryption.

**Algorithm A.8.4**: Verification procedure for Camenisch-Shoup verifiable encryption scheme. This procedure is run by the Verifier.

**Input**: Verifiable encryption public key

$VEPK = N, a_1, \ldots, a_m, b, d, e, f, n, g_1, \ldots, g_m, h, hk$, commitment $X$ (or commitments $X_1, \ldots, X_m$), a label $L$, ciphertext and its proof $u_1, \ldots, u_m, v, w, X', \pi_{VE}$

**Output**: acceptor reject.

`Verify`

1 Verify $\pi_{VE}$. If verification fails, output reject.

2 **if** $abs(w) = w \mod N^2$ **then**

3 Output accept.

4 **else**

5 Output reject.

### A.8.3 Decrypt

Only a party equipped with the correct secret key can decrypt. In the fair exchange, this party will be the Arbiter.

**Algorithm A.8.5**: Decryption procedure for Camenisch-Shoup verifiable encryption scheme. This procedure is run by the Decryptor/Arbiter.

**Input**: Verifiable encryption public key

$VEPK = N, a_1, \ldots, a_m, b, d, e, f, n, g_1, \ldots, g_m, h, hk$, associated secret key $VESK = P, Q, x_1, \ldots, x_m, y, z, p, q$, a label $L$, ciphertext $u_1, \ldots, u_m, v, w$

**Output**: Plaintext $m_1, \ldots, m_m$ or error.

`Decrypt`

1 **if** $abs(w) \neq w \mod N^2$ *OR* $v^{y+z*hash(u_1||\ldots||u_m||v||L)} \neq w^2 \mod N^2$ **then**
2  Output error.
3 Compute $t = 2^{-1} \mod N$.
4 **for** $i$ *:* $1..m$ **do**
5  Compute $m'_i = (u_i/v^{x_i})^{2t} \mod N^2$.
6  Set $m_i = (m'_i - 1)/N \mod N^2$.
7  **if** $m_i$ *is not in range* $(0, N)$ **then**
8  Output error.
9 Output $m_1, \ldots, m_m$

Proving decryption is done correctly is also possible as given in [45], although it is a complicated algorithm. But, we employ verifiable encryption in fair exchange, and the decryption is performed by a trusted third party. Since he is a trusted entity, we do not expect a proof of decryption. We leave this as possible future work.

## A.9 Merkle Tree

A Merkle tree is a binary tree where each leaf is a hash of some data, and each non-leaf node is a hash of its two children. Below we provide pseudocodes on how to employ Merkle trees to perform verification of any leaf. The nice property is that it requires $O(\log n)$ time to prove and verify that a leaf is part of a Merkle tree with a total of $n$ leaves.

First, we give the procedure $MHash(h, block, \ell)$ to compute *bhash*, the Merkle hash function [109] that outputs the root of the depth-$\ell$ Merkle tree of a given *block* with hash function $h$. Next, we give two procedures. One is used to compute the proof $MProve((h, block, \ell), i)$ that a particular value is the $i^{th}$ leaf of the Merkle hash tree of which *bhash* is the root; and the other procedure $MVerify((h, \ell), (bhash, i, chunk))$ is used to verify the proof *MProof* that the value *chunk* is the $i^{th}$ leaf of the Merkle hash tree of which *bhash* is the root.

Let us start by explaining the idea of the *MHash* procedure. Consider a rooted binary tree with $2^\ell$ leaves (i.e., it is a binary tree of height $\ell$). Associated with every node the tree, there is the address $a$ of the node. Specifically, the label $a$ associated with the root node is the empty string $\varepsilon$. The label of a left (resp. right) child is derived by concatenating 0 (resp., 1) to the label of the parent node. Thus, the label $a$ associated with the $i^{th}$ leaf is the integer $i$ written in binary. Stored at a node labeled with $a$, is a value $v_a$. In a Merkle tree, stored at each leaf $0 \leq a \leq 2^\ell - 1$ is the value $v_a = h(chunk_a)$, where $chunk_a$ is the $a^{th}$ chunk of the block; and stored at every internal node $0 \leq a \leq 2^i - 1$ at depth $i \geq 1$ is the value $v_a = h(v_{a||0}||v_{a||1})$. Finally, the value corresponding to the root node is $v_\varepsilon = h(v_0||v_1)$. The algorithm

*MHash* outputs this value.

**Algorithm A.9.1**: *MHash*: Generating a Merkle hash tree.

**Input**: A collision-resistant hash function $h : \{0,1\}^* \mapsto \{0,1\}^{hashLength}$, the data block *block*, the desired tree height $\ell$.

**Pre-conditions**: None.

**Output**: The value *bhash*.

**Post-conditions**: None.

1 If $\ell = 0$, output $h(block)$.

2 Otherwise, divide the block *block* into $block_0$ and $block_1$, 2 strings of approximately equal byte lengths (this has to be done in a deterministic fashion so that running twice on same input gives same results) and output $h(MHash(h, block_0, \ell - 1)||MHash(h, block_1, \ell - 1))$

To prove that *chunk* is the $i^{th}$ chunk of the block, whose hash value is $bhash = MHash(block)$, reveal the values $v_{a_j}$ where for $1 \leq j \leq \ell$, the label $a_j$ is obtained by taking the first $j-1$ bits of the binary representation of $i$, and concatenating to them the negation of the $j^{th}$ bit. (For example, if $i = 0101$, then $a_1 = 1$, $a_2 = 00$, $a_3 = 011$ and $a_4 = 0100$.) In the Merkle tree, the node $a_j$ will be the sibling of the $j^{th}$ node on the path from the root to the chunk in question. The procedure for doing this is as

follows:

**Algorithm A.9.2**: Generating the proof $MProof = (v_1, \ldots, v_j, chunk)$ that *chunk* is the $i^{th}$ leaf of the Merkle tree.

**Input**: A collision-resistant hash function $h : \{0,1\}^* \mapsto \{0,1\}^{hashLength}$, the data block *block*, the desired tree height $\ell$; the index $i$.

**Pre-conditions**: $0 \leq i < 2^\ell$

**Output**: The value *MProof*.

**Post-conditions**: None.

1 If $\ell = 0$, return *block*.

2 Otherwise, divide the block *block* into $block_0$ and $block_1$, 2 strings of approximately equal byte lengths (this has to be done in a deterministic fashion, the same as what is done by the algorithm *MHash*).

3 If $i \geq 2^{\ell-1}$, return $(MHash(h, block_0, \ell - 1), MProve((h, block_1, \ell - 1), i - 2^{\ell-1}))$.

4 Else return $(MHash(h, block_1, \ell - 1), MProve((h, block_0, \ell - 1), i))$.

To verify that $(v_1, ..., v_\ell)$ is a valid proof that *chunk* is the $i^{th}$ chunk of *block* associated with *bhash*, we recompute the labels $a_j$ (as above). We know that $v_j = v_{a_j}$ is the value that should be associated with node labelled $a_j$. Let $i = i_1 i_2 \ldots i_\ell$, i.e., $i_j$ is the $j^{th}$ bit of the $\ell$-bit binary representation of $i$. Let $b_j = i_1 \ldots i_j$ be the $j$-bit prefix of $i$. First, we know that $v_i = h(chunk)$. For each $j$, $\ell - 1 \geq j \geq 0$, compute $v_{b_j} = h(v_{b_j||0}||v_{b_j||1})$. We can do it because one of $(v_{b_j||0}, v_{b_j||1})$ is $v_{a_{j+1}}$, and the other one is computed in the previous step. Finally, verify that $v_\varepsilon = bhash$.

If two conflicting proofs can be constructed (i.e., for $chunk \neq chunk'$, there are proofs that each of them is the $i^{th}$ chunk of *block* associated with *bhash*), then a collision in $h$ is found, contradicting the assumption that $h$ is collision-resistant.

## A.10 Skip List

The *skip list* data structure (see Figure A.1) is an efficient means for storing a set $S$ of elements from an ordered universe. One can think of it as a randomized Merkle tree that supports the following operations efficiently: $\mathsf{find}(x)$ (determine whether element $x$ is in $S$), $\mathsf{insert}(x)$ (insert element $x$ in $S$), and $\mathsf{delete}(x)$ (remove element $x$ from $S$). It stores a set $S$ of elements in a series of linked lists $S_0, S_1, S_2, \ldots, S_t$. The base list, $S_0$, stores all the elements of $S$ in order, as well as sentinels associated with the special elements $-\infty$ and $+\infty$. Each successive list $S_i$, for $i \geq 1$, stores a sample of the elements from $S_{i-1}$. To define the sample from one level to the next, we choose each element of $S_{i-1}$ at random with probability $\frac{1}{2}$ to be in the list $S_i$.

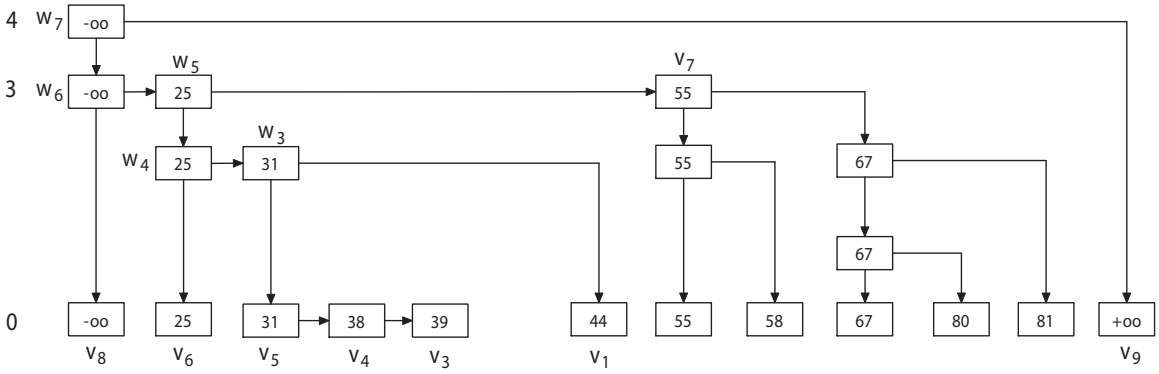


Figure A.1: A skip list used to store the ordered set $\{25, 31, 38, 39, 44, 55, 58, 67, 80, 81\}$. The proof for the existence of element 39 (and for the absence of element 40) as proposed in [86] is the set $\{44, 39, 38, 31, f(v_1), f(v_6), f(v_7), f(v_8), f(v_9)\}$. The recomputation of $f(w_7)$ is performed by sequentially applying $h(\cdot, \cdot)$ to this set.

## A.11 ASW Fair Exchange

We present, for reference, the fair exchange protocol for exchanging signatures, due to Asokan, Shoup, and Waidner (ASW) [7]. Alice and Bob would like to exchange

their signatures on some contract. The version we present below does not employ timeouts. The protocol in its basic sense (without conflict resolution details) is:

1. Alice sends Bob a non-verifiable escrow of her signature, with a label defining how Bob's signature should look like. Bob checks if the definition is the correct definition.

2. Bob sends Alice a *verifiable* escrow of his signature, with the label defining how Alice's signature should look like and also attaching the escrow he obtained in step 1. Alice verifies the verifiable escrow. She furthermore checks if the label is formed correctly. If anything goes wrong at this step or a *message timeout* occurs, she aborts the protocols and runs AliceAbort with the Arbiter.

3. Alice sends Bob her signature. Bob verifies this signature, and stops and runs BobResolve if it does not verify or a *message timeout* occurs.

4. Bob sends Alice his signature. If the signature does not verify, Alice runs AliceResolve.

The AliceAbort, BobResolve and AliceResolve protocols have a similar logic as ours. AliceAbort tells the Arbiter to consider that trade as aborted and not to honor any further resolution request on that particular trade. BobResolve gets Alice's signature by providing Bob's signature, and similarly, AliceResolve gets Bob's signature by providing Alice's signature. ASW provide a more complicated protocol for exchanging an electronic check for a digital file, building on top of their signature exchange protocol. The details of both protocols can be found in [7].