Abstract of "Anonymous Accounting for Decentralized Systems"
by C. Chris Erway, Ph.D., Brown University, May 2011

Peer-to-peer systems have been proposed for a wide variety of applications, such as file-sharing, distributed storage, and distributed computation. These systems seek the benefits of a decentralized design—chiefly, the ability of a system to self-scale as new participants join, since participants are motivated to contribute resources that offset the added workload they generate. Decentralization also offers improved fault-tolerance and user privacy, because no central authority is responsible for orchestrating or recording peer interactions.

However, these beneficial system properties are at risk from selfish participants. While many peer-to-peer systems provide incentives for encouraging participation, past work has shown that these mechanisms can be gamed by selfish peers that consume resources while providing little or none in return. For example, the pairwise reputation scheme used by BitTorrent applies only in the short term of a single download; new peers must be bootstrapped by altruistic service.

A currency-based accounting system provides fungible, long-term incentives for participation that persist beyond the limited scale of a single download or pairwise interaction. However, currency raises a number of privacy and systems concerns arising from the infrastructure required to support it. For example, a "bank" must check and count currency, which presents a potential performance bottleneck and privacy concern. In the digital world, the privacy of peer interactions can be preserved through the use of anonymous, cryptographically secure electronic cash (e-cash).

This thesis shows that e-cash is a practical technique for ensuring fairness, robustness, and better long-term incentives in decentralized systems. It investigates how to build systems that anonymously account for three different resource types—bandwidth, storage, and computation—through the application of protocols for cryptographic fair exchange and e-cash, and how to mitigate the overhead involved in doing so. As a proof of concept, this thesis introduces Cashlib, an open-source library for e-cash; ZKPDL, a programming language that speeds both the performance and development of cryptographic implementations; and FairTrader, a currency-based file sharing system that uses e-cash to provide long-term, reliable service for users.

Anonymous Accounting for Decentralized Systems

by
C. Chris Erway
B. A., Cornell University, 2002
Sc. M., Brown University, 2006

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

Providence, Rhode Island
May 2011

This dissertation by C. Chris Erway is accepted in its present form
by the Division of Computer Science as satisfying the
dissertation requirement for the degree of Doctor of Philosophy.

Date _____ _____

John Jannotti, Director

Recommended to the Graduate Council

Date _____ _____

John Jannotti, Reader

Date _____ _____

Anna Lysyanskaya, Reader

Date _____ _____

Roberto Tamassia, Reader

Approved by the Graduate Council

Date _____ _____

Peter M. Weber, Dean of the Graduate School

Dedicated to the memory of my grandparents
Dr. Charles Eugene Erway
Edith Mae Daugard Erway
Sywe-Yu Chen 陳學漁
Joyce Teng 鄧幸生

Acknowledgments

I would first like to thank my advisor, John Jannotti, whom over the years has provided me with immense perspective and helped hone my instincts and opinions relating to computer systems and networks. I am deeply indebted for his wisdom, patience, and insight as I have navigated my research career here at Brown. I am also indebted to my committee members and collaborators, Anna Lysyanskaya and Roberto Tamassia; without their assistance (and cryptographic protocols) this thesis would not have been possible.

Much of the work presented in this dissertation was produced as a result of collaborations with the other members of the Brownie Points Project: John Jannotti, Anna Lysyanskaya, Mira Belenkiy, Melissa Chase, Theodora Hinkle, Alptekin Küpçü, Sarah Meiklejohn, and Eric Rachlin [18, 17]. In particular, the Cashlib/ZKPDL project [125] would not have been possible without Sarah Meiklejohn, who was instrumental in guiding it from idea to implementation, as well as Thea Hinkle, and also our summer research collaborators Gabriel Bender and Alex Hutter. I would also like to thank Charalampos Papamanthou and Alptekin Küpçü for our fruitful collaboration on proofs of storage [71], which is also presented in this dissertation.

My time at Brown has benefited greatly from the advice and fellowship of many students here, among them my officemates Glencora Borradaile, Nikos Triandopoulos, Guy Eddon, Crystal Kahn, and Steve Gomez; fellow grads past and current David McClosky, Peter Sibley, Yanif Ahmad, Guillaume Marceau, Luc Mercier, Tomer Moscovich, Tori Sweetser, Stella Frank, Casey Marks, Eric Koskinen, Michael Gaiman, Andy Pavlo, and Carleton Coffrin (who helped getting our ZKPDL grammar going with ANTLR [143]).

I would never have ended up at Brown without the mentorship of my old co-workers at IBM Research, among them José Castaños (himself a Brown CS PhD, and a primary factor in my decision to attend Brown), José Moreira, Gheorghe Almási, and Călin Caşcaval. Their decision to keep me around at IBM and show me the ropes in a systems research lab offered an extraordinary privilege for which I will remain forever grateful.

My life in Providence would have fared much poorer without the good times shared with Eric Rachlin, Tibet Sprague, Phil Trevvett, David Segal, Harry Siple, Dan Bass, Joseph Gulezian, and Eric Johnson, and all the current and past members of the What Cheer? Brigade. I especially must thank Mindy Stock for her emotional support, camaraderie, and advice. My time in New York would have been much less stimulating without Ben Ransford, Michael Manning, and Bryan Renne.

I would also like to express my deepest gratitude to my co-founders Spiros Eliopoulos and Dan Kuebrich, and colleagues Jason Rassi and TR Jordan, for all their support in holding things down at Tracelytics while I worked to complete this thesis.

My research has been supported in part by funding from the National Science Foundation Cyber Trust grant program.

Finally, and most importantly, I would like to thank my family for their perpetual support and encouragement. My parents, Chip and Tina, brought me into this world and made sure I did well at school. My sister Cathy and uncle Jo-Jo have always been clear-minded advisors and advocates. This dissertation is dedicated to my grandparents, and I hope it may honor their memory.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The past decade has witnessed an explosion in the popularity of peer-to-peer systems, driven by increased global Internet connectivity and rising end-user bandwidth capacities. These P2P systems in general eschew the traditional, centralized client-server model of providing a network service in favor of the decentralized exchange of resources between end systems, or "peers."

Peer-to-peer systems have been designed and deployed for a wide variety of applications—such as content distribution, distributed storage, and distributed computation—and today account for a significant fraction of Internet traffic, largely from file-sharing systems systems such as BitTorrent. Their decentralized nature provides many advantages, chiefly the ability to self-scale: new participants are motivated to contribute resources that offset the added workload they generate. Decentralization also offers improved fault-tolerance and user privacy, since no central authority is responsible for orchestrating or recording peer interactions.

However, these beneficial system properties are at risk from selfish participants. While many peer-to-peer systems provide incentives for encouraging participation, past work has shown that these mechanisms can be gamed by selfish peers that consume resources while providing little or none in return. For example, the pairwise reputation scheme used by BitTorrent applies only in the short term of a single download; new peers must be bootstrapped by altruistic service.

A currency-based accounting system provides fungible, long-term incentives for participation that persist beyond the limited scale of a single download or pairwise interaction. However, currency raises a number of privacy and systems concerns arising from the infrastructure required to support it. For example, a "bank" must check and count currency, which presents a potential performance bottleneck and privacy concern. In the digital world, the privacy of peer interactions can be preserved through the use of anonymous, cryptographically secure electronic cash (e-cash).

1.1 Thesis statement

This dissertation investigates the following thesis:

> E-cash accounting techniques are practical and well-suited for providing fairness, robustness, and better long-term incentives in decentralized systems.

1.2 Contributions

This dissertation presents system designs and key primitives that enable the efficient use of anonymous e-cash to to account for three different resource types—bandwidth, storage, and computation—through the application of protocols for cryptographic fair exchange and e-cash. Specifically, these contributions are:

- *FairTrader*, a peer-to-peer file-sharing system that anonymously accounts for bandwidth through the use of protocols for cryptographic fair exchange and e-cash.
- *ZKPDL*, a programming language and interpreter for zero-knowledge proof protocols designed with the goal of mitigating the performance overhead and implementation difficulty of e-cash protocols.
- *Cashlib*, a software library implementation for endorsed e-cash that allows systems to be built that buy and barter for resources through the exchange of coins, keys, and contracts describing those resources.
- *DPDP*, a protocol that a user paying for a remote storage service could use to verify that her data had not been tampered with, using a concise proof of possession that efficiently supports updates to the outsourced data.
- A model of a system using currency incentives for outsourcing computational tasks, and an analysis of how to control malicious behavior in such a system.

1.3 Thesis outline

The remainder of this thesis is organized as follows. An overview of the BitTorrent peer-to-peer system [55], and previous work related to this thesis on incentives in BitTorrent and other distributed systems, is provided in Chapter 2. Chapter 3 presents a general design for using currency in distributed systems, introduces cryptographic electronic cash (e-cash), and the protocols, system entities, and a set of protocols using endorsed e-cash [42] that allow peers to buy and barter for resources.

This design is further refined for different resource types over the following three chapters. Chapter 4 deals with accounting for distributed computation systems, motivated by concerns for the scalability of the e-cash bank. Chapter 5 describes a scheme for concise proofs of data possession, which provide probabilistic guarantees to a consumer of storage outsourcing services like those provided by P2P backup networks or Amazon's S3. In Chapter 6 we present the design and implementation of FairTrader, a currency-based file-sharing system based on BitTorrent that uses e-cash to provide long-term, reliable service for users.

In Chapter 7 we deal with the practical considerations of implementing computationally expensive e-cash protocols, and introduce Cashlib, a software library which implements e-cash, and ZKPDL, a programming language and interpreter for zero-knowledge proof protocols that speeds both the performance and development of cryptographic implementations. Finally, Chapter 8 concludes the thesis and considers future work.

Chapter 2

Background and related work

This chapter provides an overview of peer-to-peer systems, the popular file distribution system BitTorrent [55], and surveys related work on providing incentives in distributed systems.

2.1 Peer-to-peer systems

The Internet has always supported "peer-to-peer" communication—indeed, communication between pairs of hosts has always been the primary means by which data is transferred over the Internet—but in the past decade the term *peer-to-peer* (P2P) has come to refer to a new, popular category of distributed systems. In this context, P2P refers to system designs which, in general, delegate the task of serving an end system's request for resources not to centralized entities, but rather to a much larger set of other end systems, or *peers*.

This principle of *decentralization* sets P2P systems in contrast with earlier protocols and architectures based on the popular model whereby many *clients* connect to a relatively smaller set of (often high-powered) *servers*. The client-server model is widely prevalent today, with perhaps its most prominent example being the world web web. While the HTTP protocol (which underlies each web request) is employed by billions of users today, the overwhelming majority of users of HTTP clients do not also run HTTP servers.

However, two major trends have driven the rise of decentralized systems and P2P networks online: an increase in both the number of Internet users globally, and the average bandwidth rates experienced by those users. As the number and speed of consumer Internet end-systems have grown, it has become simply more feasible for tasks such as content distribution, storage, computation, and lookup to be delegated to the millions of users who have historically been labeled as "clients" in conventional architectures.

P2P networks exploit these trends to provide decentralized services that organically *self-scale* as more users participate. An ideal peer-to-peer system design contains no centralized bottleneck limiting the ability of the system to provide service to new users. Decentralization also offers better fault-tolerance and attack resistance (since no single point of failure exists that can take down the whole system), low barrier to deployment (since less dedicated infrastructure is needed to deploy a P2P service), and enhanced user privacy (since no single entity is able to record every event that occurs between peers) [155].

The history of peer-to-peer systems began in 1999, with the release of three influential systems: the Napster music-sharing client, the Freenet anonymous data storage network, and the SETI@home distributed computing project [155]. Since then, the concepts introduced by these systems have been widely deployed in systems ranging from the popular

file-sharing protocol BitTorrent [55] to the Skype video and phone service, which routes calls through other users of Skype to their destinations [15]. The computing resources provided by these three initial systems—bandwidth, storage, and computation—are each considered in turn in Chapters 4, 5 and 6 of this thesis.

2.2 BitTorrent

BitTorrent [55], introduced by Bram Cohen in 2001, is the most popular peer-to-peer file-sharing system in use today. One measurement study in 2009 found, for example, that data transferred via the BitTorrent protocol accounts for roughly half of Internet traffic in some areas of the world [95]. Another study from 2010 reported that one-fifth of Internet traffic (and nearly half of consumer upstream traffic) in North America was due to BitTorrent [158]. Our focus on BitTorrent is motivated by its popularity, and we aim to show that the techniques presented in this thesis are applicable to the patterns of BitTorrent usage most common today.

2.2.1 System overview

A BitTorrent download begins with a metadata file, or *torrent*, listing the name and size of the file(s) for distribution, the hash value of each data block (typically sized 64KB–2MB) needed to assemble the file(s), and the address of a *tracker* assigned to manage the swarm of participants. A node enters the network by announcing itself to the centralized tracker server. The tracker's reply provides the Internet addresses of a random subset of active nodes.

Using this list, a node makes neighbors by attempting to directly connect to other nodes. Upon connecting, neighbors start to exchange block-availability advertisements and issue block requests, using a local "rarest-first" heuristic designed to improve data availability. Nodes typically maintain many neighbor connections, but at each round upload requested blocks only to a small fixed-sized set of *unchoked* neighbors, selecting those that have recently provided the fastest rate of service. The rest are *choked* until the next round, typically ten seconds later, when they will be re-evaluated for reciprocation.

2.2.2 Incentives

BitTorrent relies on a tit-for-tat bandwidth reciprocity mechanism whereby cooperative nodes are rewarded for uploading by their neighbors, who *unchoke* them. Thus nodes are encouraged to upload to their neighbors, since doing so offers the possibility of faster download rates. While the designer of BitTorrent describes this mechanism as a tit-for-tat scheme [55], Levin et al. [111] observe that the unchoking mechanism—which awards the next round's upload slots to the best peers from the last round—is best modeled as an auction, and suggest a modified sharing strategy to reflect this.

Whether an auction or tit-for-tat, BitTorrent presents many opportunities for selfishness [146, 116] and free-riding [100, 119]. For example, In search of ever-better neighbors, nodes also select two peers at random for *optimistic unchoking* every three rounds; this also helps to bootstrap new users, who may not have enough data to reciprocate. Once a download is complete, nodes receive no benefit from participating, and most depart [146]; nodes that persist become altruistic *seeders*. By providing unaccounted service, both optimistic unchoking and seeding may potentially enable free-riding.

The tracker has limited ability to perform resource accounting: when nodes periodically re-announce themselves (*e.g.*, to check for new nodes), they report the total amount of data uploaded and downloaded for the torrent so far. Online communities have emerged around registration-only trackers that enforce sharing ratios by totaling user activity across multiple torrents, leading to increased seeding activity [5]. These communities are discussed further in Section 2.2.6.

2.2.3 Fairness and performance

Studies have found the tit-for-tat choking mechanism highly effective in discouraging free-riding, encouraging the clustering of similar-bandwidth peers, and fully utilizing participants' uplink bandwidth.

Legout *et al.* [110] highlight these properties through experiments on PlanetLab, limiting each node's maximum upload rate to define different classes of participation. Their measurements show that, in a "flash crowd" scenario with a fast seed, nodes are more likely to exchange data with neighbors of their own class, forming similar-bandwidth clusters. This follows from the intuition that, to a faster node, a slower neighbor's unchokes are less useful, so are less often reciprocated. As a result, higher-class participants are rewarded with faster downloads. The simulation-based study of Bharambe *et al.* [23] also finds the choking mechanism effective in fully utilizing upload bandwidth.

However, both studies also provide evidence that high uplink utilization (and the resultant fast average download times) comes at the expense of *fairness*: fast nodes typically contribute much more data than they upload. In [23] the authors find that in heterogeneous settings, some high-bandwidth nodes upload more than seven times more data than they download. The experiments of [110] also observe similar unfairness, and show it is exacerbated as clustering breaks down in the presence of a slow seed. Piatek *et al.* [146] observe and model unfairness as *altruism*—their term for any data uploaded in excess of that downloaded—which they exploit with a selfish client implementation.

Unfairness can be partially explained by the slow search process nodes undertake when looking for better neighbors, with optimistic unchokes occurring only once every 30 seconds. Converging on a set of similar-bandwidth peers in a large, high-churn torrent may take a long time (or forever), and along the way many high-capacity nodes provide unmetered service to slower neighbors [146]. In response, [23] considers modifying the tracker, allowing it to induce clustering by matching similar-bandwidth nodes. The authors report that reducing bandwidth-mismatched pairings leads to improvements in both uplink utilization and fairness.

This has led some researchers to design alternate choking mechanisms that provide stronger fairness to participants. A strict block-level tit-for-tat choking policy, bounding the number of excess blocks transferred to a neighbor without reciprocation, is considered as a replacement for the default rate-based policy in [23]. It is shown to reduce unfairness at the expense of lower average upload utilization, slowing overall performance (though its performance fares better in conjunction with a bandwidth-matching tracker, or at high node degree). Fan *et al.* [72] present an analytical model characterizing the design space of rate assignment strategies in BitTorrent-like systems, demonstrating a fundamental trade-off between optimal performance and fairness.

2.2.4 Free-riding

The ease of free-riding in BitTorrent arises from the problem of bootstrapping new peers and new peer-to-peer connections. Altruism is required when peers first enter a swarm; since these new arrivals have no data to reciprocate, they must rely on uploads from seeders or from the optimistic unchokes of fellow downloaders to receive an initial set of data blocks and begin exchanges with others.

Optimistic unchokes are performed between peers which have had no history of past contact. The repeated game performed by peers as they seek ever-better neighbors requires an initial trial period (usually lasting 30 seconds) during which upload bandwidth is provided to a new neighbor in hopes of reciprocation. In a large swarm, however, many hundreds or thousands of neighbors might be considered in this manner before settling on a set of neighbors that offer satisfactory reciprocal benefits, wasting much bandwidth in the process.

Altruistic behavior can be gamed by selfish participants, and research implementations of BitTorrent have been developed that successfully take advantage of the altruistic behavior of typical BitTorrent users to provide better performance [146] or even to obtain downloads without providing any reciprocative uploads at all [119].

2.2.5 Multiple swarms

Finally, BitTorrent's tit-for-tat scheme provides incentives based on local reputations that are limited in scope to single swarms only; i.e., a user's activity in each swarm is considered completely in isolation. As a result, users engaged in multiple concurrent swarms cannot use uploads, reputation, or credit in one swarm to improve performance in another. This poses a problem, because users are often engaged in multiple swarms at once, and not just in single-swarm "flash crowds."

This problem also extends to a user's activity over time. Under the current scheme, a new peer starts "from scratch" each time it enters a new swarm; this is true even if that user performed admirably by providing altruistic service a day or an hour before beginning the new swarm.

2.2.6 Ratio trackers

A variety of popular web-based communities surrounding BitTorrent trackers have proliferated in recent years. These communities, which often focus on a specific genres (such as music, television shows, content in specific languages, etc), offer primarily search services, but also provide user registration, discussion fora, and collect user statistics.

Some trackers use these statistics to implement additional incentive mechanisms, such as imposing minimum ratio requirements or upload/download credit systems, in order to motivate to seed torrents after they have completed downloading and provide them with fair rewards proportional to their effort. Recent measurement studies [5, 128, 50] describe a wide variety of incentive systems, minimum ratios, and credit schemes that have been developed by administrators of these communities. Meulpolder *et al.* report that despite these differences, "Apparently, it is most of all important that there is a ratio enforcement mechanism in place; the precise rules matter less." [128].

These schemes allow ratio tracker communities to provide tremendous performance gains over trackers that do not institute ratio requirements. Download performance, average seeding duration, and content availability all increase by orders of magnitude; peers continue seeding content for days, rather than minutes, enabling these trackers to offer a library of

content available at any time for peers to download directly from seeders, rather than just a snapshot of those files popular at the moment. As Meulpolder *et al.* summarize:

> We conjecture that ratio enforcement mechanisms are the primary cause of the high numbers of seeders in the private communities, and in the end render BitTorrent's tit-for-tat reciprocity mechanism virtually irrelevant in such communities. [128]

There are strong similarities between these trackers and the currency-based accounting system proposed in this thesis. Kash *et al.* provide an economic analysis of one such ratio-tracking site, describing how users employ strategies to (in their words) "earn money," seek out torrents with high earning potential, and respond to economic forces and other system events such as "free leech" periods (during which administrations suspend the ratio system, alleviating unmet demand from users who want files they cannot normally afford) [103]. These BitTorrent economies provide a model for the currency design (Chapter 3) and FairTrader system (Chapter 6) proposed here.

However, ratio trackers rely on self-reported client statistics that can be easily manipulated: the BitTorrent specification [54] requires only that the total number of bytes uploaded and downloaded be reported to the tracker periodically (as a base-10 integer). While huge distortions in these figures could be detected automatically, more modest inflations of these statistics in busy swarms are less likely to be detected by tracker administrators, and proxy software is available for selfish users to do so [90]. Ratio tracking sites also weaken user privacy, as site administrators must retain data on every swarm that each user has joined—especially if they want to check for ratio manipulators—and summarize those statistics when computing ratios.

In contrast, a currency-based accounting system that uses e-cash offers both strong security properties (as upload credit cannot be forged) as well as strong privacy-preserving properties. A comparable minimum ratio requirement could be enforced by simply imposing minimum balance limits for each user's e-cash bank account. The credit earned and deposited with the e-cash bank would not be linkable by anyone to user activity, and site administrators would have no need to record or retain user activity.

2.3 Related work

This thesis is not the first to propose currency-based approaches to resource allocation in distributed systems. Previous work has seen currencies used to allocate resources in grids [96] and sensor networks [52], pay for distributed storage [93], and optimize queries in distributed databases [166]. Game-theoretic analyses of peer-to-peer currency systems and associated equilibria are detailed in [87, 76, 102].

Systems based on Byzantine fault tolerance (BFT) [1] provide safety and liveness guarantees given a certain tolerable fraction of malicious users; typically at least two-third of participants must act correctly. In BAR [1], the system is able to tolerate any number of rational (i.e., selfish) users, as it creates incentives for even the most self-serving of peers to behave in a manner that is fair. BAR gossip [114] 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. A high degree of fault tolerance is guaranteed even in the face of a high proportion of completely self-serving, or even malicious peers. 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.

2.3.1 P2P currencies

In the file-sharing setting, a number of other systems have attempted to use currency to provide incentives. Mojo Nation [175] (incarnated later as Mnet) used currency, called Mojo, to incentive users to upload content, much as we do in FairTrader, but without the privacy-preserving guarantees. (A former employee of Mojo Nation, Bram Cohen, drew much inspiration from the block-exchange file-transfer mechanism used by that system in his later design for BitTorrent.)

In academia, Karma [172] describes a P2P currency built atop of a DHT to avoid centralization; this places the responsibility of maintaining each peer's bank account and transaction history with a "bank-set" that consists of randomly-chosen nodes from the DHT. As with Mojo Nation, however, the currency used does not provide any privacy for the system's users. This decentralization makes it difficult to effectively manage the currency (see Section 3.5.2), requiring $O(N^2)$ messages.

In Dandelion [165], users are again awarded credit when they share content with authorized users. The transaction protocol, however, requires server interaction every time (which we avoid by using off-line e-cash) and thus has limited privacy guarantees and the potential to run into problems with scalability. Finally, BitStore [150] also offers currency incentives for peers to store and upload files in an effort to solve what they call the Blocked Leecher problem. As with the other systems, their currency is not privacy-preserving and their incentive mechanism does not reward behavior across swarms. Though these systems provide useful examples of currency-based design, they do not address issues of privacy and security that we consider essential for adoption in peer-to-peer systems.

There has been fascinating work by game theorists that model and analyze theoretical peer-to-peer currency systems and associated economic equilibria [87, 76, 102]. Kash et al have analyzed P2P systems consisting of agents which exchange a "scrip" currency with other agents in return for service [76, 102]. Their analysis provides many useful insights on the behavior of selfish agents in a P2P currency economy [76] and how a currency system's bank might manage the currency to avoid inflation and monetary crashes [102]. These analyses describe systems very similar to those proposed by this thesis.

2.3.2 Reputation systems

Beyond currency-based incentive mechanisms, reputation systems, both local and centralized, have also been used to provide incentives in peer-to-peer systems [177, 124, 147]. Some systems even quantify local reputation values in terms of credit [133] or currency [3], thus extending the barter economy by identifying chains of indebted nodes that can be used for transitive exchanges.

In one example, Scrivener [133], a structured lookup service allows peers to maintain an "account balance" with neighbors they know. Local reputation is represented as a non-fungible currency in Scrivener, requiring peers to have balanced behavior over short time periods. Piatek et al. [147] rely on highly-connected peers to maintain reputation information and act as intermediaries; this causes the system to rely on a relatively small number of super-users for its effectiveness, and poses potential problems similar to those existing already in BitTorrent.

However, reputation schemes are less effective when discredited pseudonyms can be shed easily, thus limiting their applicability in privacy-enhancing peer-to-peer systems [123] or anonymous networks such as Tor [64]. Friedman and Resnick [77] call this the "social cost of cheap pseudonyms" and show that in this environment, participants must distrust new users until they have earned enough reputation; this inevitably yields problems with efficiency at the start of sessions.

The currency-based described in this thesis, unlike these reputation systems, can be thought of as providing fungible reputation independent of identity-based observations, thus making it well-suited for systems with high churn, or ones that use pseudonymous or anonymous identities.

2.3.3 Distributed computing and grid resource sharing

The growth in the number of computers connected to the Internet, in the size of federated resource-sharing clusters spanning different administrative domains (such as PlanetLab [51]), and in demand for distributed computation on these platforms have led to interest by researchers on market-based approaches to resource allocation.

Resource-sharing cluster systems such as Spawn [173], Popcorn [152], and Tycoon [108] 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.

The outsourced computation system described in Chapter 4 is similar to public-resource computing systems such as Distributed.net [67] and BOINC [29], which parcel and distribute computation to vast armies of volunteer users. BOINC provides scientific projects such as SETI@Home [163] and Rosetta@Home [156] 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 [171, 131]. 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 [131, 144]. 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.

2.3.4 Cryptographic programming languages and libraries

The work presented in Chapter 7, which enables the implementation zero-knowledge proofs and e-cash protocols using a new programming language and interpreter, bears many similarities to FairPlayMP [22] (and its predecessor, FairPlay [122]), which provides a language-based system for secure multi-party computation. These protocols allow multiple parties to jointly compute a function on private inputs while revealing nothing but the resulting value.

At the heart of FairPlayMP is a programming language, SFDL 2.0 (short for Secure Function Definition Language), that allows programmers to specify a multi-party computation. The authors provide a compiler that transforms SFDL programs into boolean circuits, and an engine that securely evaluates these circuits and distributes the resulting values among the involved parties. Although this is a very useful tool, it uses generic circuit techniques, and thus from an efficiency standpoint it is often more desirable to develop a multi-party computation scheme specific to the intended application.

IBM's Idemix project [34, 24] has independently developed a library for zero-knowledge proofs and anonymous credentials using Java; their library provides a system for obtaining, proving, and verifying anonymous credentials for use in a privacy-preserving identity systems. While Idemix and ZKPDL/Cashlib both provide implementations of anonymous credentials and CL signatures, our focus on efficient, repeated executions of e-cash transactions has led us to pursue our language-based strategy and develop a performance-optimized interpreter, unlike the Idemix implementation. The CACE project, independent of our efforts, has also designed a high-level language for zero-knowledge protocols; their work has focused on a compiler that can output implementation and LaTeX code from these descriptions [12, 11], and automatically check the soundness of compiled protocols using theorem proving techniques [2].

There are also compilers available [26, 14] for the generation of proofs of security and correctness for cryptographic protocols. While this is an interesting and important area of research, these tools largely focus on static analysis of protocols rather than performance. Perhaps more similar to ZKPDL, the languages Cryptol [112] and Stupid [109] provide a simple interface for developing low-level implementations of cryptographic primitives (such as hash functions) which can then be analyzed and translated into native code on different platforms.

Chapter 3

Currency for P2P systems

This chapter focus on how to design a peer-to-peer system that uses endorsed e-cash. The main application considered here is how peers might use e-cash to *buy* and *barter* for blocks of data in a file-sharing system like BitTorrent. However, e-cash accounting techniques are more widely applicable to other distributed systems, and this chapter also outlines how endorsed e-cash may be applied to applications such as distributed lookup, computation, storage, and onion routing.

This chapter is organized as follows. Section 3.1 reviews the requirements on a P2P currency and introduces *endorsed e-cash*. The application of endorsed e-cash to BitTorrent-like systems is described in 3.2. Protocols for buying and bartering data are presented in Section 3.3. In Section 3.4 practical measures are described to mitigate the performance costs of e-cash, with preliminary simulations to measure their effectiveness. Economic issues introduced by our currency-based design are discussed in Section 3.5. Other applications of e-cash to distributed systems are described in Section 3.6.

3.1 Currency requirements

First, an appropriate currency to incentivize peer-to-peer systems must be chosen. The currency must be *fungible*: a user must be able to be paid for service and spend this payment for service from any other user. The payment protocol should ensure a *fair exchange* of money for the content: either the seller gets paid and the user gets the content, or neither of them gets anything. Users should be able to spend the currency *anonymously*: there should be no way to link an e-coin to the user that spent it, even if the bank colludes with all sellers. The currency must also be *unforgeable*: users should not be able to forge money (or at least be caught and punished when they try). Finally, we want this currency to be *efficiently implementable*.

One currency that satisfies all these requirements is *e-cash* [45, 46], which offers the following properties:

- **Anonymity.** E-cash ensures that it is impossible to trace an e-coin to the user who spent it, even when the bank colludes with all the sellers. It is not even possible to tell *if* two different e-coins were spent by the same user. The only exception is if a user tries to *double-spend* an e-coin. In this case, the bank can learn the identity of a cheater (and in some cases even trace *all* e-coins the dishonest user ever spent), and punish him accordingly.

- **Unforgeability.** A dishonest user might try to spend the same e-coin more than once. The simplest solution to this problem is to use *on-line* e-cash: the seller consults the bank during every transaction to ensure the promised e-coin is new. However, this approach places a heavy burden on the bank and compromises the privacy of the user (if a fair exchange fails, new e-coins can be linked to old ones), and thus is not preferable.

  In *off-line* e-cash, a buyer and seller perform a transaction without consulting the bank. At some later point, the seller can deposit all of the coins he has received. The bank will then check whether any of these coins have been spent before. If a user spends the same e-coin more than once, the bank can use the forged e-coin to identify the user. Note that, at this point, the user has already carried out multiple transactions and obtained more content than to which he is entitled. It is up to the system to devise a punishment sufficient to deter forgery.

  The bank can limit the ability of dishonest users to forge e-coins indefinitely, by publishing an Authorization List of users who are permitted to spend money off-line. The list would in reality be condensed to single value, called an accumulator [39]. Users can prove that they are in the accumulator without revealing their identities. As long as sellers receive frequent updates to the Authorization List, a user would not be able to double-spend many times. If a user double-spends, the bank can simply remove him from the accumulator and punish him (*e.g.*, by banishing his account).

- **Fair exchange.** To be useful, an e-cash scheme must enable users to perform a fair exchange of e-coins for digital content. All fair exchange protocols require a trusted third party (TTP) that is responsible for resolving disputes. Here, we refer to the TTP as the *arbiter*, to emphasize its potential difference from the *bank*. In an optimistic fair exchange protocol, the arbiter is only involved when one party cheats. There has been extensive prior work on fair exchange, however none of the existing protocols is sufficient for file-sharing applications.

  We need to provide a protocol for exchanging an e-coin for a file. Jakobson [98] and Reiter, Wang, and Wright [153] provide protocols for exchanging e-coins for data. However, these protocols are not truly fair exchanges: with both the user loses his e-coin if the exchange fails. Asokan, Shoup and Waidner [7] provide a protocol which allows a user to reuse his e-coin after a failed exchange, but unlinkability is no longer guaranteed for the reused coin: it is possible for a merchant to identify that the same coin is being respent in a second transaction. Furthermore, these protocols all require that e-coins be withdrawn one by one from the bank.

3.1.1 Endorsed e-cash

The best solution that satisfies these requirements is Camenisch *et al.*'s endorsed ecash [42], which allows users to withdraw and store many coins at once, and to respend coins after failed exchanges with the guarantee that these coins will still be unlinkable. However, this protocol has the disadvantage that in case of a conflict, the arbiter must download and verify an entire file. Thus in Section 3.3.3 we modify this protocol so that the arbiter only has to examine a short proof provided by one of the parties.

As generating e-coins is somewhat expensive, we also wish to provide a more efficient protocol for bartering files. There is no existing protocol that addresses this situation. Bao, Deng, and Mao [13] provide protocols for fair exchange of signatures, but not of other forms

of data. Asokan *et al.* [7] provide a protocol for fair exchange of data, but they assume that there is an online verifier who verifies each block of an encrypted file and provides a signature on that file. We provide an efficient protocol which allows two users to exchange files given only that both know the correct hashes on those files in Section 3.3.

In endorsed e-cash, a central bank maintains an account for each user. Users can withdraw a wallet of multiple e-coins from the bank in one efficient operation. Users can spend their e-coins anonymously by performing an efficient fair exchange of e-coins for digital content. However, users must deposit e-coins they have earned back into their bank accounts before spending them again.

Overview An endorsed e-cash transaction proceeds as follows: to spend an e-coin, the buyer chooses a random value, called the *endorsement*, and uses it to encrypt the e-coin to get $coin'$. The buyer gives the seller $coin'$, but retains the *endorsement*. The seller verifies that (1) $coin'$ is a valid unendorsed e-coin and (2) the buyer knows an *endorsement* that will decrypt the e-coin. Once the seller is satisfied, the buyer and seller perform a fair exchange of the endorsement for the content. It is possible to simultaneously exchange several e-coins for multiple goods and services. If a fair exchange fails, the buyer re-encrypts the e-coin with a new randomly chosen $endorsement'$. The new $coin''$ cannot be linked to the original coin, the previous $coin'$, or the user, *unless the coin is double-spent.*

Fair exchange Endorsed e-cash allows us to perform an efficient optimistic fair exchange on e-coins, which is the unique feature of endorsed e-cash. We used endorsed e-cash to create two new fair exchange protocols, for buying and exchanging files, that place a significantly smaller load on the arbiter than prior work. To ensure fairness, the arbiter needs to (1) download the unendorsed e-coin (this requires it to download just one integer) and (2) verify that the seller's supplied content is correct. In the case of the buy protocol, the arbiter randomly tests parts of the file to verify correctness. In the case of the barter protocol, the aggrieved party supplies the arbiter with a short proof that a segment of the file is corrupted. The details of this new, efficient fair exchange protocol using endorsed e-cash is given in Section 3.3.

Enforceable contracts Each endorsed e-coin is explicitly associated with a contract. Only the owner of the e-coin can create the contract. The buyer gives the contract to the seller along with the encrypted $coin'$. The seller can either accept the terms of the contract and initiate the fair exchange protocol, or reject the contract and simply terminate. Endorsed e-cash [42] primarily use the contract to provide some necessary randomness for the e-coin. In this work, we explore how to use this contract to explicitly ensure fairness, via the arbiter. Two new fair exchange protocols in Section 3.3 use specially structured contracts to allow peers to buy and exchange files. In Section 3.6, we explore how to use contracts to enforce fairness in distributed look-up, distributed storage, and distributed computation.

3.2 System design

We now describe a peer-to-peer content distribution system that uses e-cash to provide strong accountability. We draw inspiration from BitTorrent, but strengthen its loose barter accounting with two new protocols allowing a node to *buy* and *barter* encrypted data blocks

from its neighbors. These protocols, detailed in Section 3.3, enable these transactions through the fair exchange of decryption keys for e-coins, or for other decryption keys.

3.2.1 Bank and arbiter

Our design requires the existence of two trusted entities: a *bank*, which provides secure resource accounting, and an *arbiter*, which ensures the fair exchange of e-cash for data. These nodes are trusted to be fair, but otherwise are not trusted with private information, and may operate separately.

The *bank* maintains each user's bank account, handling and validating deposits and withdrawals. The centrally-administered bank is also responsible for administering monetary policy, which we discuss in Section 3.5.2. For our application, a user's accumulated savings represents the amount of data uploaded in excess of that downloaded; it also bounds the maximum number of simultaneous buying and bartering transactions a user may undertake. Like the tracker, the bank cannot link activity between peers. The application of unlinkable currency makes BitTorrent's design—while not originally privacy-preserving—no *less* private by e-cash.

The *arbiter* protects the fair exchange of keys for e-coins by resolving aborted transactions in cases of node failure or intentional misbehavior. (We expect the prospect of future exchanges, especially in a system like BitTorrent that prizes high-bandwidth neighbors so highly, to be a strong incentive against the latter.) The arbiter seems well-suited for distribution: distributing it would require only the system's escrow key and a put/get database, similar to the design the email postage system DQE [174].

The introduction of these two centralized entities, required by our goals of secure accountability, may seem to contradict the fully decentralized nature of some peer-to-peer systems. But it fits our goal of enhancing the scalability of these systems with proper accounting. We note that previous work on incentives and accountability in P2P systems have also relied on centralized entities, whether by providing a central reputation service [25] or employing trusted agent(s) to punish misbehavior [114, 129]. BitTorrent also uses a trusted, centralized tracker to coordinate peer activity. These systems reinforce the view that the value of P2P design is in its scalability, not necessarily in its decentralized nature.

3.2.2 Users

Each user must establish an account with the bank before she can withdraw and deposit e-cash. Rather than provide new users with a starting balance, we look to social networks: new users are invited by friends with existing accounts, who transfer some e-cash from their own bank account to the invitee's account. This technique limits the utility of Sybil attacks [69]; we discuss bootstrapping new users further in Section 3.5.1.

Our protocols require users to deposit each earned coin before it can be re-spent, presenting a potential performance bottleneck. The computational requirements necessary for our cryptographic protocols may weigh heavily on both users and the bank; we describe practical approaches to lessening this burden in Section 3.4.

3.2.3 Obtaining blocks

Suppose Alice requests a block from Bob. Bob encrypts the block using a randomly chosen key, and sends the ciphertext to Alice. Alice responds with an unendorsed e-coin and a contract describing the file she wants. Having spent bandwidth on transferring the ciphertext,

both parties now have an incentive to perform an optimistic fair exchange of the decryption key for the e-coin's endorsement. If Alice tries to avoid paying Bob, then Bob can give his key to the arbiter and prove that it decrypts the ciphertext correctly, as specified by the contract. The arbiter would endorse the e-coin for Bob. If Bob fails to give Alice a proper key, then neither Alice nor the arbiter will endorse the e-coin.

Alice and Bob may also barter blocks when mutual coincidence exists, *e.g.*, when each has received the other's encrypted blocks. To begin, they exchange unendorsed e-coins, along with the coins' endorsements encrypted under escrow. They can now perform fair exchange for decryption keys indefinitely, with the escrowed coin as collateral. Details on both protocols are provided in Section 3.3.

These two protocols are designed to correspond with the unbalanced and balanced exchanges used in a system like BitTorrent. We envision their use as follows: instead of relying on altruism, nodes buy blocks at the outset, and later switch to bartering once they have acquired enough data to participate fully in the torrent. We describe in Section 3.4 how the performance benefits of bartering over buying provide sharing incentives. After completing, nodes may continue to sell blocks to earn credit for concurrent or future torrents.

Key to the suitability of our protocols for this use is how our design *decouples data transfer from accounting*: nodes download encrypted blocks and pay for decryption keys later. This allows us to accommodate the "optimistic" behavior inherent in BitTorrent's choking protocol, since a node may optimistically send neighbors encrypted blocks and reasonably expect to be paid for their keys later. A pair of nodes might delay reckoning debts some number of rounds in order to wait for reciprocation and bartering opportunities, or to improve performance by paying for multiple keys with higher-denomination coins. Section 3.4 examines this further.

3.3 The buy and barter protocols

In Section 3.3.1, we present a fair exchange protocol that lets Alice buy a block of a file from Bob. In Section 3.3.2, we present a fair exchange protocol that lets Alice and Bob trade two blocks; we call this process bartering. In both protocols, when users are honest, the arbiter will never be involved. If something goes wrong, the aggrieved party asks the arbiter to resolve the dispute. In the barter protocol, this process is straightforward. The buy protocol conflict-resolution is more involved and we describe it in Section 3.3.3.

We minimize the amount of work the arbiter performs so that a few malicious users cannot overload it. Our buy protocol improves on the e-coin fair exchange protocols in Camenisch *et al.* [42] and Asokan *et al.* [7] because the arbiter never has to download the entire block (nor the encryption of it). Instead, the arbiter uses the VerifyKey protocol in Section 3.3.3 to randomly test the correctness of the block. Our barter protocol also improves on the Asokan *et al.* [7] digital content exchange protocol, which requires some trusted party to verify that each block is correct and to sign the encryption of each block together with the commitment to the encryption key. Our protocol only assumes that both parties know the desired hash of each block; thus no trusted signer is required. The arbiter never has to download the block; instead, Alice and Bob bring the arbiter a short proof that the block they got does not match the desired hash. The barter protocol assumes that Alice and Bob maintain a continuous relationship; as a result, Alice and Bob only have to create one e-coin each to initiate a relationship, and can reuse the e-coins indefinitely.

Our protocol requires a symmetric block cipher: we write $\mathrm{Enc}_K(block)$ to denote encrypting *block* with key K; $\mathrm{Dec}_K(ctext)$ means decrypting *ctext* using key K. We assume

that a block is large enough to be divided into chunks and $\text{Enc}_K(\textit{block})$ encrypts each chunk separately. We also use verifiable escrow [43]: $\text{Escrow}_{\textit{Arbiter}}(\textit{buydata}, \textit{contract})$ this encrypts *buydata* under the public-key of the arbiter. The decryption key to the escrow is a combination of the arbiter's secret-key and the *contract*; this lets the arbiter ensure he decrypts the escrow only when the terms of the *contract* have been fulfilled. Anybody who knows the arbiter's public-key and the *contract* can verify that the escrow is valid. Finally, write $\text{Commit}(\textit{buydata})$ to denote a commitment that can only be opened to *buydata*. In practice, these would be implemented as Pedersen commitments [145].

We use Merkle hash trees [127] to create short descriptions of a block (or ciphertext). We write $\text{MHash}(\textit{block})$ to denote a Merkle hash of *block*. A person who knows the entire file can publish $(\textit{chunk}, \textit{proof}, \text{MHash}(\textit{block}))$ to prove that *chunk* is in (or is not in) *block*; *proof* is short, efficiently calculated, and includes the position of the chunk in the block. (We require a collision-resistant hash function h.)

3.3.1 How to buy data

Retrieve metadata from the tracker Alice queries the tracker about a particular file. As in BitTorrent, the tracker provides her metadata describing this file, in the form of hashes for each block of data and a list of users who are interested in the file. The tracker adds Alice to its user list.

Select a neighbor Alice contacts Bob and they determine whether either one has a file the other wants. This is done using the same technique as BitTorrent.

Block acquisition Alice now decides she would like to buy a block of a file from Bob. Assume Alice has acquires $\textit{bhash} = \text{MHash}(\textit{block})$ from a trusted authority (*i.e.* tracker). Alice and Bob will agree on a *timeout* by when Bob must provide Alice with the block. The protocol proceeds as follows:

1. Bob chooses a random key K and sends $\textit{ctext} = \text{Enc}_K(\textit{block})$ to Alice.

2. Alice constructs an endorsed e-coin $(\textit{coin}', \textit{endorsement})$. Alice calculates $\textit{chash} = \text{MHash}(\textit{ctext})$, chooses a random value r and calculates the exchange ID $v = h(r)$. Alice sets $\textit{contract} = (\textit{bhash}, \textit{chash}, \textit{timeout}, \textit{coin}', v)$. She escrows the *endorsement* under the arbiter's public-key: $\textit{escrow} = \text{Escrow}_{\textit{Arbiter}}(\textit{endorsement}, \textit{contract})$. Alice sends Bob $(\textit{coin}', \textit{contract}, \textit{escrow})$.

3. Bob verifies that $(\textit{coin}', \textit{contract}, \textit{escrow})$ is formed correctly. If he is satisfied, he establishes a secure connection to Alice using standard techniques and sends the key K. Otherwise, Bob terminates.

4. If Alice receives a K that lets her decrypt *ctext* correctly before *timeout*, she responds with *endorsement*. Otherwise, Alice waits until *timeout* and then calls $\mathsf{AliceResolve}(r)$ on the arbiter, as in Algorithm 3.3.1.

5. If Bob does not receive a correct *endorsement* before *timeout*, he calls $\mathsf{BobResolve}(K,$ *escrow*, *contract*$)$ on the arbiter, as in Algorithm 3.3.2.

Security We sketch why our protocol ensures a fair exchange. Suppose Alice wants to avoid paying. If Bob calls BobResolve before *timeout*, he is guaranteed to be paid, as long as the unendorsed e-coin is valid (recall that Alice cannot abort the protocol before *timeout*). If the unendorsed e-coin is invalid (either badly formed $coin'$ or incorrect *contract*), Bob would not accept it and terminate in step 4 without giving Alice the key K.

Suppose Bob wants to avoid giving Alice a correct key. If he calls BobResolve after *timeout* he will not get paid. If he calls BobResolve before *timeout*, due to the *contract* associated with the *escrow*, he can only get paid if he deposits the correct key K. Alice can then retrieve K at her convenience.

3.3.2 How to barter for data

We present a protocol that lets Alice and Bob perform a fair exchange of two files. The exchange proceeds in two phases. First, Alice and Bob give each other an unendorsed e-coin and an escrow of the endorsement. This establishes a collateral that an aggrieved party can collect if something goes wrong. In the second phase, Alice and Bob perform a fair exchange of the file. If the exchange fails, the wronged party can ask the arbiter to endorse the e-coin. As long as Alice and Bob are honest, they can continue in a bartering relationship indefinitely using the same e-coin as collateral.

Suppose Alice has $block_A$ and she wants $block_B$ which is owned by Bob. They both get $hash_A = \text{MHash}(block_A)$, $hash_B = \text{MHash}(block_B)$ from a trusted authority (*i.e.* the tracker). They perform the exchange as follows:

1. Alice chooses a new signing key $(sk_{\mathcal{A}}, pk_{\mathcal{A}})$ and gives $pk_{\mathcal{A}}$ to Bob. Bob does the same, responding with $pk_{\mathcal{B}}$.
$$A \to B: \quad pk_{\mathcal{A}}$$
$$B \to A: \quad pk_{\mathcal{B}}$$

2. Alice creates an endorsed e-coin $(coin'_A, endorsement)$ and calculates
$$escrow_A = \text{Escrow}_{Arbiter}(endorsement, contract),$$
where the *contract* states that the arbiter can endorse $coin'$ for anyone who presents some $contract'$ that is (1) signed by $pk_{\mathcal{A}}$ and (2) whose terms are fulfilled. Alice gives $(coin'_A, escrow_A)$ to Bob, who performs the corresponding operation and gives Alice $(coin'_B, escrow_B)$.
$$A \to B: \quad (coin'_A, escrow_A)$$
$$B \to A: \quad (coin'_B, escrow_B)$$

3. Alice calculates a ciphertext $ctext_A$ and a commitment to the decryption key $K'_A = \text{Commit}(K_A)$. Alice gives $(ctext_A, K'_A)$ to Bob. Bob similarly computes $(ctext_B, K'_B)$ and gives it to Alice.
$$A \to B: \quad (ctext_A, K'_A)$$
$$B \to A: \quad (ctext_B, K'_B)$$

4. Alice and Bob both compute
$$contract' = (pk_{Arbiter}, pk_{\mathcal{A}}, K'_A, \text{MHash}(ctext_A), hash_A, pk_{\mathcal{B}}, K'_B, \text{MHash}(ctext_B), hash_B).$$

This contract states that to collect collateral, one of two conditions must be met: (1) the owner of $pk_{\mathcal{B}}$ can prove that the opening of K'_A does not decrypt a ciphertext corresponding to MHash($ctext_A$) to a plaintext corresponding to $hash_A$, or (2) the owner of $pk_{\mathcal{A}}$ can prove that the opening of K'_B does not decrypt a ciphertext corresponding to MHash($ctext_B$) to a plaintext corresponding to $hash_B$. This can be proved using standard techniques from Merkle hashes.

5. Alice gives Bob her signature on $contract'$ and Bob gives Alice his signature on $contract'$.

$$A \rightarrow B: \quad sig_A(contract')$$
$$B \rightarrow A: \quad sig_B(contract')$$

6. Alice and Bob execute a fair exchange protocol where Alice gets K_B (the opening of K'_B) and Bob gets K_A (the opening of K'_A). This can be done using Asokan *et al.* fair exchange [7].

$$A \leftrightarrow B: \quad K_B, K_A \quad \text{(fair exchange)}$$

7. If K_B does not decrypt $ctext_B$ correctly, Alice goes to the arbiter with the signed $contract'$, $escrow_B$, K_B, a proof showing that $ctext_B$ did not decrypt correctly, and a proof that she knows the secret key corresponding to $pk_{\mathcal{A}}$. The arbiter would give Alice the endorsement to Bob's e-coin and his signature on the e-coin. Alice can bring the endorsed e-coin to the bank and deposit it in her account. Bob would do the same if K_A is incorrect.

Note: Showing that $ctext_B$ does not decrypt correctly can be done efficiently. Alice gives the arbiter the signed $contract'$, K_B, and a *chunk* that does not decrypt correctly. The arbiter can check that K_B is the promised opening of K'_B. Then the arbiter can test if (1) *chunk* is in $ctext_B$, (2) MHash($ctext_B$) is in $chash_B$ and (3) $\text{Dec}_{K_B}(chunk)$ is *not* in $hash_B$.

Steps 1 and 2 of the protocol only have to be done once to establish a bartering relationship between Alice and Bob. Subsequently, Alice and Bob can perform steps 3–7 to exchange a block. The bartering protocol has more efficient conflict resolution. If Bob cheats Alice, Alice can show the arbiter which chunk decrypted incorrectly. As a result, (1) conflict resolution is more efficient and (2) a cheating Bob is caught with *overwhelming probability*. Finally, we note that, bartering two files is more efficient for the users than executing two purchase protocols; this is because the Asokan *et al* [7] fair exchange only has to be performed once instead of twice (per block). This is true even if Alice and Bob decide to preserve anonymity by using new signing keys and e-coins each time they exchange files.

The main security challenge is to ensure that Alice cannot deposit the e-coin she put up for collateral (and that Bob cannot deposit his collateral e-coin). We have to make an assumption about the endorsed e-cash deposit protocol: the bank can verify the contract associated with an endorsed e-coin. Specifically, the bank will have to verify that the arbiter signed the e-coin (the arbiter's signing key is included in the contract, so the bank does not have to know the arbiter's identity in advance). As a result, to deposit the e-coin under $contract'$, Alice has to get the arbiter's signature. Alice cannot enforce clause (1) of the contract because she does not know Bob's secret key. Alice can enforce clause (2) only if Bob cheats, in which case she is entitled to get her e-coin back. The only other option Alice has is to deposit her e-coin under a new contract. If Alice does this, and Bob later deposits

the endorsed e-coin under the old contract, the bank will see that Alice double-spent an e-coin. Due to the construction of endorsed e-cash, if the same e-coin is deposited under two different contracts, the bank can trace the owner of the e-coin. Thus Alice cannot deposit her own collateral e-coin unless Bob cheats. The same argument shows that Bob also cannot deposit his own collateral e-coin unless Alice cheats.

3.3.3 How to resolve disputes

We give a protocol that lets a seller prove to the arbiter that he provided the buyer with the correct ciphertext and decryption key.

Recall that when Bob calls BobResolve in Algorithm 3.3.2, he has to prove to the arbiter that he has provided it with the correct key. Specifically, he has to show that K decrypts $ctext = c_0||c_1||\ldots||c_n$ to $block = b_0||b_1||\ldots||b_n$, where $ctext$ is in $chash$ and $block$ is in $bhash$. Bob and the arbiter execute VerifyKey, as shown in Algorithm 3.3.3.

Algorithm VerifyKey will detect if chunk c_i does not decrypt correctly. We call such a chunk *corrupted*. If Bob corrupts an nth fraction of the chunks, and the arbiter verifies k chunks, then the arbiter will catch Bob with probability $1 - (1 - n)^k$. Suppose Bob corrupts 10% of the chunks. To catch Bob with 90% probability, the arbiter needs to check 22 chunks; to catch Bob with 80% probability, the arbiter needs to check 16 chunks. This approach might not deter a *malicious* Bob who just wants to perform a denial of service attack. However, a *selfish* Bob who wants to try to get paid for a bad file block would be deterred.

3.4 Scaling by bartering

The requirements of anonymity and security that we have placed on our currency and associated protocols impose computational and communication overheads. This section describes these overheads, and explains how they can be mitigated by careful system design.

Our currency system requires the bank to perform expensive verification for each e-coin presented for deposit, performing roughly as much work as the seller in the buy protocol. This high transactional overhead implies that a distributed, load-balanced bank service would be required to meet the demands of a large system. A mixed hardware-software approach [58] also offers a way to accelerate each bank node's performance.

Still, it is clear that to lessen the load on the bank and users alike, we should also consider reducing the number of buy-sell exchanges (and thus deposits) performed in the system. One may begin by selecting a large block size, lowering the number of transactions required, to the detriment of fine-grained accounting. This provides a guideline for uses in other applications: using e-cash to account for a very large number of small transactions may not yet be practical.

Our barter protocol, described in Section 3.3.2, sidesteps this potential bottle-neck by avoiding the bank altogether. Bartering requires expensive computation only for the first exchange, when both parties create escrow coins, a task comparable to performing both the buyer's and seller's jobs. Successive bartering between the same two parties involve only key exchanges, not heavy computation: the bank becomes involved only in the case of a dispute, when one of the parties may deposit the other's escrow coin. As discussed in Section 2.2, BitTorrent's choking mechanism and clustering behavior suggests bartering would be fairly common, and made even more so with a block-level tit-for-tat policy [23] or a choking policy designed for fairness [170].

Algorithm 3.3.1: AliceResolve, run by the arbiter

Input: Exchange ID r (ensures only Alice can resolve)
$v \leftarrow h(r)$;
if $\{v, K\} \in DB$ **then**
    send K to Alice.
end

Algorithm 3.3.2: BobResolve, run by the arbiter

Input: key K, *escrow*, and contract { *bhash*, *chash*, *timeout*, *coin*$'$, v }, all sent by Bob
if *currentTime* < *timeout* **then**
    endorsement $\leftarrow$ Decrypt(*escrow*);
    if *endorsement not valid for coin*$'$ **then**
        return error.
    end
    Run VerifyKey with Bob for K, using (*bhash*, *chash*) from the contract.
    // ensures Alice sent them
    if K *verifies* **then**
        add $\{v, K\}$ to *DB*.
        send *endorsement* to Bob.
    else
        return error.
    end
end

Algorithm 3.3.3: VerifyKey

Arbiter's Input: Two Merkle hashes *bhash* and *chash*, key K
Bob's Input: Ciphertext $c_0||\ldots||c_n$), key K
`Step 1: Arbiter's challenge`
The arbiter sends Bob a set of indices I.
`Step 2: Bob's response`
Bob replies with $(c_i, cproof_i, bproof_i)$ for every $i \in I$, where $cproof_i$ proves that c_i is the Merkle tree corresponding to *chash* and $bproof_i$ proves that $b_i = \mathrm{Dec}_K(c_i)$ is in the Merkle tree corresponding to *bhash*.
`Step 3: Verification`
The arbiter *accepts* the key if Bob responds with valid $(c_i, cproof_i, bproof_i)$ for every $i \in I$, and *rejects* otherwise.

Of course, bartering may only be used when a pair of participants have mutually desirable content. We evaluate the likelihood of these exchanges using the discrete event simulator from [23]. We attempted to replicate the heterogeneous "flash crowd" scenario from their published results, using 1000 nodes split evenly between cable (6 Mbps down; 3 Mbps up), DSL (1.5 Mbps down; 400 Kbps up) and slow DSL (784 Kbps down; 128 Kbps up). The participating nodes exchanged a 400-block file with the help of a 6Mbps seed node.

The simulator provides a complete log of node interactions; we analyzed logs generated from both the default choking policy and the block-level tit-for-tat (TFT) policy. For each piece received, we searched through nearby events, looking for a reciprocating piece in the opposite direction. If blocks were exchanged in both directions during a short window, we assume that the peers involved could have bartered for the blocks in question.

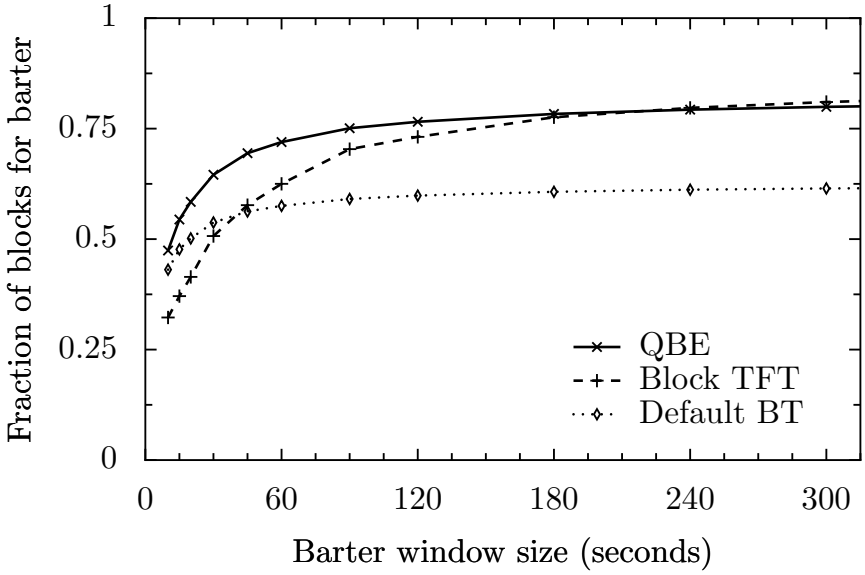


Figure 3.1: Bartering opportunities for different barter windows. Choking policies designed for fairness and clustering produce more opportunities for bartering.

We present the results of our simulations in Figure 3.1, charting the fraction of blocks found eligible for barter (out of a total of 400,000 possible transactions) as we increase the barter window length. Each line represents a simulation run using a different choking policy. We find that with a 30-second barter window (three rounds, the length of an optimistic unchoke), the block-level TFT and default policies reveal roughly equivalent opportunities for bartering, at a little more than half of all transactions. With longer barter windows, the fairer block-level TFT policy produces more opportunities, but not as many as with the quick bandwidth estimation (QBE) scheme [23]. The authors designed QBE to obviate the need for optimistic unchoking (a major source of unfairness), by simulating the effect of instantaneous, accurate bandwidth probes between nodes. This allows nodes to immediately begin unchoking their fastest neighbors, without need for the trial-and-error method of tit-for-tat.

These simulation results indicate that more than half of peer transactions between BitTorrent nodes involve reciprocation soon after. Finding more bartering opportunities requires modified choking policies designed for fairness and clustering; interestingly, the currency overhead aligns greater fairness, through barter, with reduced computational bur-

den. However, achieving these results in practice may be more difficult: we do not simulate node churn, failure, nor do we attempt to integrate currency budgets or debt-reckoning at certain intervals into the choking policy. We leave this for future work.

3.4.1 Batched transactions

Bank deposits may also be reduced by grouping transactions over time: the bank may issue different coin denominations (*i.e.*, \$1, \$2, \$4, etc.), which can be used to pay for multiple blocks at once. The bank must publish different keys corresponding to each coin denomination, so that they may be withdrawn and recognized by all users. A buyer could then negotiate a contract to pay for n decryption keys with $log(n)$ coins. We expect peers may use larger coins once a long-standing relationship has been established (so that in case of a dispute, not many resources would have been wasted), thereby decreasing load on themselves and the bank.

We note that these techniques may apply to other currency systems as well, regardless of the underlying implementation. For example, Karma [172] builds a currency atop a distributed hash table (DHT); as a result, its karma-for-file exchange protocol requires many DHT lookups. Here, the overhead is defined by network latency, rather than computation, but this overhead may nevertheless be similarly reduced by bartering and batching.

3.5 Economic Issues

In a monetized P2P network selfish peers serve as agents in a virtual economy. Peers that behave correctly not only earn money, but add value to the network. As peers join, the economy grows, requiring the creation and distribution of currency. The bank must implement a monetary policy and peers must navigate the marketplace. In this section we consider some simple solutions to these problems.

3.5.1 Entering the Network

Users in a monetized file sharing network make money by sharing the files they acquire. This presents a bootstrapping problem: new users must obtain the resources (either e-cash or files) required to obtain files.

If the bank simply gives new users e-cash, the network becomes highly vulnerable to Sybil attacks [69]. Users will join repeatedly and e-coins will become worthless. A more reasonable option is to give away blocks from a randomly selected file. Unfortunately this places significant demand on the bank's bandwidth, and requires that the bank obtain a steady stream of desirable (and legal) content.

To avoid incentivizing Sybil attacks, we provide new users with neither coins nor files. Instead we assume an existing social network capable of distributing files and e-coins. It is in a P2P network's interest to admit productive users. We expect existing users to give (or lend) e-coins and files to trusted friends.

If a potential user cannot obtain resources through friends, they can also obtain e-coins using some real world resource. The bank may either sell e-coins for real money, or facilitate an auction between new and existing users. Alternatively, users can be given coins when they perform a bandwidth intensive task (for example, users look up webpages and compute hashes, which the bank then crosschecks). If the payoff of this task is small compared to the payoff of sharing files, users will not waste bandwidth to join multiple times.

3.5.2 Creation of Money

In any monetized P2P network, there is a time and cost associated with each transaction. The bank must provide enough e-cash to satisfy the network's demand for simultaneous transactions. If the bank produces too few e-coins, the network cannot operate near full capacity, and if the bank produces too many e-coins inflation potentially leads to a monetary crash [102].

One proposed heuristic for currency creation is to grow the number of e-coins linearly with the number of users [172]. This heuristic ignores the fact that certain users add more value to a network than others. Furthermore, it does not address how new e-coins should be distributed. Simply paying users interest on their bank balances is problematic since it allows wealthy users to continually earn money without participating.

A simple, yet robust approach to wealth distribution is for the bank itself to participate in the network. The bank can inject new e-coins when making purchases, or destroy existing e-coins after making sales. To determine when currency should be created or destroyed the bank can monitor the rate at which files are bought and sold, as well as the rate at which e-coins are withdrawn and deposited. If currency becomes sparse, purchase requests will slow. If currency is overly abundant, purchase requests will increase but finding sellers will become difficult. In either case withdrawal and deposit rates with slow.

A more analytic approach to currency regulation is pursued in [102]. Using a restricted economic model, the authors demonstrate that bank balances (*e.g.* knowledge of wealth distribution) can predict how much currency can be added to a monetized P2P network without resulting in a crash. The model also allows the authors to bound the impact of altruistic behavior and money hoarding. Though their approach is promising, their results assume that all users have equal file-sharing resources (*i.e.* bandwidth, files to share). The authors believe that more realistic assumptions do not fundamentally change their results.

3.5.3 Variable vs. fixed pricing

In our protocol, the price of a BitTorrent block is fixed at one e-coin. This minimizes the overhead associated with purchasing files, but requires the value of an e-coin to remain relatively constant; it also assumes that all content is priced the same per block. A protocol with variable prices permits sellers to correct for inflation and deflation. It also allows buyers to pay more for faster delivery of high-priority blocks (this would be useful for streaming). To enable variable pricing, the bank can produce multiple denominations of e-coins. If multiple coins are involved in a transaction, however, the overhead associated with depositing these coins grows linearly. Furthermore, client behavior becomes much more complex, since prices must be intelligently negotiated.

Our approach also provides incentives for the BitTorrent's "rarest first" heuristic. Even with fixed pricing, the rare blocks are more likely to be sold. Participants have an incentive to buy rare blocks first, making them more common, and helping to solve the problem of blockage at the end.

3.6 Currency for other distributed applications

We believe e-cash can be securely and anonymously applied to many other distributed peer-to-peer systems. This section briefly describes how to make use of e-cash to incentivize such applications. The key requirement for any application of e-cash accounting schemes to other

systems is the formulation of a *contract* describing the resource exchanged in the buy or barter protocol. In order to ensure fair exchange of resources for coins, this contract must be able to concisely describe the resource in case of the need to resolve a claim with the arbiter.

3.6.1 Distributed lookup

In BitTorrent, users contact a centralized tracker for a list of neighbors who are likely to have the file. We want to remove any central trust points (except possibly the bank). We need the system to be efficient: lookup queries should not flood the whole network. We want to introduce incentives to *encourage* peers to help other users find files and to *discourage* unnecessary lookup queries.

One approach is to pay each node along the query path; the interaction between search depth and query price has been studied in random-tree networks [104]. In a fully incentivized DHT, payments would insure an efficient lookup structure and honest responses to queries. Suppose Alice wants to find some file (or a peer responsible for tracking the file). She asks Bob, the closest peer she knows to the file, for the next hop. Alice pays for this information using endorsed e-cash only if the next hop node is at least half-way closer to the file than Bob (which can be checked using hashes), as specified in the contract. Then Alice would contact the next node and repeat the process. Alice pays all peers that help her find the file, encouraging peers to cooperate, and discouraging fake lookups.

3.6.2 Onion routing

The anonymity properties provided by our currency is particularly suited to privacy-preserving systems or anonymous networks, since peers cannot be linked to their e-coins or resource transactions. In onion routing (*i.e.* anonymous remailing) [44, 84, 65] systems, participants route their messages through randomly chosen intermediate peers to disguise the origin and destination of messages. Each router only knows about the next hop, and so the real sender and receiver is hidden. Selfish peers might want to use the system to send and receive messages, but refuse to route other peers' packets.

As an incentive, a router could be paid only if it passes a message to the next router in chain, by using a technique involving threshold endorsed e-cash described by Camenisch *et al.* [42]. They outline a system whereby senders distributes portions of the endorsement needed by each router to redeem its payment to its next- and previous-hop neighbors in the routing chain. This technique would require that in order for a router to get paid, it must perform a fair exchange with both of its neighbors (since each has a portion of an endorsement that the other wants), ensuring message delivery to the final destination (who holds the final endorsement).

3.6.3 Distributed computation

In current distributed computing projects, like Rosetta@Home [156], people voluntarily donate their excess CPU cycles to perform computation-heavy tasks. We can transform this one-way system into a mutually beneficial computing cluster. Users can accept outside jobs when their CPU is not fully loaded, and pay other users to perform some computations when they need more resources.

Suppose Alice wants some computation-heavy task to be done (although I/O-heavy tasks may also be considered). We propose that using e-cash can provide an incentive for

other uses to not only contribute in this computation but also perform it correctly. For some problems, such as graph coloring (in NP), verifying the correctness of the answer is easy, and so can be a part of the contract. In optimization problems, the contract may specify that the best answer within a deadline will be paid (the most), and again the verification is easy. Unfortunately, this is not true for some other types of problems. We further consider this application in Chapter 4, modeling Alice as a *boss* employing multiple *contractors* interested in performing computational tasks for currency.

3.6.4 Distributed storage

A distributed storage system allows a user to backup her data on another peer's machine. Suppose Alice wants to backup her data, and this will be done on Bob's machine. If Alice pays Bob upfront, then Bob has no incentive to store the data, he already got the money. However, if Alice pays him upon retrieval, then if she decides she no longer needs the data, she would never pay and Bob would have wasted his resources storing her data.

A simple approach might propose that Alice pays Bob upfront, but Bob gives her a *warranty* check in return (via a fair exchange protocol) that contains a Merkle hash of the data (for the arbiter to easily verify without downloading the whole data), an unendorsed e-coin, and an escrow of the endorsement. If Bob ever loses or corrupts the data, the arbiter would decrypt the escrow and pay Alice for her damage. The bank can ensure that Bob always maintains a balance large enough to cover his liability. When Alice gets her data, the check is invalidated.

However, this simple approach does not provide any guarantee to Alice that her data is still in Bob's possession until she finally decides to retrieve all of it later—and by then, if the data is missing or corrupted, she is out of luck. In Chapter 5 we describe how Bob can generate *dynamic proofs of data possession* that concisely prove to Alice (and also the arbiter) that he continues to possess her data. The dynamic nature of these storage proofs allows Alice to efficiently modify, insert, or delete portions of the data she has stored with Bob, just as she can with data on a local filesystem, without an expensive round spent retrieving, recomputing, then re-storing of all her remote data. These storage proofs provide for an unlimited number of random challenges, meaning that Alice can check every hour or every day that Bob still has her data, and pay for this periodic check (representing his continued service as a storage provider) with e-coins.

Chapter 4

Accounting for outsourced computation

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 [163] and Rosetta@home [156], 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 chapter is motivated by efforts to build peer-to-peer systems that rely on cryptographic electronic cash (e-cash) to provide incentives for participation as described in Chapter 3. Such a system prevents free-riding without sacrificing the privacy of its participants. However, the verification of e-coins by the bank is an expensive computational operation, and we wish to offload this work from the bank to the participants.

The naïve 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.

There problems are not limited to e-cash systems, however. Users of the distributed computing network SETI@home have developed their own clients, for both malicious and selfish reasons [131, 144] (see Section 2.3.3). Multi-player games cannot assume that players will not modify their clients to give themselves an in-game advantage.

This 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 4.2, we define a game-theoretic framework to analyze different scenarios.

Section 4.3 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 4.4 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. In Section 4.5, 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. Finally, we evaluate the performance of different settings the boss can set to influence the behavior of contractors in Section 4.6.

4.1 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 the e-cash system described in Chapter 3, peers use e-cash to exchange resources; 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.

4.2 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 4.4.) 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.

$$p = Pr[\mathit{check} \mid \mathit{incorrect}]\ Pr[\mathit{incorrect}]$$

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: $c = Pr[\mathit{check} \mid \mathit{incorrect}]$. 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 4.2.1 *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 ε 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 ε will have negative utility. This means any rational contractor will either perform the job with accuracy at least ε, or will refuse to do the job. ■

Corollary 4.2.1 *Any rational contractor will use the least costly algorithm that provides correct answers with at least ε probability.*

4.3 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 ϵ (*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 4.3.1 *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* $\mathsf{neg} = O(2^{-l})$.

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 4.3.1 (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* $\gamma + (1-\gamma)\mathsf{neg}'$ *(provided* $0 \leq \gamma \leq 1$*).*

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

Theorem 4.3.1 *Let* $\mathsf{cost}(1)$ *be the cost of the original algorithm. Let* D*,* $\mathsf{neg}' < \mathsf{neg}$ *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* $\delta + (1-\delta)\mathsf{neg}$ *(provided* $0 \leq \delta \leq 1$*).*

Proof (of Theorem 4.3.1) 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 $\gamma + (1-\gamma)\mathsf{neg}'$. 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, it has at best neg probability of success. That means that it's total success probability can be at most $\frac{\delta}{\gamma}(\gamma + (1-\gamma)\mathsf{neg}') + (1-\frac{\delta}{\gamma})\mathsf{neg} \leq \delta + (1-\delta)\mathsf{neg}$. ■

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

Theorem 4.3.2 *Suppose that definition 4.3.1 holds for our input distribution. If* $\frac{f}{r} \geq \frac{1}{c}$*, and* $r > \mathsf{cost}(1)$ *and* $c > \mathsf{neg}/(1-\mathsf{neg})$*, 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 4.3.1, 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 $\delta + (1-\delta)\mathsf{neg}$. That means the total utility will be $(\delta + (1-\delta)\mathsf{neg})r - (1-\delta-(1-\delta)\mathsf{neg})cf + (1-\delta-(1-\delta)\mathsf{neg})(1-c)r - \delta\mathsf{cost}(1)$. 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 remainder of this chapter, we can then assume $\varepsilon \cong 0$, and therefore $p \cong c$.

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

4.4 When to check an answer

In Section 4.2, 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.

4.4.1 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.

4.4.2 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 [151].

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 4.4.1 *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 4.4.1 *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 4.2, 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 4.5.

4.4.3 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 4.4.2 *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 $c = (1-h^m-(1-h)^m)+(h^m+(1-h)^m)t = 1-(1-t)(h^m+(1-h)^m)$. ■

4.4.4 Hiring two rational contractors

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.

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 4.3 for how to use hashing to set q arbitrarily close to 0.

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

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 4.4.3 *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

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

■

4.5 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).

4.5.1 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 4.5.1 *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 4.5.2 *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 4.5.1 *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 4.5.1 *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.

4.5.2 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 4.5.2 *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 4.5.3 *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 4.5.3 requires the following Lemma. We omit the proof, which follows from the Binomial Theorem and basic algebra.

Lemma 4.5.1 *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 4.5.3) 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 $pgm/(p+d)$. ■

4.6 Evaluation

Throughout this 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 4.2 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

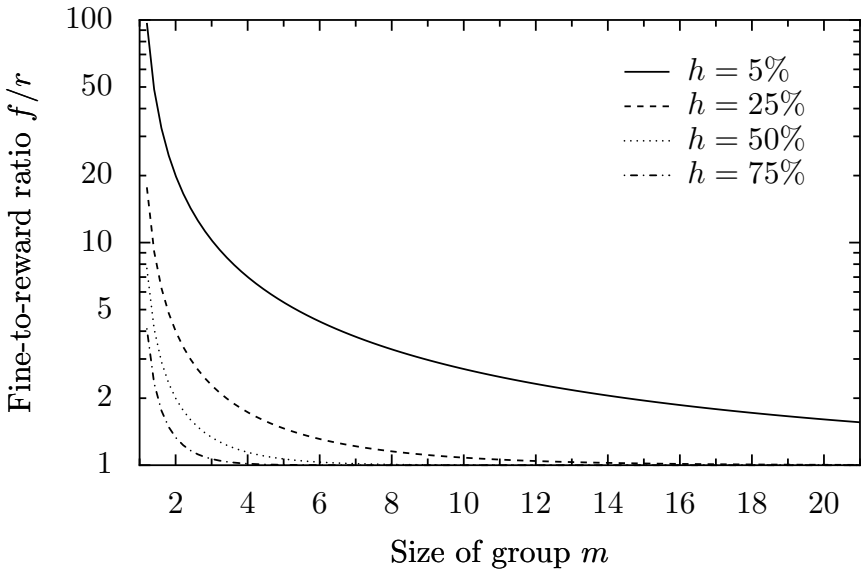


Figure 4.1: Example parameter settings for f/r and m that provide valid incentives assuming a fraction h of honest users. (Theorem 4.4.1)

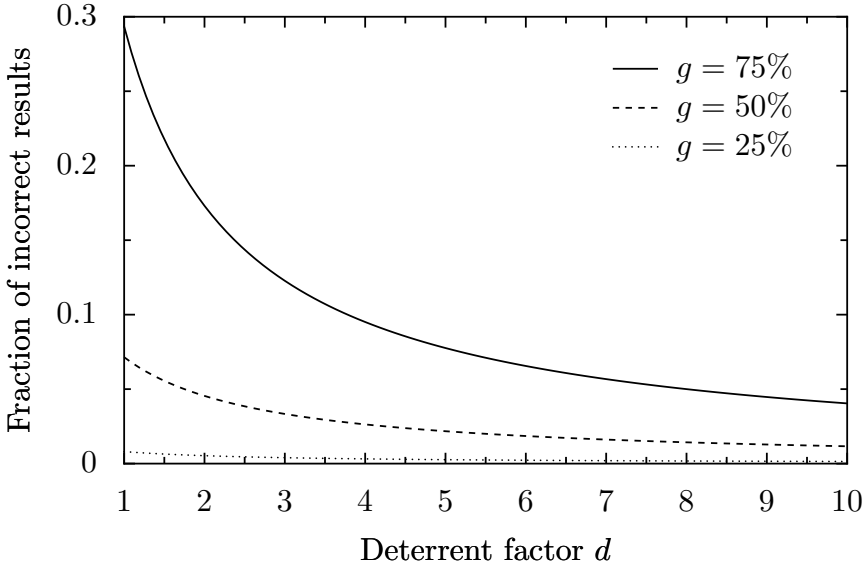


Figure 4.2: The maximum fraction of incorrect results that the boss will accept due to a fraction g of malicious contractors, for different settings of the deterrent factor d. (Corollary 4.5.1)

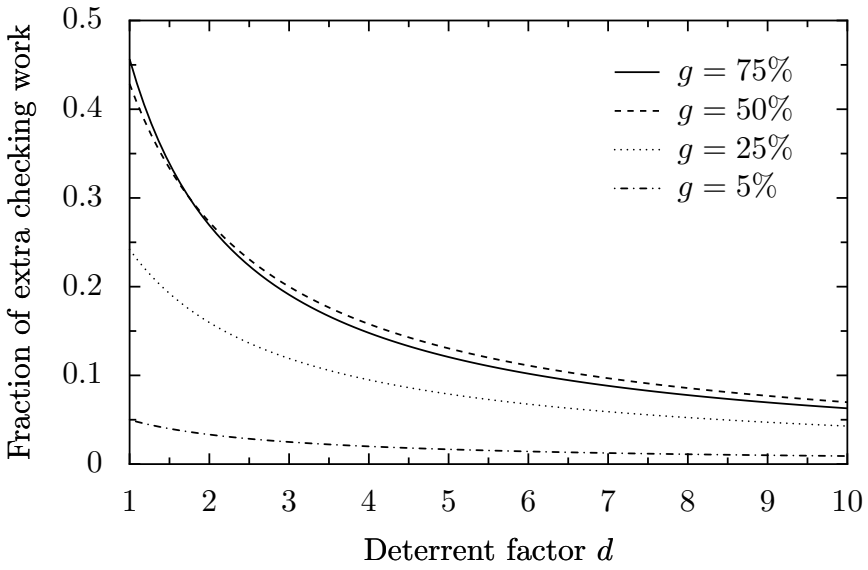


Figure 4.3: The maximum amount of extra double-checking work that a group of malicious colluders controlling a fraction g of all contractors can force the boss to perform, for different settings of the deterrent factor d. (Theorem 4.5.3)

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

4.7 Summary

This chapter has 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 4.4.2), or when the contractors cannot collude in large scale in the long run (Sec-

tion 4.4.4). Finally, we have shown that using these 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 the 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 future work.

Chapter 5

Accounting for outsourced storage

In storage systems, the node responsible for storing a client's data is not necessarily trusted; this is especially true in peer-to-peer storage systems, where malicious or greedy nodes may not be interested in providing reliable service to others. Therefore, users would like to check if their data has been tampered with or deleted. However, outsourcing the 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 [168] cannot be directly applied in this scenario).

Ateniese et al. [8] have formalized 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.

However, PDP and related schemes [8, 68, 99, 164] apply only to the case of static, archival storage, *i.e.* a file that is outsourced and never changes. 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 [101, 115], database services [121], and peer-to-peer storage [106, 132].

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 DPDP 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 [88]). This scheme yields a provable outsourced versioning system (*e.g.,* CVS), which is evaluated in Section 5.7 by using traces of CVS repositories of three well-known projects. The main contributions of this chapter are summarized as follows:

1. A formal framework for *dynamic provable data possession* (DPDP);
2. The first efficient *fully dynamic* PDP solution;
3. 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
4. Practical applications of the DPDP constructions to outsourced file systems and versioning systems (*e.g.,* CVS, with variable block size support);
5. An experimental evaluation of this 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 $O(\log n)$; 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 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, we could use a dynamic Merkle tree (e.g., [113, 134]) 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 [141] but not for Merkle trees.

5.1 Related work

The PDP scheme by Ateniese et al. [8] 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, 138], linear storage for the client [162], or do not provide security guarantees for data possession [161]. Note that using [8] in a dynamic scenario is insecure due to replay attacks. As observed in [70], 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) [99], 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

Scheme	Server comp.	Client comp.	Comm.	Model	Block operations				Probability of detection
					append	modify	insert	delete	
PDP [8]	$O(1)$	$O(1)$	$O(1)$	RO	✓				$1-(1-f)^C$
Scalable PDP [9]	$O(1)$	$O(1)$	$O(1)$	RO	✓*	✓*		✓*	$1-(1-f)^C$
DPDP	$O(\log n)$	$O(\log n)$	$O(\log n)$	standard	✓	✓	✓	✓	$1-(1-f)^C$

Table 5.1: Comparison of PDP schemes. 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 a constant, i.e., independent of n. In all constructions, the storage space is $O(1)$ at the client and $O(n)$ at the server.

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 [164], but their solution is also static (see [68] for a summary of POR schemes and related trade-offs).

Ateniese et al. have also developed a dynamic PDP solution called Scalable PDP [9]. 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. 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 [70] would be violated), they provide a protocol with optimal asymptotic complexity $O(1)$ in all complexity measures giving the same probabilistic guarantees as the DPDP scheme. Lastly, their work is in the random oracle model whereas DPDP is provably secure in the standard model (see Table 5.1 for full comparison).

Finally, proofs of possession are closely related to memory checking, for which lower bounds are presented in [70, 135]. Specifically, in [70] 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 from DPDP. Note that for schemes based on cryptographic hashing, an $\Omega(\log n)$ lower bound on the proof size has been shown [53, 169]. Related bounds for other primitives have been shown by Blum et al. [28].

5.2 Model

We build on the PDP definitions from [8]. 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 sk *and a public key* 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* 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* 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* $\mathsf{PrepareUpdate}$*. 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* $\mathsf{PrepareUpdate}$ *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;*

- $\mathsf{Verify}(\mathsf{sk}, \mathsf{pk}, M_c, c, P) \rightarrow \{\mathsf{accept}, \mathsf{reject}\}$ *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 [8] does not store any metadata at the client side. Also sk, 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 $M_c = M'_c$.

[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).

Retrieval of a (part of a) file is similar to the challenge-response protocol above, composed of Challenge, Verify, Prove 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. We also note that a PDP scheme is consistent with the DPDP scheme definition, with algorithms PrepareUpdate, PerformUpdate and 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 [8] has the same algorithm definition for key generation, defines a restricted version of PrepareUpdate that can create the metadata for only one block at a time, and defines Prove and Verify algorithms similar to our definition. It lacks an explicit definition of Challenge (though one is very easy to infer). PerformUpdate consists of performing a full re-write or an append (so that *replay* attacks can be avoided), and 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 [8, 68]. Note that the restriction to the PDP scheme gives a security definition for PDP schemes compatible with the ones in [8, 9].

Definition 5.2.2 (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 an extractor that can extract (at least) the challenged parts of the file by resetting and challenging the adversary polynomially many times.*

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* $\mathsf{KeyGen}(1^k) \rightarrow \{\mathsf{sk}, \mathsf{pk}\}$ *and sends the public key* 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* info *specifying what kind of update to perform (see Definition 5.2.1) and sends these to the challenger. The challenger runs* 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* 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* 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* 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*, which is 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 creates a challenge using the algorithm* $\mathsf{Challenge}(\mathsf{sk}, \mathsf{pk}, M_c) \rightarrow \{c\}$ *and*

sends it to the adversary. The adversary returns a proof P. If $\mathsf{Verify}(\mathsf{sk}, \mathsf{pk}, M_c, c, P)$ *accepts, 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. Overall, the goal is to extract (at least) the challenged parts of F from the adversary's responses which are accepting.*

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 [68, 99, 164], 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 Verify algorithm should not make use of the secret key. This could be accomplished with a signature scheme using the skip list roots but is not elaborated on further here.

5.3 Rank-based authenticated skip lists

In order to implement our first DPDP construction, we use a modified authenticated skip list data structure [89]. 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 [127] 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., [113]). Yet, such algorithms have been extensively studied for the case of the authenticated skip list data structure [141]. Before presenting the new data structure, we briefly introduce authenticated skip lists.

The authenticated skip list is a skip list [148] (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.

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 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 [89], 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

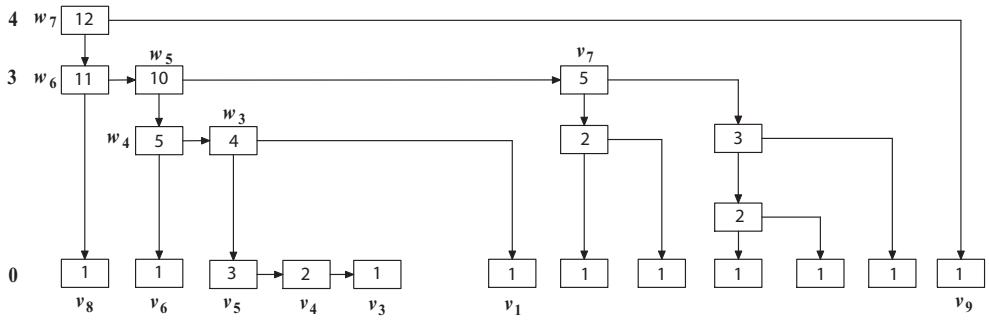


Figure 5.1: Example of rank-based skip list.

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.

5.3.1 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 high and 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 [89]. 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).

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.

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: $l(v) = 0$

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

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 $+\infty$.

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 $\mathcal{T}$;

5.3.2 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).

5.3.3 Verification

After receiving from the server the representation $\mathcal{T}$ of block i and a proof Π for it, the client executes Algorithm 5.3.2 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 Π = (A_1, ..., A_k), where A_j = (l_j, q_j, d_j, g_j) for j = 1, ..., k;
2: λ_0 = 0; ρ_0 = 1; γ_0 = T; ξ_0 = 0;
3: for j = 1, ..., k do
4:    λ_j = l_j; ρ_j = ρ_{j-1} + q_j; δ_j = d_j;
5:    if δ_j = rgt then
6:       γ_j = h(λ_j, ρ_j, γ_{j-1}, g_j);
7:       ξ_j = ξ_{j-1};
8:    else {δ_j = dwn}
9:       γ_j = h(λ_j, ρ_j, g_j, γ_{j-1});
10:      ξ_j = ξ_{j-1} + q_j;
11:   end if
12: end for
13: if γ_k ≠ M_c then
14:    return reject;
15: else if ρ_k − ξ_k ≠ i then
16:    return reject;
17: else {γ_k = M_c and ρ_k − ξ_k = i}
18:    return accept;
19: end if
```

Algorithm 5.3.2 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 ξ_j. If the returned block representation $\mathcal{T}$ and proof Π 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 δ_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 ξ_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.

5.3.4 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 Π'. 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 Π' 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 [141]. 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 in Algorithm 5.3.3 the update algorithm performed by the server (performUpdate) and in Algorithm 5.3.4 the update algorithm performed by the client (verUpdate). Input parameters $\mathcal{T}'$ and Π' 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 by performUpdate is $O(\log n)$ whp.

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

```
if upd is a deletion then
    set j = i − 1;
else {upd is an insertion or modification}
    set j = i;
end if
set (T', Π') = atRank(j);
if upd is an insertion then
    insert element T in the skip after the i-th element;
else if upd is a modification then
    replace with T the i-th element of the skip list;
else {upd is a deletion}
    delete the i-th element of the skip list;
end if
update the labels, levels and ranks of the affected nodes;
return (T', Π');
```

To give some intuition of how Algorithm 5.3.4 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

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 verify(j, M_c, T', Π') = reject then
7:     return reject;
8: else {verify(j, M_c, T', Π') = accept}
9:     from i, T, T', and Π', compute and store the updated label M'_c of the start node;
10:    return accept;
11: end if
```

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 5.3.4 for the update "insert a new block with data $\mathcal{T}$ after block 5 at level 1".

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 addition of one more block.

5.4 DPDP scheme construction

In this section, we present our DPDP 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 [89, 142].

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

- $\mathsf{KeyGen}(1^k) \to \{\mathsf{sk}, \mathsf{pk}\}$: 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;
- $\mathsf{PrepareUpdate}(\mathsf{sk}, \mathsf{pk}, F, \mathsf{info}, M_c) \to \{\mathsf{e}(F), \mathsf{e}(\mathsf{info}), \mathsf{e}(M)\}$: 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 $\mathsf{PrepareUpdate}$, 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 5.3.3 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 $\mathsf{e}(\mathsf{info})$). Note that the index j and the type of the update upd is taken from $\mathsf{e}(\mathsf{info})$ and the new data $\mathcal{T}$ is $\mathsf{e}(F)$. Finally, Algorithm 5.3.3 outputs M'_c and $P_{M'_c} = \Pi(j)$, which are output by $\mathsf{PerformUpdate}$. The expected runtime is $O(\log n)$ whp;

- $\mathsf{VerifyUpdate}(\mathsf{sk}, \mathsf{pk}, F, \mathsf{info}, M_c, M'_c, P_{M'_c}) \rightarrow \{\mathsf{accept}, \mathsf{reject}\}$: 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 5.3.4 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 5.3.4. It runs in expected time $O(\log n)$ whp;

- $\mathsf{Challenge}(\mathsf{sk}, \mathsf{pk}, M_c) \rightarrow \{c\}$: 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 $O(C)$;

- $\mathsf{Prove}(\mathsf{pk}, F_i, M_i, c) \rightarrow \{P\}$: 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;

- $\mathsf{Verify}(\mathsf{sk}, \mathsf{pk}, M_c, c, P) \rightarrow \{\mathsf{accept}, \mathsf{reject}\}$: 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 5.3.2 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 [8]. 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 [30] as in Compact POR [164].

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 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 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 [8]. 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 [8] 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) 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 5.2.2;*
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 [148].

5.5 Security

In this section we prove the security of our DPDP scheme. We begin with the following lemma, which follows from the two-party authenticated skip list construction (Theorem 1 of [141]) 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.*

To prove security, we are also using the *factoring 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 5.5.1 (Security of DPDP protocol) *The DPDP protocol is secure in the standard model according to Definition 5.2.2, assuming the existence of a collision-resistant hash function and that the factoring assumption holds.*

Proof The challenger is given a hash function h, and an integer $N = pq$ but not p or q. The challenger then samples a high-order element g. He interacts with the adversary in the data possession game honestly, using the given hash function, and creates and updates the tags while using N as the modulus and g as the base.

Suppose now the challenger challenges C blocks, namely the blocks with indices $i_1, i_2, \ldots, i_C$. We recall that in response to each challenge, the proof contains:

1. The tags $T_{i_1}, T_{i_2}, \ldots, T_{i_C}$ for each block $i_1, i_2, \ldots, i_C$, along with the respective skip list proofs that correspond to each tag $T_{i_1}, T_{i_2}, \ldots, T_{i_C}$;

2. A "weighted" sum of the form $S_1 = a_{i_{11}} b_{i_1} + a_{i_{12}} b_{i_2} + \ldots + a_{i_{1C}} b_{i_C}$, where $a_{i_{1j}}$ $(j = 1, \ldots, C)$ are random numbers known by the challenger.

According to Definition 5.2.2, the DPDP scheme is secure if, whenever the verification succeeds with non-negligible probabilty (i.e., the adversary wins the data possession game), the challenger can extract the actual blocks (which we denote with $m_{i_1}, m_{i_2}, \ldots, m_{i_C}$) in polynomially-many interactions with the adversary. We extract the actual blocks by means of the "weighted" sums sent by the adversary as follows. Suppose the challenger challenges the adversary for a polynomial number of times and gets C verifying responses. Then if $S_1, S_2, \ldots, S_C$ are the weigthed sums sent each time, we have the following equations:

$$\begin{aligned} S_1 &= a_{i_{11}} b_{i_1} + a_{i_{12}} b_{i_2} + \ldots + a_{i_{1C}} b_{i_C} \\ S_2 &= a_{i_{21}} b_{i_1} + a_{i_{22}} b_{i_2} + \ldots + a_{i_{2C}} b_{i_C} \\ &\vdots \\ S_C &= a_{i_{C1}} b_{i_1} + a_{i_{C2}} b_{i_2} + \ldots + a_{i_{CC}} b_{i_C} \end{aligned}$$

where $a_{i_{j1}}, a_{i_{j2}}, \ldots, a_{i_{jC}}$ for $j = 1, \ldots, C$ are different sets of random numbers sent each time with the challenge and $b_{i_1}, b_{i_2}, \ldots, b_{iC}$ are the blocks that the adversary claims to possess. By solving this system of linear equations we extract the blocks $b_{i_1}, b_{i_2}, \ldots, b_{iC}$. We recall that the actual blocks are denoted with $m_{i_1}, m_{i_2}, \ldots, m_{iC}$. Since all the responses verified we have that for all $j = 1, \ldots, C$ the following statements are true:

1. $T_{i_j} = g^{m_{i_j}} \mod N$, whp. Otherwise the adversary can break the collision resistance of function h by Lemma 5.5.1;
2. $T_{i_1}^{a_{i_{j1}}} T_{i_2}^{a_{i_{j2}}} \ldots T_{i_C}^{a_{i_{jC}}} = g^{S_j} \mod N$, which by the linear system equations can be written as

$$T_{i_1}^{a_{i_{j1}}} T_{i_2}^{a_{i_{j2}}} \ldots T_{i_C}^{a_{i_{jC}}} = (g^{b_{i_1}})^{a_{i_{j1}}} (g^{b_{i_2}})^{a_{i_{j2}}} \ldots (g^{b_{i_C}})^{a_{i_{jC}}} \mod N\,. \tag{5.1}$$

Suppose now there is a subset of challenged blocks $\{b_1, b_2, \ldots, b_k\} \subseteq \{b_{i_1}, b_{i_2}, \ldots, b_{iC}\}$ such that $b_j \neq m_j$ for all $j = 1, \ldots, k$, i.e., we extract some blocks that are not the actual blocks. Let $a_1, a_2, \ldots, a_k$ and $T_1, T_2, \ldots, T_k$ be the random numbers and the tags respectively that correspond to some response (i.e., to some linear equation of the system) for blocks $\{b_1, b_2, \ldots, b_k\}$. Equation 5.1 can be written as

$$T_1^{a_1} T_2^{a_2} \ldots T_k^{a_k} = (g^{b_1})^{a_1} (g^{b_2})^{a_2} \ldots (g^{b_C})^{a_C} \mod N\,, \tag{5.2}$$

since for the remaining blocks $b_j = m_j$ and $T_j = g^{m_j} \mod N$ is the correct tag with high probability, and therefore the expressions referring to them are cancelled out. Now, for Equation 5.2 to be satisfied, we have

$$g^{a_1 m_1 + a_2 m_2 + \ldots + a_k m_k} = g^{a_1 b_1 + a_2 b_2 + \ldots + a_k b_k} \mod N\,,$$

while $a_1 m_1 + a_2 m_2 + \ldots + a_k m_k \neq a_1 b_1 + a_2 b_2 + \ldots + a_k b_k$. This means that the adversary can find $A \neq B$ such that $g^A = g^B \mod N$, which means that $A - B = k\phi(N)$ and therefore $A - B$ can be used to factor N, by using Miller's Lemma [130].

Therefore, the exracted blocks must be the correct ones. Otherwise, the adversary can either break the collision resistance of the function used, or factor N. ■

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 $((n-t)/n)^C$.

5.6 Extensions to outsourced storage 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.

5.6.1 Variable-sized blocks

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.

5.6.2 Directory hierarchies

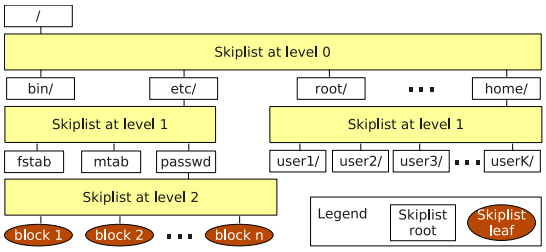
We can also extend our DPDP scheme for use in storage systems consisting of multiple files within a directory hierarchy. The key idea is to place the start node of each file's rank-based authenticated structure (from our single-file scheme) at the bottom node of a parent dictionary used to map file names to files. Using key-based authenticated dictionaries [141], 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(a)).

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.

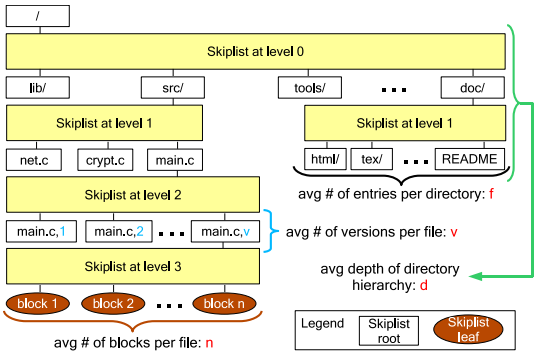
Since the start node of the authenticated structure of the directory hierarchy is the bottom-level node of another authenticated structure at a higher level in the hierarchy, upper levels of the hierarchy must be updated with each update to the lower levels. Still, the proof complexity stays relatively 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.

5.6.3 Version control

We can build on our extensions further to efficiently support a versioning system (*e.g.,* a CVS repository, or versioning filesystem). Such a system can be supported by adding another additional layer of key-based authenticated dictionaries [141], keyed by revision number, between the dictionaries for each file's directory and its data, chaining proofs as in previous extensions. (See Figure 5.2(b) 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



(a) A file system skip list with blocks as leaves, directories and files as roots of nested skip lists.



(b) 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 [4]: the complexity of these proofs will then be $O(\log n + \log v + d \log f)$.

Figure 5.2: Proving directions of files and versioned resources atop the DPDP scheme.

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 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 [4] 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.

5.7 Performance evaluation

We evaluate the performance of our DPDP 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 [8], where a server wishes to prove possession of a 1GB file. As observed in [8], 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.

5.7.1 Proof size

The expected size of proofs of possession for a 1GB file under different block sizes is illustrated in Figure 5.3. 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 [8], [2] and is thus represented by the line labeled PDP. The distance between this line and those for our DPDP 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.

5.7.2 Server computation

Next, we measure the computational overhead incurred by the server in answering challenges. Figure 5.4 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 [8] 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.

5.7.3 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 5.6. Using CVS repositories for the Rsync [157], Samba [157] and Tcl [139] 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 [9]) 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.

[2] 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.

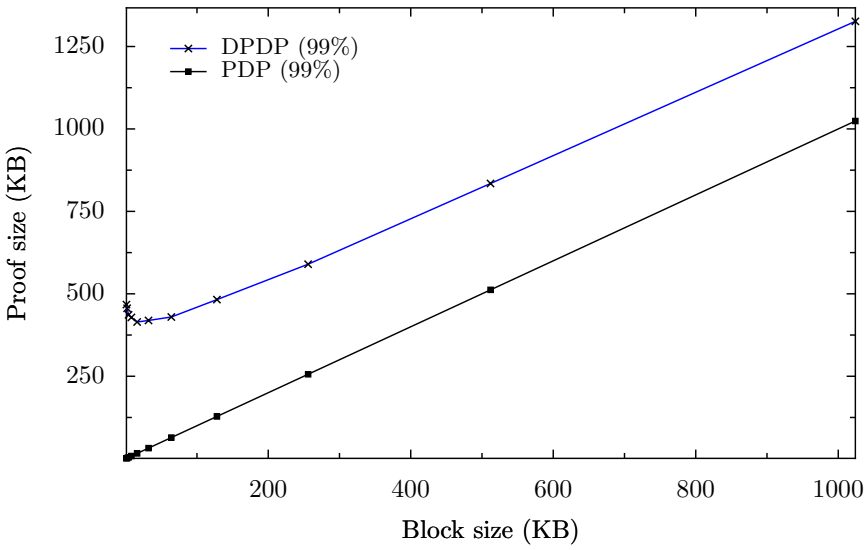


Figure 5.3: Size of proofs of possession on a 1GB file, for 99% probability of detecting misbehavior.

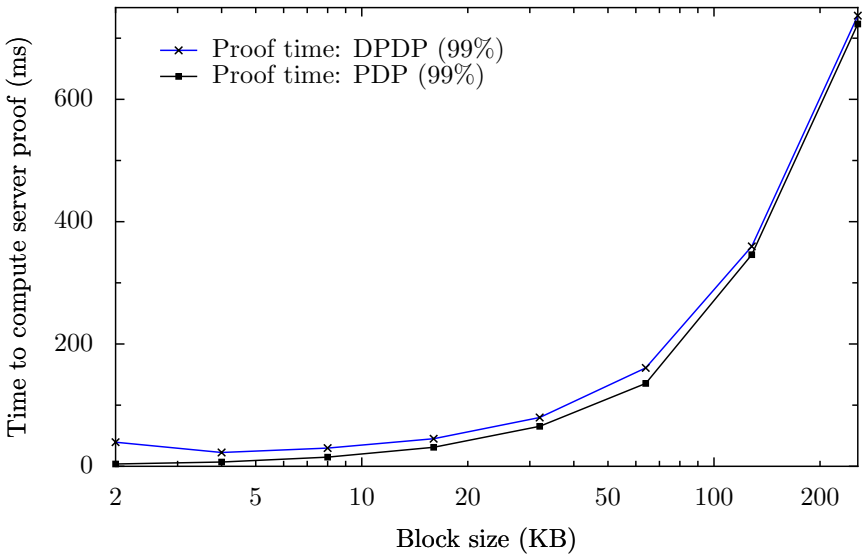


Figure 5.4: Computation time required by the server in response to a challenge for a 1GB file, with 99% probability of detecting misbehavior.

	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
Fraction of total size	8.4%	6.7%	3.4%
Max v	44123	9880	59664
Max proof size	2003 KB	4632 KB	4075 KB

Table 5.4: Authenticated CVS server characteristics.

Table 5.4 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 scheme: as described in Section 5.6, 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 5.4 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.

Chapter 6

Accounting for bandwidth

As outlined in Chapter 1, systems designed in a peer-to-peer (P2P) style aim to take advantage of the benefits of decentralized architecture. These benefits include the ability to *self-scale* as new participants join, with new peers contributing resources to offset the added workload they generate. Additionally, decentralization offers improved fault-tolerance and user privacy, as no central authority is responsible for orchestrating or recording peer interactions.

However, this decentralized, semi-anonymous structure also makes it difficult to enforce policies over clients (software agents acting on behalf of their users) in order to promote even greater long-lived scalability, fault-tolerance, and fairness. Though peer-to-peer systems rely on the contributions of peers, when faced with free-riders who consume more resources than they provide to others, some systems (such as BitTorrent) must rely on altruistic members to provide adequate performance.

The system described in this chapter, FairTrader, addresses this problem by enforcing strict fairness among peers. FairTrader uses e-cash to allow participants to earn points in exchange for work they contribute and spend their points to obtain service from others.

The use of e-cash offers a solution to many issues relating to identity and accountability that are common in peer-to-peer systems. For example, Sybil attacks [69] exploit the use of many identities to receive extra benefit from a system. In a currency-based system, as long as creating a new identity does not provide the attacker with more currency, Sybil attacks provide no economic benefit and lose their meaning.

Further, e-cash provides a strong and persistent accounting mechanism which incentivizes desirable *long-term* behavior, since clients earn credit for future benefit. BitTorrent uses a simpler "tit-for-tat" scheme that is both gameable [146, 119, 100] and concerned only with the present. Once the desired file has been obtained, a peer gains no benefit from continuing to upload. This lack of fungible credit linking disparate files is especially problematic when a participant joins a new BitTorrent swarm: since the new node has no data, it is ignored by other participants. Only after receiving blocks from an altruistic peer can the new peer participate normally.

Fairness due to stricter accounting of peer activity with currency also helps to yield a more robust system. In today's BitTorrent systems, a large fraction of upload bandwidth is contributed by a few well-provisioned, altruistic peers [146], while many free-riding peers consume significantly more resources than they contribute. This imbalance leads to two problems. First, the performance of such imbalanced swarms is critically dependent on a few participants: their failure or departure hurts performance for the remaining majority of peers. Second, observed performance for individual members may be highly variable: only

those participants lucky enough to be chosen by the fast, altruistic peers are well served.

The benefits of fairness and accountability provided by FairTrader come at a cost, however, as the use of the cryptographic protocols requires semi-centralized components, the *bank* and the *arbiter*, described in Chapter 3. We say they are "semi-centralized" because each can be widely distributed, but their services are not provided by peers. These components pose a scaling challenge, but we have taken care to reduce their overhead and demonstrate that large swarms can still be supported. We also ensure that these components do not weaken user privacy: our protocols allow a client to be linked to its P2P interactions only if it has acted maliciously by attempting to double-spend a coin.

Outline This chapter presents the FairTrader design, described in Section 6.1, which demonstrates a solution to the free-riding problems of BitTorrent-like systems, as well as the problem of maintaining a persistent reputation across swarms. In doing so, we increase the overall fairness of the system (i.e., peers are served in proportion to their own contributions); we also do so in a way that is privacy-preserving. In particular, we show how e-cash accounting, described in Section 6.2, can be added to a system with an existing incentive scheme to ensure accountability while still preserving privacy of the participants in the P2P system.

We also present an implementation of the protocols and architecture of FairTrader. In Section 6.5 we demonstrate that our system is feasible for practical application, and that its currency-based approach offers benefits for both short- and long-term performance.

6.1 Design

FairTrader builds on the existing block-transfer mechanism of BitTorrent, while layering on top of it the the cryptographic accounting needed for e-cash and fair exchange. We describe our design, both in terms of the changes required to add currency to BitTorrent and the strategy changes that must take place in order for peers to effectively use our new system.

6.1.1 Goals

Motivated by the problems outlined in the previous section, FairTrader's chief design goal is to increase fairness — which we believe provides better long-term incentives and improved reliability — through a currency-based credit system that persists throughout each swarm in which a peer participates. Fairness guarantees that seeders of past files will get proportional or preferential treatment in future files; i.e., they will see better performance relative to non-contributors. Achieving fairness solves the seeding and free-riding problems described in the previous section.

We introduce persistent fungible credit that allows peers to exchange past work (for example, as a seeder) for future resources, which means that they will no longer enter new swarms without their history to boost (or lower) their reputation. Our credit system ensures that a peer's good behavior in one swarm can be used to benefit another.

FairTrader specifically intends to enforce *long-term* fairness, rather than force users into short-term tit-for-tat exchanges, as BitTorrent does today. Short-term fairness is sufficient when users have simultaneous demands and comparable capacity — e.g., in a flash crowd for a popular file — but does not provide strong incentives for long-lived applications. A peer joining a BitTorrent swarm for last week's most popular file is likely to find that most downloaders who acquired it last week are now devoting their resources to something new,

since no incentive exists for them to continue uploading the old one. In fact, peers will receive the best service for downloading today's files by dedicating all of their network capacity to uploading just those files. Our use of a *fungible*, currency-based incentive mechanism is motivated by a "long view" of system performance as spanning multiple, disparate swarms over weeks or months, including concerns for the future availability of data and continued performance for new peers.

In pursuit of fairness and fungibility, FairTrader does not sacrifice the privacy of users. While many systems, including BitTorrent, do not make privacy guarantees, it is important that our scheme not weaken the existing level of peer privacy. Privacy-preserving or anonymity-providing system designs would be poorly served by an accounting mechanism that required online access to a centralized entity during peer interactions or allowed a third party to observe or infer peer interactions.

In order to meet these three goals of fairness, fungibility, and privacy, we use protocols for electronic cash and fair exchange that are provably secure against fraud, i.e., double spending and/or counterfeiting coins, and do not require access to a centralized bank during each interaction (cryptographically speaking, this means we are using *offline* e-cash). We have altered and augmented these protocols to better suit their use in a P2P application. For example, a buyer cannot lose the key sent to them by the sender (either through intentional maliciousness or network failure), and similarly a sender cannot lose the coin sent to them by the buyer.

Although the use of offline e-cash does take some of the burden off the bank, FairTrader still requires coins to be withdrawn and deposited with a bank. Furthermore, a sender must deposit any earned coins with the bank rather than spending them himself, as otherwise coins grow excessively in size [49]. In addition, fair exchange protocols, in which two parties exchanging data are guaranteed that either both or neither of them will get what they want, cannot be achieved without the presence of some trusted third party (in our case, the arbiter) to intervene in the case of failure [140]. In summary, both the bank and the arbiter are inherently centralized entities. Still, the work of these entities can in fact be distributed across multiple machines (with little coordination required), or outsourced to potentially untrusted third parties [17].

6.1.2 Currency-enabled BitTorrent

Our e-cash and fair exchange protocols follow the structure depicted in Figure 6.1. Clients hold accounts with the bank, and use them to withdraw wallets consisting of coins; the coins in these wallets are then used to buy and barter for blocks of data. If an exchange fails (e.g., by someone trying to cheat or an inopportune network failure), the buyer or seller can turn to the arbiter to resolve the failure. If an exchange is successful and a buy operation took place, the receiving party will later deposit the coin with the bank in order to receive credit. The anonymity property of e-cash guarantees that the bank is unable to link a spent coin to the original user who withdrew the coin, unless that user attempts to cheat by spending the same coin twice.

To adapt this currency system to the design of BitTorrent, we first consider a strawman design in which an e-cash transaction is used to account for each block exchange in BitTorrent. In such a design, every block transfer in one direction could be paid for by a coin sent in the opposite direction. A fair exchange protocol would be used to ensure that both or neither of the transfers occur.

This simple design has several drawbacks, due to the overhead of the cryptographic

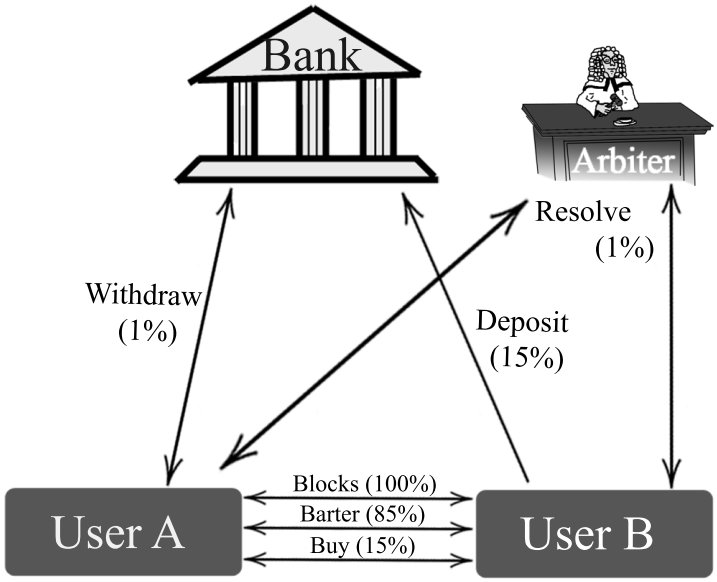


Figure 6.1: The FairTrader system components. Users A and B will exchange encrypted blocks; if both users have blocks that the other one desires, A and B may barter. Otherwise A will buy the desired block from B. Following a buy operation, B can deposit the earned coin with the bank. If a problem occurs at any point in the exchange, either party can go to the arbiter. The percentages seen illustrate reasonable expectations for the relative frequency of these operations; they are quite subject to change, however, as they will be highly dependent on system characteristics such as peer failure rates and the percentage of peers that are actively downloading (i.e., not just seeding).

protocols involved. First, e-cash transactions are computationally expensive, both for the peers involved and for the bank. While the burden on peers is distributed, the bank is a centralized entity that must eventually accept a deposit for each coin that is exchanged; this design would present an impractically high computational overhead for the bank.

One approach to reducing CPU overhead for the bank would be to increase the block size, thus reducing the number of transactions required to download a file. However, this solution does not match the typical usage of BitTorrent, in which users make many requests for small blocks. For example, a large block size would be inconvenient when buying from a slow peer, as the peer downloading the block would be prevented from selling his received data, and additionally crediting the peer for BitTorrent's tit-for-tat mechanism, until the entire block was finished (this is to ensure that the block is not corrupted, since its hash can be checked only after the whole block is downloaded). Furthermore, large blocks would also cause problems for the arbiter since it is sometimes required to decrypt and transmit an entire disputed block.

To alleviate these drawbacks, FairTrader decouples block transfer from accounting. BitTorrent is used, nearly unmodified, to disseminate small encrypted blocks among peers; the process of data transfer can now be considered separately from the process of obtaining the decryption key needed to get the content within the block. Once an encrypted block is downloaded, peers later acquire the decryption keys for the block using a fair exchange protocol; this significantly reduces the overhead of the protocol, as the fair exchange now involves trading coin in exchange for the decryption key value (both comparably small), as opposed to the entire block. Again, the cryptographic protocols in place prevent the users from being cheated, as the Buy protocol protects the buyer from paying for an invalid key through the use of enforceable contracts.

To further reduce the cryptographic overhead of for typical peer interactions, we introduce the Barter protocol, which allows peers to exchange keys for keys (as opposed to a key for a coin in the Buy protocol) if mutual interest exists; i.e., when peers have sent each other encrypted blocks. A barter exchange places a coin in escrow to protect the initiator of the barter transaction, but does not require the coin to be "spent" (and therefore deposited) as long as both parties provide valid keys. This escrow coin can be re-used for future barter operations, thus reducing overhead even further. In the common case, the Barter protocol eliminates most of the cryptographic overhead for peers and for the bank; for more on this, see Section 6.2.

Finally, FairTrader reduces overhead by issuing e-coins in multiple denominations; e.g., one denomination for each power of two. A coin of a larger denomination is the same size as a coin of a smaller denomination, meaning the computational overhead the same for each coin; this allows clients to buy or barter for a large number of recently-received blocks (e.g., 512 blocks) at fixed intervals with a constant overhead for each interval, regardless of the amount of data being transferred. At the last interval, the remaining balance of N blocks can be paid using only $O(\lg N)$ coins.

6.1.3 Payment strategy

By decoupling block transfer from accounting, our aim is to modify the existing tit-for-tat "choking" strategy that BitTorrent clients employ in the short term as little as possible. While we do not modify the way BitTorrent peers download and upload encrypted blocks, we do influence client strategy by periodically "settling debts"; i.e., by buying or bartering for the keys to any encrypted blocks received since the last settling operation. When settling

debts, our strategy is to barter for as many blocks as possible, and afterwards, if any unpaid blocks remain, to use the biggest single coin by denomination (nearest power of two) to buy the majority of remaining blocks. This ensures that the overhead due to our currency accounting remains constant at each settling step.

FairTrader modifies interactions with seeders more extensively. Under BitTorrent's incentive scheme, seeders provide purely altruistic service to downloading peers. FairTrader rewards seeders for their contributions in the form of currency payments; this allows a peer entering a new swarm to begin obtaining blocks more quickly than before. Our design does not prescribe that FairTrader clients buy blocks solely from seeders and avoid all other interactions, though: we believe the performance benefits due to bartering, as described above, offer significant incentives for peers to continue seeking tit-for-tat partners, when possible, that will reduce the overhead involved in buying.

6.1.4 Centralized components

The service provided by a bank is well-suited for distribution: all it has to do is store a balance for each account holder (identified by public key) and a short record of each coin deposited; it does not even need to record completed withdrawals. As such, a bank requires no administrative relationship with BitTorrent's centralized tracker, nor with the arbiter that resolves aborted transactions, so FairTrader clients may select any combination of these three entities for a given exchange or swarm.

Our bank is logistically similar to the deployment of private trackers in some BitTorrent communities, described in detail in Section 2.2.6. Private trackers measure and enforce upload/download ratio requirements on peer activity across multiple files [5]. While effective, these tracker ratio schemes rely on easily forgeable, self-reported client statistics; with e-cash, on the other hand, uploads by peers in excess of the amount downloaded are securely accounted for by each user's bank balance without loss of privacy.

The proliferation of private trackers suggests that different banks, each issuing its own currency and enforcing its own policy, might be well-suited to the state of BitTorrent file-sharing communities today. Many private trackers restrict their membership to those invited by well-performing peers, but offer up-front credit to new users in the form of relaxed ratio requirements (i.e., below 1.0), which become stricter as the user continues to participate. FairTrader banks could implement a similar policy by granting e-cash to new users. Of course, this sort of system requires external controls on membership, or strong identities, to avoid Sybil attacks.

The ratio schemes and invitation restrictions employed by private trackers serve as a defense against Sybil attacks: even though new members receive instant benefit by joining, the number of invitations is restricted to prevent users from receiving this benefit repeatedly. A design based on an open, unrestricted bank would have to be careful not to provide any starting balance to users, and instead rely on gifts of currency from existing users, or an opportunity to perform work for the system at the outset.

Finally, dealing with fraud is also an important consideration for the bank. While the bank can detect double-spending after the fact, no offline e-cash mechanism can stop a malicious user from the simultaneous use of the same coin in multiple transactions. In this case, the deposit of a duplicate coin would reveal enough information to identify the malicious spender, and the bank could punish the user. A simple punishment is account revocation; if new accounts can be created effortlessly, however, then this would open the door to Sybil attacks. Throttling or otherwise limiting the number of new accounts (through

confirmation by email, SMS, credit card, social networks [176], etc.) represents one tactic to avoid this problem. Another approach is to obtain a "high value secret" or monetary deposit for each account that could be revealed or retained in case of bad behavior. E-cash supplies the mechanism to support such policies, and FairTrader allows the coexistence of multiple banks, allowing further innovation in this area.

In addition to the bank, FairTrader requires another semi-centralized entity: the arbiter. The arbiter facilitates fair exchange by serving as a fallback mechanism to resolve aborted transactions; its protocols are described in Section 6.2. It serves as the trusted third party (TTP) required by fair exchange, and its role is limited to decrypting escrowed keys and coins left by peers that disconnect before finishing the Buy or Barter protocol. To establish itself as an arbiter, it must simply publish a public key for peers to use for encrypting escrow messages, and record aborted transactions it resolves; it is also therefore well-suited for distribution. A distributed arbiter might, for simplicity, share a single public key; otherwise, peers might draw from a list of available arbiters before beginning an exchange. Note that under our system, a peer gains no benefit from refusing to send the last message of an exchange, a sthe message is small and the arbiter will provide whatever information is missing anyway; rather, it is in the peer's best interest to retain goodwill with the peer in order to receive further blocks. We therefore expect the load on the arbiter to be low, and due only to unexpected node failure and churn.

6.2 Cryptographic Protocols

FairTrader builds upon the protocols for endorsed e-cash for buying and bartering described in Chapter 3, and also on the improved barter protocols of Küpçü and Lysyanskaya [107]. Here we provide details on these protocols, as adapted for FairTrader.

6.2.1 Withdraw

Withdraw is a two-round protocol between a client and the bank. The client contacts the bank and proves her identity; she can then request the withdrawal of a certain number of coins. If the bank is satisfied that the client is who she says she is, it will respond with a wallet, signed by the bank, that contains the specified number of coins. Because we use *compact e-cash* [36], the size of the wallet is not dependent on the number of coins inside. Cryptographically, the wallet can be thought of as the bank's signature on the seed of some pseudorandom function [82]; the serial number of each coin is then a value generated by the pseudorandom function using this seed. To keep the user's transactions anonymous with respect to the bank, we use *blind signatures* [45, 40], in which the signature on the pseudorandom function seed can be created jointly by the user and the bank but without the bank learning the seed (i.e., the message that it just signed). As a result, the bank cannot link a spent coin when it is deposited later to the user who withdrew it, because it does not know the serial number. Nevertheless, to keep users accountable the coins still contain mechanisms to guarantee that the bank can easily check the validity of the coin, or catch a user who is attempting to spend a coin twice; for details on how this is accomplished, we refer the reader to Camenisch et al. [36, 42] or Belenkiy et al. [18].

To best suit our purposes, as mentioned in Section 6.1.2 we have modified the original withdraw protocol to allow for coins of different denominations. To allow multiple coin denominations, the bank uses multiple public keys; each public key corresponds to a different coin denomination. When a client requests a withdrawal, she specifies the number and values

of the coins she would like to withdraw. If the client has enough money in her account, the bank signs the wallet using the public key corresponding to the selected denomination and adjusts the client's balance accordingly. A seller can easily check the worth of a coin, as its validity will verify only under the corresponding public key of the bank.

Finally, the withdraw protocol is run over a secure communication channel (e.g., SSL) to protect the user's privacy, as an eavesdropping party would be able to learn information about the identity of the user performing the withdrawal otherwise.

6.2.2 Deposit

Deposit is a non-interactive protocol in which the client sends a single message to the bank that serves to both prove her identity (so that the deposit goes into the right account) and deposit her coin(s). Optionally, the bank may acknowledge that the deposit was successful.

Before crediting the client's account, the bank must check each coin to make sure it is valid and has not been spent twice. Although the bank needs to check each coin separately, there are methods for batch-checking some parts of the deposit [40, 37, 74] that would increase the bank's efficiency on deposits of multiple coins. Furthermore, untrusted contractors (i.e., computing devices) can be employed to outsource the checking of e-coins [17]. The deposit protocol is also run over a secure channel so that outside observers do not infer information about the wealth of the client.

6.2.3 Fair block exchange

As mentioned in Section 6.1.2, peers may exchange blocks in two different ways: either through a Buy or a Barter protocol. For our Buy protocol, which allows for the fair exchange of a coin and a block (more precisely, the decryption key for an encrypted block), we adapt the buy protocol of from Section 3.3.1. For our Barter protocol, which allows for the fair exchange of a block for a block, we adapt the barter protocol of Küpçü and Lysyanskaya [107]. Currently, all known optimistic fair exchange protocols make use of an expensive cryptographic primitive known as *verifiable escrow* [43]. In the buy protocol, verifiable escrow is used for buying each block, while in the barter protocol it is used only once for any number of (successful) exchanges between the same peers. Bartering is therefore much more efficient than buying, especially when peers can barter repeatedly (for example, when they are seeking the same content).

6.2.4 Buy

To buy a file from Bob, Alice computes a contract; i.e., a document that says that she (she can be identified by a temporary signing key, so that her true identity remains unknown) promises to give to Bob (whose true identity need not be known either) the contents of an escrow in exchange for a key that will decrypt the ciphertext (identified by its hash alone) to the data promised in some given hash, by some given expiration date. Alice can then sign this contract under her temporary signing key.

Bob, upon receiving this contract, can send to Alice the key; he next receives an "endorsement" from Alice which acts to complete the coin, thus allowing him to submit it for deposit. If Bob does not receive the endorsement, he must contact the arbiter before the expiration date and engage in a resolution protocol (for details on this resolution, see Belenkiy et al. [18]). If the resolution goes well, the arbiter can give the endorsement to Bob using information provided in the contract. If, on the other hand, Alice does not receive the

key prior to the expiration date, she can also contact the arbiter and engage in a resolution protocol to get the key.

6.2.5 Barter

If Alice and Bob choose to barter, then Alice will now be providing Bob with a decryption key of her own rather than a coin. She still needs to include the original escrow (containing the coin), however, as if the exchange goes wrong and Bob does not get Alice's decryption key, he will at least be able to get the coin using the resolution protocol above. But, if all goes well and everyone is honest, then this escrow (which, remember, is quite expensive) will remain untouched and can be kept in place for future exchanges without any additional overhead.

In addition to the escrow on the coin, Alice will also form an escrow on her decryption key; although this is another escrow, it is substantially less computationally expensive than the one containing the coin. The contract will now contain essentially the same information as before, with the addition of this new escrow. Bob can then send Alice his decryption key, and Alice can respond in turn with her own. If something goes wrong with either of these last steps, the parties can resolve with the arbiter as they did before (using slightly different protocols; see Küpçü and Lysyanskaya [107] for details).

We expect that in general peers will prefer to barter rather than buy whenever possible, as bartering offers a way for peers to receive content immediately, as opposed to buying in which they will sell files to receive credit that they will then invariably just use to buy files anyway. Furthermore, the efficiency improvements gained from reusing the expensive coin escrow are quite noticeable; for example, for peers who exchanges tens to hundreds of blocks (which is highly typical [94]), the efficiency gain in bartering over buying is between one and two orders of magnitude (in temrs of both CPU time and bandwidth overhead).

6.3 Implementation

We have developed a prototype implementation of FairTrader which extends the BitTorrent protocol to use e-cash for strong accountability. In this section we describe the implementation of the client, the bank, and the underlying cryptographic operations.

6.3.1 Client

We built the FairTrader client using libtorrent, an open-source C++ BitTorrent client library. Libtorrent uses Boost ASIO to provide fast, multi-platform data transfers for a number of front-end client implementations. Our implementation aims to separate control of block transfer from accounting, by leaving BitTorrent underlying block transfer and short-term incentive mechanisms largely untouched. To accomplish this, we allow libtorrent to manage the transfer of encrypted blocks with as few modifications as possible, and implement our e-cash protocols as BitTorrent extension messages using a plugin interface used by other protocol extensions.

Our main changes to the libtorrent core involve the encryption and decryption of blocks. We encrypt blocks before they are sent to a waiting peer, and interpose decryption at the point when a newly received block is asynchronously queued for hash-checking. From the library's point of view, a block missing an encryption key is still waiting to be hashed by

the disk helper; since its contents has not yet been validated, this ensures that encrypted blocks from one peer are not uploaded to other peers.

We implemented our buying and bartering protocols by adding new BitTorrent extension messages for each round of the buy and barter protocols. The BitTorrent protocol allows peers to announce support for new extension message types, and client developers have used these messages to implement new standards such as peer exchange, DHT support, and encryption. We use our cryptographic library, described in Section 6.3.3, to process the messages and decrypt blocks.

In FairTrader, peers periodically settle debts separately from the underlying peer selection and bandwidth throttling modules of Libtorrent. After each interval a peer barters with each of its peers that it has exchanged blocks with. Then, whichever peer has received more encrypted blocks over the interval will purchase the largest power-of-two blocks that is less than the difference. This strategy yields a constant computational and network overhead per unit time.

Although we use the basic BitTorrent sharing mechanisms to exchange encrypted blocks, some modifications were necessary to account for block purchases. BitTorrent clients make peer selections based on the amount of data they receive from peers. In FairTrader, clients are credited for sending encrypted blocks *or* buying encryption keys. This modification is necessary to keep the underlying block transfers moving along for peers that have a pure buyer/seller relationship.

Our fair exchange protocols require that each block of data bought or bartered be described with a contract, as described in Section 6.2. Both parties must agree on what they are exchanging using a prepublished hash value. We use the metadata in the torrent file for this purpose, which provides a hash for each block. However, the blocks described in a typical torrent file range between 256kB and 2MB, while the subpieces actually transferred between peers in tit-for-tat interactions are typically 16kB. This difference complicates BitTorrent operation—it may be difficult to tell which peer supplied corrupted data if a block hash comparison, which consists of many subpieces, fails. This complication is unacceptable for FairTrader's precise accounting, so we instead chose a 16kB block size to correspond with the size of a tit-for-tat unit.

6.3.2 Bank server

Our bank server is implemented as a single-threaded, event-driven HTTP server which answers requests to withdraw and deposit coins, or register a new user. We chose HTTP because it is well-supported by clients, especially BitTorrent clients, which already connect to the torrent tracker service using HTTP. Our protocols match the nature of HTTP requests well: deposit takes a single round to complete, while withdraw takes two (a session variable is used to match the second round's request to the first).

Due to this simple implementation, the bank can be easily scaled like most web apps, by adding additional server process on available cores of each available machine, making it suitable for deployment on a cluster of servers. A reverse HTTP proxy (in our deployment testing, based on lighttpd) does the work of load-balancing and mapping users to available bank backends. The work of the bank is almost purely computational, and scales nearly linearly with available cores or machines. See Section 6.4 for details.

Deposit We use a SQL database to store bank information. The bank database must store a record of every coin deposited, keyed on each coin's unique serial number, with

each record storing the information required to detect double-spending. Since these serial numbers are chosen at random, this database schema would distribute well on a distributed key-value store or DHT.

Withdraw The database must also store a balance for each account; account-holders identify themselves to the bank solely using a randomly-chosen public key. After the second round of the withdraw protocol has completed and the account's balance has been decremented the correct amount, no record of the withdrawn wallet needs to be stored with the bank.

6.3.3 Cryptographic library

For our cryptographic operations, we use the Cashlib library described in Chapter 7. This library is built on top of a custom programming language, ZKPDL, that allows for the implementation of all the cryptographic primitives required for our application; it additionally uses custom modular multi-exponentiation routines that help to reduce some of the overhead required by our cryptographic protocols.

In addition to the computational savings provided by Cashlib, we chose to use it for a number of other reasons. As an optimization for fair exchange protocols in which large blocks of data may be exchanged, Cashlib supports the purchase of multiple non-contiguous blocks of data by using a compact representation of these blocks; specifically, it creates a Merkle hash [127] of all the blocks and puts the tree root in the contract, instead of having one hash per block. We also found that Cashlib provides ways, using the Boost C++ library [31], to serialize containers such as messages, contracts, and public/secret keys. Using this serialization, our clients can save objects to disk and load them back when needed, and the bank and the arbiter can convert their public keys to files and distribute them to the clients should the need arise.

Finally, we observed that another optimization could be made once we chose to run our cryptographic protocols over secure channels (as recommended by Cashlib). To prevent man-in-the-middle attacks, e-cash constructions require a random value that must be generated jointly by the buyer and the seller for every coin, which would normally require an extra round trip between peers. Instead, we eliminated this round trip by reusing the randomness generated for secure communication.

6.4 Performance of the bank and arbiter

As discussed in Section 6.1.2, our FairTrader client buys or barters keys for encrypted blocks periodically, in multiples of 16kB. Our protocols allow for these blocks to be batched into one exchange using a large-value coin; for illustration, let us assume that a FairTrader client with a 2Mbit/s Internet is exchanging blocks with four peers; this means that every 15 seconds, a peer might exchange about 1MB of data with each peer. By exchanging one coin per peer per interval the size of the message is quite low (38kB, less than 4%) when compared with the amount of data exchanged.

Computationally, once barter setup has completed (which typically requires about half a second, but again is required only once for multiple exchanges), a barter exchange requires only 18ms of overhead (for the peer initiating the exchange; for the other peer the overhead is even lower). Buying, however, takes slightly more than half a second for the buyer and

about half that for the seller; as described earlier, we mitigate this overhead by buying only at fixed intervals, with the highest value coin possible.

Specific benchmarks for the buy, barter, withdraw, and deposit protocols can be found later in the evaluation of Cashlib and ZKPDL, in Section 7.9.

Bank To run our bank, we used an EC2 node of instance type `c1.xlarge`, which provides eight virtual cores (where each core provides performance approximately equivalent to that of a 2GHz processor). With this machine, we were able to consistently accept almost 100 deposits per second, and we observed that this number scaled linearly with the number of cores (as would be expected). To obtain a higher throughput, a single bank could be distributed over multiple machines, and outsourcing computation techniques could be applied to further reduce the load of the bank [17]. We also mention that although multiple banks tend to operate independently (meaning coins withdrawn at one bank may not be accepted by another), this is analogous to the situation in BitTorrent, in which independent private trackers are used to share among a particular set of peers.

Arbiter We also consider the efficiency of the arbiter. A resolution is necessary only when ciphertext and verifiable escrow exchanges are completed, but one of the parties fails to send her key. Belenkiy et al. [18] showed that the 400kB data the arbiter need to download is independent of the size of the blocks transferred, and it provides 99% confidence in the arbiter's decision. In our experiments, the time taken to send the key and check the received key is less than 10ms. In FairTrader, peers exchange keys every 15 seconds. This means that the probability that a random machine failure will occur in a critical section of the protocol is 1/1500. Following Stutzbach and Rejaie [167], we estimate that the average uptime of a peer is roughly 2.5 hours, which means that we can expect to see one failure every $2.5 \cdot 3600 = 9000$ seconds.

In a BitTorrent system with 1.7 million users, $1700000/9000 = 190$ machines will fail every second, but only 190/1500 of those failures will require resolution by the arbiter. This rate means a resolution will be necessary once every 7.9 seconds. If we consider the largest BitTorrent system to date (with 22 million users [105]), then a resolution will be necessary every 0.6 seconds. In either case, all resolutions can be handled by a single arbiter, as the arbiter typically needs between 90ms and 280ms per resolution in a single core.

6.5 FairTrader performance

Our evaluation illustrates the performance benefits and costs arising from the use of e-cash in FairTrader. We compare the performance of FairTrader to the BitTorrent system upon which it is based, and measure the overhead due to our design, which we recall buys and sells encrypted blocks, as opposed to the simple exchange of unencrypted data in BitTorrent.

Afterwards, we demonstrate the fungibility and fairness guarantees enforced by our system, through experiments designed to highlight the benefits they create. We show how FairTrader enforces fairness, by rewarding peers that have made contributions to the system in the past. We also show that FairTrader provides fungibility in its reward mechanism; i.e. the ability to make exchanges between peers using e-cash that would not have been possible with BitTorrent. We demonstrate situations where FairTrader enables cooperation between peers that would be impossible with BitTorrent (because they were involved in different swarms) by uniting incentives across the entire system.

We ran our swarm experiments on the Amazon Elastic Compute Cloud (EC2), using the `m2.xlarge` instance type. Each instance provides two virtual cores, each core providing performance approximately equivalent to a 2.67GHz processor on modern physical machines. We use Dummynet [154] to emulate bandwidth limitations between instances.

6.5.1 Comparisons with BitTorrent

Experiments which attempt to understand the performance of a decentralized network of many participants rely on certain core assumptions about the characteristics of participants. Apart from standard experimental parameters like bandwidth capacity, computational speed, etc., in P2P file-sharing applications assumptions on *client behavior* have a large impact on performance measurements.

For example, consider the "flash crowd" scenario in which many peers begin downloading a file at the same time until it is complete. When should these participants leave the network and discontinue uploading to their peers: right after finishing the download, or at some point afterwards? Should the experiment assume peers are all devoting their total bandwidth capacity to a single swarm, or that users often seek many files at once, participating as a downloader in several swarms at a time while seeding in others?

Altruism The importance of this first question adds *altruism* as an experimental parameter: we define altruism as upload activity that continues after a download has completed. Since under BitTorrent, a peer receives no direct benefit for continuing to upload a file once complete—in fact, the spare upload capacity due to altruism might be better used by other, concurrent swarms in which the peer is involved—the rational choice for is to leave a swarm immediately after the download has finished.

We use this rational, selfish behavior as a baseline measurement for comparison with BitTorrent. We also note that even more selfish behavior is possible with BitTorrent: research implementations exist which game BitTorrent's mechanisms to more selfishly allocate traffic to peers [146], or download while providing no data to others at all [119]; this evaluation does not consider these non-standard clients, though FairTrader's benefits would be greater in such a scenario.

Research has suggested, however, that much of BitTorrent's performance is due to altruistic behavior by a small fraction of peers with high bandwidth capacities [146]. So in practice, it's clear that some peers do persist in a swarm as seeders, whether due to altruism, the influence of ratio-tracking networks, or default client settings. To model this activity, we have also evaluated BitTorrent performance under increasing levels of altruism, defined by setting a target parameter R for each peer. Once a peer finishes downloading a swarm, it compares its upload/download ratio to R, and continues uploading until the target has been met before leaving the swarm.

Multiple concurrent downloads In addressing the second question, we also consider scenarios where peers are involved in multiple swarms at a time. Single-file flash crowd evaluations of BitTorrent performance are common [110, 146, 55] and capture a basic content-distribution use case, but these types of experiments unrealistically consider each file's swarm in isolation, ignoring the pattern of P2P use most prevalent today. (*cite needed*)

Most users of P2P networks do not enter a single file's swarm at the same time as every other peer, and then discontinue use of the P2P network forever after. In practice, many users of BitTorrent and other P2P networks engage in multiple swarms simultaneously,

either as seeder or downloader, and are often continuously engaged with these swarms over periods of days or weeks (as described in Section 6.1).

Further, many popular BitTorrent tracker communities focus less on facilitating "flash crowds" than on providing access to a continuously-available library of content. On these trackers, swarms for popular content may persist for months or years, and user download patterns are much less synchronized as in the flash crowd scenario (e.g., they follow a natural pattern resulting from user interest). In these communities, just as in a library, not every user is mutually interested in the content of every other peer, but rather in a subset of the content available.

6.5.2 Scheduled download experiments

Based on the aforementioned observations, we have constructed three general scenarios for evaluating the performance of FairTrader and BitTorrent. These scenarios are each based on first creating files (torrents) and peers interested in them, then defining a schedule of each peer's arrival to the swarms for the files in which it has interest.

We create a network of N peers, and a set of available files F, with each peer participating in M of the total $|F|$ files (the set F_n, where $|F_n| = M$). At the beginning of the experiment, we label a fraction s of each file's M peers as seeders (meaning these peers are assumed to already have downloaded the content). The other $(1 - s)M$ peers are considered interested downloaders who, during the course of the experiment, will enter the swarm for that file and attempt to download it. To schedule the arrival of peers, we assign for each peer $n \in N$ and each file $f \in F_n$ a start time selected over a window of time T (here, in seconds). Variations of these values induce the settings described as follows.

Flash crowd Studies of flash crowds exercise the classic content distribution scenario where, typically, a single seed publishes a file that becomes suddenly popular. Under a client-server architecture, the sudden onset of a crowd of downloaders can often cause severe performance degradation for all participants; thus this scenario is often employed to demonstrate the effectiveness of P2P systems in mitigating these performance problems. We use parameter settings of $|F| = 1$ and $M = 1$ with different crowd sizes for N, a seeding fraction $s = 1/N$, and time window $T = 0$ to induce this setting.

Our flash crowd experiments imitate the scenario considered by Legout et al. [110], introducing many peers simultaneously interested in a single file. A small number of seeders initially provides the file, but since their upload capacities are just as constrained as their peers, downloaders are forced to interact with their neighbors rather than download exclusively from the seed.

Figure 6.2 demonstrates the performance of FairTrader as well as the overhead of using e-cash for incentives. While download times seem distributed around in roughly the same way, we see a noticeable overhead due to the use of the cryptography and exchange mechanisms required by our protocols. This is due to the way in which FairTrader clients batch buying, selling, and barter operations at fixed debt-settling intervals, *i.e.* by only buying or bartering for keys every 15 or 30 seconds. We show download times for the FairTrader client using two different settle intervals: when a shorter interval is used, performance improves. However, this need to procure keys before blocks can be re-sold prevents FairTrader clients from being able to quickly upload incoming blocks as soon as they is available, as BitTorrent can. For the single-file flash crowd scenario, BitTorrent is simply able to distribute blocks at a faster rate.

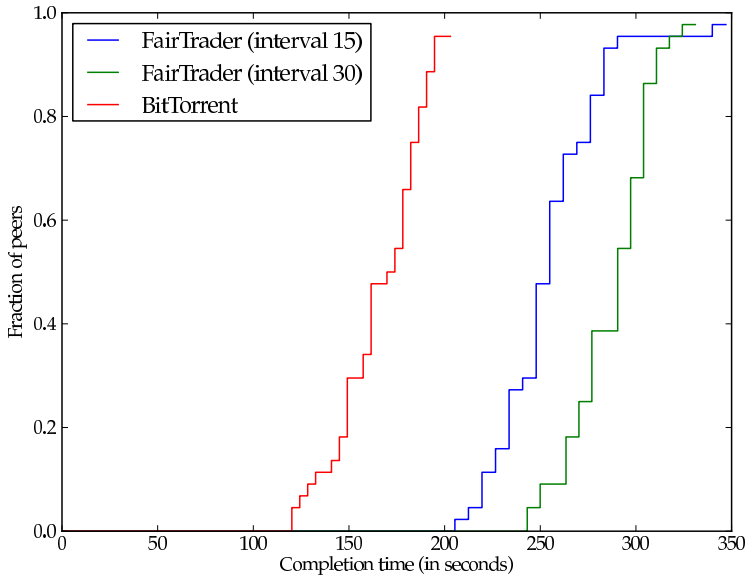


Figure 6.2: Cumulative distribution of download times for the flash crowd scenario. Here, $N = 49$, $F = 1$, and $s = 0.1$, and file size 100MB. Download time is measured in seconds, on the x axis.

Multiple concurrent swarms For these scenarios, we have performed experiments that demonstrate peers engaged in multiple concurrent downloads, using parameters that choose larger sets of files such as $|F| = 5$ or $|F| = 10$, with $N = 20$ or $N = 49$, and $N = 99$, and distributed interest among peers in these files to small subsets such as $M = 2$, and $M = 3$. Like the steady-state scenario, we have chosen arrival times at random over a time window of $T = 400$.

To capture the benefit of fungible credit, we run our long-term experiments for many files. Each peer begins this scenario with a "schedule" comprising a subset of the available files that they would like to download. To better compare average download times, we set each file to be of identical size, and provide each peer with a random file that it can seed at the beginning of the experiment. We require that FairTrader clients continue to upload data until they have earned back their initial balance.

Figure 6.3 shows the distribution of download times for each file downloaded: here, each of the 98 peers are engaged in download swarms for 3 files chosen randomly from a set of 10 files, with each file 20 megabytes in size. Peers begin downloading each file at randomly-chosen points within the 400-second arrival window T. We see that FairTrader's download performance is now more competitive with BitTorrent than it was in the single-file case, which does not provide fungible incentives between swarms for different files. However, this scenario bears some similarity to the flash-crowd case, since at any moment there are typically only several seeders available for each file, limiting the number of nodes that new downloaders may contact to begin purchasing and then re-selling blocks.

Figure 6.4 starkly demonstrates the difference between the incentive systems provided by FairTrader and BitTorrent. In both experimental settings, both systems' P2P clients may leave an individual swarm only when two conditions are met: first, that it has completed downloading the file, and second, that it has reached a 1.0 ratio for that file ("ratio"

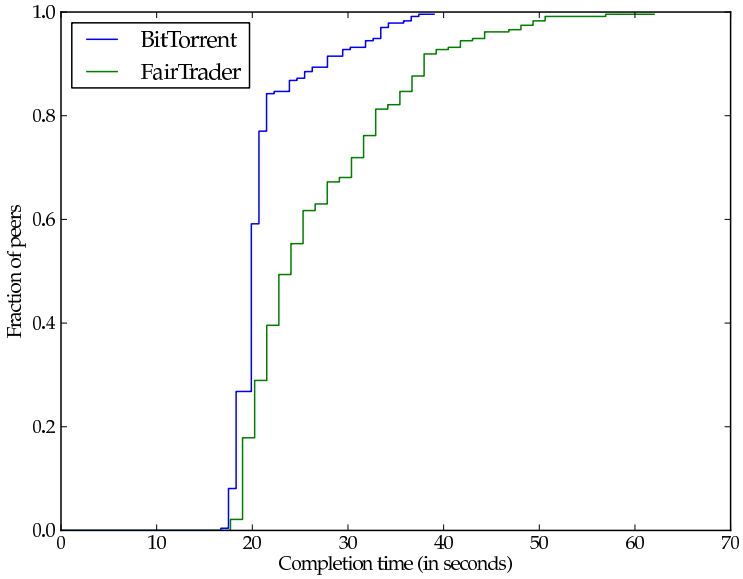


Figure 6.3: Cumulative distribution of download times for the multi-swarm scenario using $N = 98$ nodes, $|F| = 10$ files, $M = 3$, $s = 0.2$, and peers staggering their arrival times in each torrent over a time interval spanning $T = 400$ seconds.

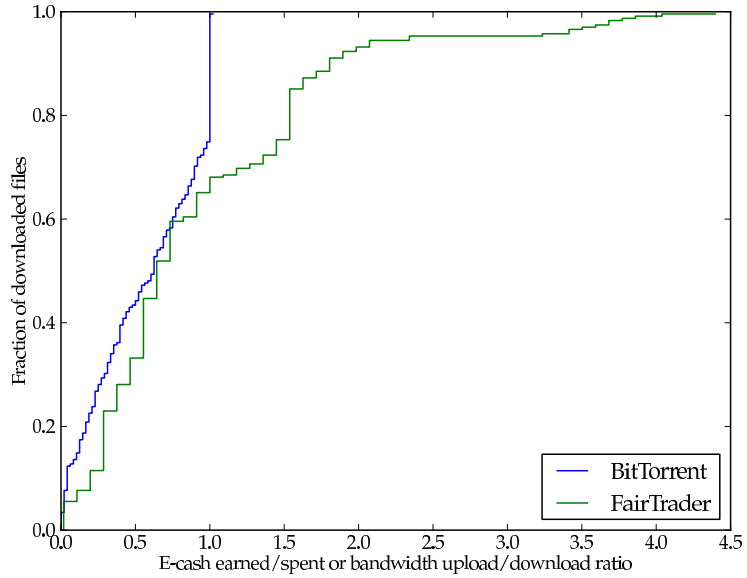


Figure 6.4: Cumulative distribution of ratio values for the multi-swarm scenario, with parameter settings $N = 98$, $|F| = 10$, $M = 3$, $s = 0.2$, and $T = 400$.

refers to bytes uploaded/downloaded for BitTorrent clients, and currency earned/spent for FairTrader). Once all peers have completed downloading, the experiment ends, leaving lower per-file ratios than 1.0 if there was not enough demand for the file. Figure 6.4 plots the distribution of these ratios for each file downloaded by each peer, after the experiment has completed.

Here we see that BitTorrent nodes stop participating in each file's swarm as soon as they have uploaded enough bandwidth to meet the ratio requirement, seen in the steep line at the 1.0 mark. Since BitTorrent provides no performance gain from continuing to upload these files, the rational choice for these peers is to commit upload bandwidth to other swarms that are still in progress. However, in FairTrader, since clients that continue to upload are rewarded with currency that may used to pay for blocks from other swarms in progress, we see that roughly 30% of files continue to be seeded beyond this point, with some files uploaded several times over. Clearly, e-cash provides FairTrader clients with extra seeding incentives not found in BitTorrent.

Fully-interested scenario While the previous scenarios have focused on the performance of BitTorrent and FairTrader under situations where blocks are relatively scarce (with low values of seeding fraction s), we have also examined scenarios where blocks are more readily available; that is, where every peer is interested in its neighbors' blocks, and vice-versa. The goal of these experiments is to provide a "level playing field" where all peers are equally able to conduct block transactions with each other, in order to examine the performance characteristics of each system in a highly competitive environment.

To induce this scenario we have created $|F| = 10$ files, each 20 megabytes in size, with $N = 19$ peers overall. At the start of the experiment, each peer begins with a randomly-selected subset of 50% of the total blocks in each file. Once the experiment starts, each peer immediately joins all 10 swarms and begins downloading every file simultaneously. The experiment ends when all peers have finished downloading the remaining 100 megabytes needed to complete their files. Additionally, just as in the previous scenario, a BitTorrent or FairTrader node may not leave an individual swarm until it has uploaded enough bandwidth to meet the 1.0 ratio requirement.

Figures 6.5 and 6.6 highlight the effects of fungibility and increased fairness provided by FairTrader over BitTorrent. Under BitTorrent, nodes cannot pool credit for uploads across different swarms, and thus are more likely to leave a swarm after completing it, resulting in a much higher variation of UL/DL ratios as seen in Figure 6.6. This can also be evidenced in the distribution of download times experienced by BitTorrent clients in Figure 6.5, where a fraction of files downloaded by peers take significantly longer to complete than with FairTrader. These unlucky nodes encounter slower download performance because fewer and fewer of their peers continue to upload in the swarms for files they are trying to complete as time progresses.

In contrast, the FairTrader system exhibits fairer performance characteristics: since fungible currency incentivizes clients to persist in all swarms even after completing a download, the performance across the entire system (that is, considering every file downloaded by every node) is much more even. This can be seen in the tight grouping of ratios around 1.0 in Figure 6.6 and the smoother distribution of download times in Figure 6.5.

We find that in this highly-competitive environment, where peers are engaged in multiple simultaneous downloads, BitTorrent's single-file tit-for-tat incentives do not guarantee good performance for all participants. In fact, as seen in Figure 6.6, BitTorrent arbitrarily assigns an unfair upload bandwidth burden on roughly 20% of nodes, while also arbitrarily

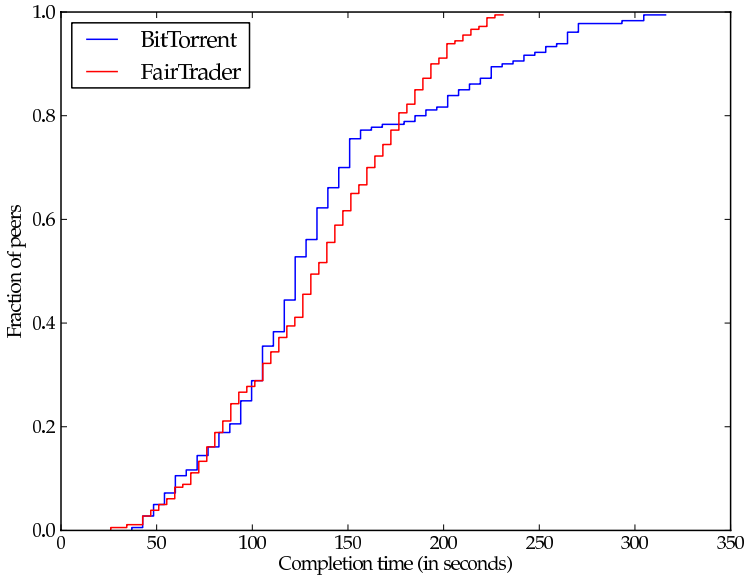


Figure 6.5: Cumulative distribution of download times for the fully-interested scenario using $N = 19$ nodes, each downloading all $|F| = 10$ files.

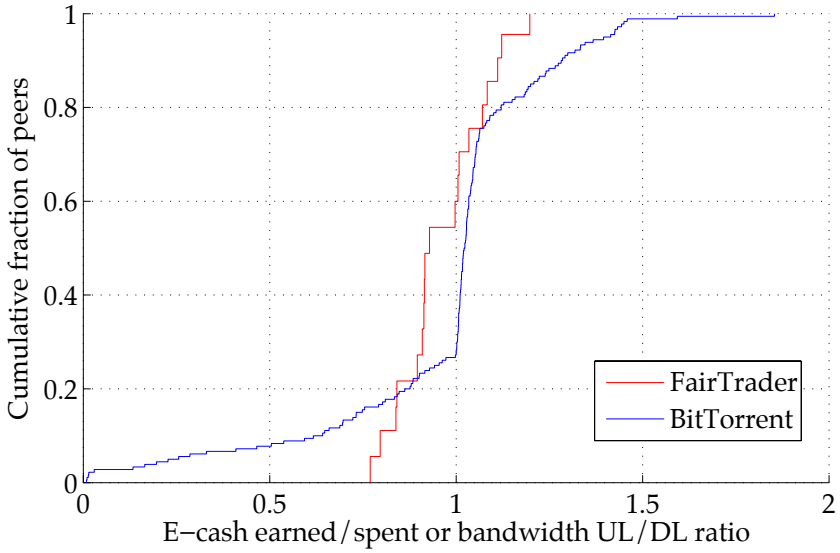


Figure 6.6: Cumulative distribution of fairness ratios for the fully-interested scenario using $N = 19$ nodes, each downloading all $|F| = 10$ files. FairTrader's incentives provide a much fairer distribution of UL/DL ratios around 1.

rewarding another 20% of downloaders, who need to upload only a fraction of the bandwidth they downloaded.

6.6 Summary

In general, previous work in providing better fairness guarantees in peer-to-peer systems has either been forced to sacrifice the privacy of the peers involved or has neglected to consider behavior across multiple swarms. To address these concerns, we have introduced FairTrader, a currency-based BitTorrent system that aims to provide long-term fairness guarantees to peers without having to track their every exchange. To achieve these goals, our design of FairTrader has incorporated well-studied cryptographic primitives while managing to keep the basic block-exchange mechanisms of BitTorrent largely the same.

In addition to our design, we have also implemented FairTrader and shown, in Section 6.5, the performance of our system in a variety of circumstances. We find that our client by nature motivates stronger fairness among peers, and that in environments where peers are engaged in many swarms, currency-based accounting can provide nodes with seeding incentives not found today in BitTorrent that improves overall download performance. These incentives provide a secure, tamper-proof, and privacy-preserving alternative to similar mechanisms already employed today by BitTorrent ratio tracker communities.

Chapter 7

Implementing e-cash with ZKPDL

Modern cryptographic protocols are complicated, computationally intensive, and, given their security requirements, require great care to implement. However, one cannot expect all good cryptographers to be good programmers, or vice versa. As a result, many newly proposed protocols—often described as efficient enough for deployment by their authors—are left unimplemented, despite the potentially useful primitives they offer to system designers. We believe that a lack of high-level software support (such as that provided by OpenSSL, which provides basic encryption and hashing) presents a barrier to the implementation and deployment of advanced cryptographic protocols, and in this work attempt to remove this obstacle.

One particular area of recent cryptographic research which has applications for privacy-preserving systems is *zero-knowledge proofs* [85, 83, 27, 73], which provide a way of proving that a statement is true without revealing anything beyond the validity of the statement. Among the applications of zero-knowledge proofs are electronic voting [91, 118, 63, 97], anonymous authentication [35, 61, 137], anonymous electronic ticketing for public transportation [92], verifiable outsourced computation [17, 80], and essentially any system in which honesty needs to be enforced without sacrificing privacy. Much recent attention has been paid to protocols based on anonymous credentials [47, 60, 38, 41, 19, 16], which allow users to anonymously prove possession of a valid credential (e.g., a driver's license), or prove relationships based on data associated with that credential (e.g., that a user's age lies within a certain range) without revealing their identity or other data. These protocols also prevent the person verifying a credential and the credential's issuer from colluding to link activity to specific users. As corporations and governments move to put an increasing amount of personal information online, the need for efficient privacy-preserving systems has become increasingly important and a major focus of recent research.

Another application of zero-knowledge proofs is electronic cash. The primary aim of ZKPDL has been to enable the efficient deployment of secure, anonymous electronic cash (e-cash) in network applications. Like physical coins, e-coins cannot be forged; furthermore, given two e-coins it is impossible to tell who spent them, or even if they came from the same user. For this reason, e-cash holds promise for use in anonymous settings and privacy-preserving applications, where free-riding by users may threaten a system's stability.

These e-cash protocols can also be used for payments in other systems that face free-riding problems, such as anonymous onion routing [42]. In such a system, routers would be paid for forwarding messages using e-cash, thus providing incentives to route traffic on behalf of others in a manner similar to that proposed by Androulaki et al. [6]. Since P2P systems like these require each user to perform many cryptographic exchanges, the need to

provide high performance for repeated executions of these protocols is paramount.

7.1 Goals

In this chapter, we hope to bridge the gap between design and deployment by providing a language, ZKPDL (Zero-Knowledge Proof Description Language), that enables programmers and cryptographers to more easily implement privacy-preserving protocols. We also provide a library, Cashlib, that builds upon our language to provide simple access to cryptographic protocols such as electronic cash, blind signatures, verifiable encryption, and fair exchange.

The design and implementation of our language and library were motivated by collaborations with systems researchers interested in employing e-cash in high-throughput applications, such as the P2P systems described earlier. The resulting performance concerns, and the complexity of the protocols required, 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. These twin concerns led to our language-based approach and work on the interpreter.

The high-level nature of our language brings two benefits. First, it frees the programmer from having to worry about the implementation of cryptographic primitives, efficient mathematical operations, generating and processing messages, etc.; instead, ZKPDL allows the specification of a protocol in a manner similar to that of theoretical descriptions. Second, it allows our library to make performance optimizations based on analysis of the protocol description itself.

ZKPDL permits the specification of many widely-used zero-knowledge proofs. We also provide an interpreter that generates and verifies proofs for protocols described by our language. The interpreter performs optimizations such as precomputation of expected exponentiations, translations to prevent redundant proofs, and caching compiled versions of programs to be loaded when they are used again on different inputs. More details on these optimizations are provided in Section 7.5.

Our e-cash library, Cashlib, described in Section 7.8, sits atop our language to provide simple access to higher-level cryptographic primitives such as e-cash [42], blind signatures [40], verifiable encryption [43], and optimistic fair exchange [18, 107]. Because of the modular nature of our language, we believe that the set of primitives provided by our library can be easily extended to include other zero-knowledge protocols.

Finally, we hope that our efforts will encourage programmers to use (and extend) our library to implement their cryptographic protocols, and that our language will make their job easier; we welcome contribution by our fellow researchers in this effort. Documentation and source code for our library can be found online at `http://github.com/brownie/cashlib`.

7.2 Cryptographic Background

There are two main modern cryptographic primitives used in our framework: *commitment schemes* and *zero-knowledge proofs*. Briefly, a commitment scheme can be thought of as cryptographically analogous to an envelope. When a user Alice wants to commit to a value, she puts the value in the envelope and seals it. Upon receiving a commitment, a second user Bob cannot tell which value is in the envelope; this property is called *hiding* (in this analogy, let's assume Alice is the only one who can open the envelope). Furthermore, because the

envelope is sealed, Alice cannot sneak another value into the envelope without Bob knowing: this property is called *binding*. To eventually reveal the value inside the envelope, all Alice has to do is open it (cryptographically, she does this by revealing the private value and any randomness used to form the commitment; this collection of values is aptly referred to as the *opening* of the commitment). We employ both Pedersen commitments [145] and Fujisaki-Okamoto commitments [78, 62], which rely on the security of the Discrete Log assumption and the Strong RSA assumption respectively.

Zero-knowledge proofs [85, 83] provide a way of proving that a statement is true to someone without that person learning anything beyond the validity of the statement. For example, if the statement were "I have access to this sytem" then the verifier would learn only that I really do have access, and not, for example, how I gain access or what my access code is. In our library, we make use of sigma proofs [59], which are three-message proofs that achieve a weaker variant of zero-knowledge known as *honest-verifier zero-knowledge*. We do not implement sigma protocols directly; instead, we use the Fiat-Shamir heuristic [75] that transforms sigma protocols into non-interactive (fully) zero-knowledge proofs, secure in the random oracle model [21].

A primitive similar to zero-knowledge is the idea of a *proof of knowledge* [20], in which the prover not only proves that a statement is true, but also proves that it knows a reason why the statement is true. Extending the above example, this would be equivalent to proving the statement "I have access to the system, and I know a password that makes this true."

In addition to these cryptographic primitives, our library also makes uses of hash functions (both universal one-way hashes [136] and Merkle hashes [127]), digital signatures [86], pseudo-random functions [82], and symmetric encryption [57]. The security of the protocols in our library relies on the security of each of these individual components, as well as the security of any commitment schemes or zero-knowledge proofs used.

7.3 Design

The design of our library and language arose from our initial goal of providing a high-performance implementation of protocols for e-cash and fair exchange for use in applications such as those described in the introduction. For these applications, the need to support many repeated interactions of the same protocol efficiently is a paramount concern for both the bank and the users. In the bank's case, it must conduct withdraw and deposit protocols with every user in the system, while in the user's case it is possible that a user would want to conduct many transactions using the same system parameters.

Motivated by these performance requirements, we initially developed a more straightforward implementation of our protocols using C++ and GMP [81], but found that our ability to modify and optimize our implementation was hampered by the complexity of our protocols. High-level changes to protocols required significant effort to re-implement; meanwhile, potentially useful performance optimizations became difficult to implement, and there was no way to easily extend the functionality of the library.

These difficulties led to our current design, illustrated in Figure 7.1. Our system allows a pseudocode-like description of a protocol to be developed using our description language, ZKPDL. This program is compiled by our interpreter, and optionally provided a list of public parameters, which are "compiled in" to the program. This produces an interpreter object, used by each party to prove and verify that the prover's private values satisfy a

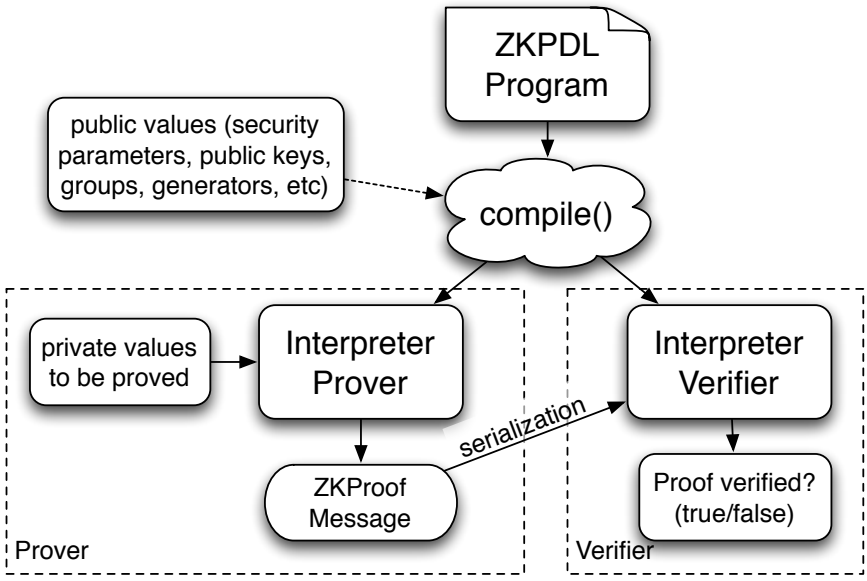


Figure 7.1: Usage of a ZKPDL program: a program is compiled separately by the prover and verifier, who also provide with a set of fixed public parameters.

certain set of relationships. Serialization and processing of proof messages are provided by the library.

At compile time, a number of transformations and optimizations are performed on the abstract syntax tree produced by our parser, which we developed using the ANTLR parser generator [143]. Once compiled, these interpreter objects can be used repeatedly by the prover to generate zero-knowledge proofs about private values, or by the verifier to verify these proofs.

Key to our approach is the simplicity of our language. It is not Turing-complete and does not allow for branching or conditionals; it simply describes the variables, equations, and relationships required by a protocol, leaving the implementation details up to the interpreter and language framework. This framework, described in the following section, provides C++ classes that parse, analyze, optimize, and interpret ZKPDL programs, employing many common compiler techniques (e.g., constant substitution and propagation, type-checking, providing error messages when undefined variables are used, etc.) in the process. We are able to understand and transform mathematical expressions into forms that provide better performance (e.g., through techniques for fixed-base exponentiation), and recognize relationships between values to be proved in zero-knowledge. All of these low-level optimizations, as well as our high-level primitives, should enable a programmer to quickly implement and evaluate the efficiency of a protocol.

We also provide a number of C++ classes that wrap ZKPDL programs into interfaces for generating and verifying proofs, as well as marshaling them between computers. We build upon these wrappers to additionally provide Cashlib, a collection of interfaces that allows a programmer to assume the role of buyer, seller, bank, or arbiter in a fair exchange system based on endorsed e-cash [42], as described in Chapter 3.

sample.zkp

```
computation: // compute values required for proof
  given: // declarations
    group: G = <g,h>
    exponents in G: x[2:3]
  compute: // declarations and assignments
    random exponents in G: r[1:3]
    x_1 := x_2 * x_3
    for(i, 1:3, c_i := g^x_i * h^r_i)

proof:
  given: // declarations of public values
    group: G = <g,h>
    elements in G: c[1:3]
    for(i, 1:3, commitment to x_i: c_i = g^x_i * h^r_i)
  prove knowledge of: // declarations of private values
    exponents in G: x[1:3], r[1:3]
  such that: // protocol specification; i.e. relations
    x_1 = x_2 * x_3
```

Figure 7.2: A sample program proving a product of two values.

7.4 The zero-knowledge proof description language (ZKPDL)

To enable implementation of the cryptographic primitives discussed in Section 7.2, we have designed a programming language for specifying zero-knowledge protocols, as well as an interpreter for this language. The interpreter is implemented in C++ and consists of approximately 6000 lines of code. On the prover side, the interpreter will output a zero-knowledge proof for the relations specified in the program; on the verifier side, the interpreter will be given a proof and verify whether or not it is correct. Therefore, the output of the interpreter depends on the role of the user, although the program provided to the interpreter is the same for both.

7.4.1 Language overview

Here we provide a brief overview of some fundamental language features to give an idea of how programs are written; a full grammar for our language, containing all of its features, can be found in our documentation available online, and further sample programs can be found in Section 7.6. A program can be broken down into two blocks: a computation block and a proof block. Each of these blocks is optional: if a user just wants a calculator for modular (or even just integer) arithmetic then he will specify just the computation block; if, on the other hand, he has all the input values pre-computed and justs wants a zero-knowledge proof of relations between these values, he will specify just the proof block. Figure 7.2 presents a sample program written in our language (indentations are included for readability, and are not required syntax).

In this example, we are proving that the value x_1 contained within the commitment c_1 is the product of the two values x_2 and x_3 contained in the commitments c_2 and c_3. The program can be broken down in terms of how variables are declared and used, and the computation and proof specifications. Note that some lines are repeated across the computation and proof blocks, as both are optional and hence considered independently.

7.4.2 Variable declaration

Two types of variables can be declared: group objects and numerical objects. Names of groups must start with a letter and cannot have any subscripts; sample group declarations can be seen in lines 3 and 12 of the above program. In these lines, we also declare the group generators, although this declaration is optional (as we will see later on in Section 7.6, it is also optional to name the group modulus).

Numerical objects can be declared in two ways. The first is in a list of variables, where their type is specified by the user. Valid types are `element`, `exponent` (which refer respectively to elements within a finite-order group and the corresponding exponents for that group), and `integer`; it should be noted that for the first two of these types a corresponding group must also be specified in the type information (see lines 4 and 13 for an example). The other way in which variables can be declared is in the compute block, where they are declared as they are being assigned (meaning they appear on the left-hand side of an equation), which we can see in lines 7 and 8. In this case, the type is inferred by the values on the right-hand side of the equation; a compile-time exception will be thrown if the types do not match up (for example, if elements from two different groups are being multiplied). Numerical variables must start with a letter and are allowed to have subscripts.

7.4.3 Computation

The computation block breaks down into two blocks of its own: the `given` block and the `compute` block. The `given` block specifies the parameters, as well as any values that have already been computed by the user and are necessary for the computation (in the example, the group `G` can be considered a system parameter and the values `x_2` and `x_3` are just needed for the computation).

The `compute` block carries out the specified computations. There are two types of computations: picking a random value, and defining a value by setting it equal to the right-hand side of an equation. We can see an example of the former in line 6 of our sample program; in this case, we are picking three random exponents in a group (note `r[1:3]` is just syntactic sugar for writing `r_1, r_2, r_3`). We also support picking a random integer from a specified range, and picking a random prime of a specified length (examples of these can be found in Section 7.6). As already noted, lines 7 and 8 provide examples of lines for computing equations. In line 8, the `for` syntax is again just syntactic sugar; this time to succintly specify the relations `c_1 = g^x_1*h^r_1`, `c_2 = g^x_2*h^r_2`, and `c_3 = g^x_3*h^r_3`. We have a similar `for` syntax for specifying products or sums (much like $\prod$ or $\sum$ in conventional mathematical notation), but neither of these `for` macros should be confused with a for loop in a conventional programming language.

7.4.4 Proof specification

The proof block is comprised of three blocks: the `given` block, the `prove knowledge of` block, and the `such that` block. In the `given` block, the parameters for the proof are specified, as well as the public inputs known to both the prover and verifier for the zero-knowledge protocol. In the `prove knowledge of` block, the prover's private inputs are specified. Finally, the `such that` block specifies the desired relations between all the values; the zero-knowledge proof will be a proof that these relations are satisfied. We currently support four main types of relations:

sample_prover.cc

```
group_map g;
variable_map v;
g["G"] = G;
v["x_2"] = x2;
v["x_3"] = x3;
InterpreterProver prover;
prover.check("sample.zkp", g); // compile program with groups
prover.compute(v); // compute intermediate values needed for proof
// compute and produces serializable proof message object
ProofMessage proofMsg(prover.getPublicVariables(),
                      prover.computeProof());
```

Figure 7.3: A sample C++ wrapper for the prover.

- Proving knowledge of the opening of a commitment [159]. We can prove openings of Pedersen [145] or Fujisaki-Okamoto commitments [78, 62]. In both cases we allow for commitments to multiple values.
- Proving equality of the openings of different commitments. Given any number of commitments, we can prove the equality of any subset of the values contained within the commitments.
- Proving that a committed value is the product of two other committed values [62, 32]. As seen 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, 117]. We can prove that the value x contained within a given commitment satisfies $lo \leq x < hi$, where lo and hi are both public values.

There are a number of other zero-knowledge proof types (e.g., proving a value is a Blum integer, proving that committed values satisfy some polynomial relationship, etc.), but we chose these four based on their wide usage in applications, in particular in e-cash and anonymous credentials. We note, however, that adding other proof types to the language should require little work (as mentioned in Section 7.5), as we specifically designed the language and interpreter with modularity in mind.

7.4.5 Sample usage

In addition to showing a sample program, we would also like to demonstrate a sample usage of our interpreter API. In order to use the sample ZKPDL program from Section 7.4.1, one could use the C++ code in Figure 7.3 (assuming there are already numerical variables named `x2` and `x3`, and a group named `G`):

The method is the same for all programs: any necessary groups and/or variables are inserted into the appropriate maps, which are then passed to the interpreter. Note that the group map in this case is passed to the interpreter at "compile time" so that it may precompute powers of group generators to be used for exponentiation optimizations (described in the next section); however, both the group and variable maps may be provided at "compute time." Any syntactic errors will be caught at compile time, but if the inputs provided

sample_verifier.cc

```
group_map g;
variable_map v;
g["G"] = G;
InterpreterVerifier verifier;
verifier.check("sample.zkp", g); // compile program
verifier.compute(v, proofMsg.publics); // compute intermediate values
bool verified = verifier.verify(proofMsg.proof); // verify proof
```

Figure 7.4: A sample C++ wrapper for the verifier.

at compute time are not valid for the relations being proved, the proof will be computed anyway and the error will be caught by the verifier. The `ProofMessage` is a serializable container for the zero-knowledge proof and any intermediate values (e.g., commitments and group bases) that the verifier might need to verify the proof.

The method is almost identical for the verifier, as seen in Figure 7.4. As we can see, the main difference is that the verifier uses both its own public inputs and the prover's public values at compute time (with its own inputs always taking precedence over the `ProofMessage` inputs), but still takes in the proof to be checked afterwards so that the actions of the prover and verifier remain symmetric.

7.5 ZKPDL Interpreter Optimizations

In our interpreter, we have incorporated a number of optimizations that make using our language not only more convenient but also more efficient. Here we describe the most significant optimizations, which include removing any redundancy when multiple proofs are combined and performing multi-exponentiations on cached bases when the same bases are used frequently. Other improvements specific to existing protocols can be found in Section 7.6.

7.5.1 Translation

To eliminate redundancy between different proofs, we first translate each proof described in Section 7.4.4 into a "fundamental discrete logarithm form." In this form, each proof can be represented by a collection of equations of the form $A = B^x \cdot C^y$. For example, if the prover would like to prove that the value x contained within $C_x = g^x h^{r_x}$ is equal to the product of the values y and z contained within $C_y = g^y h^{r_y}$ and $C_z = g^z h^{r_z}$ respectively, this is equivalent to a proof of knowledge of the discrete logarithm equalities $C_y = g^y h^{r_y}$ and $C_x = C_y^z h^{r_x - z r_y}$.

Our sample program in the previous section is first translated into this discrete logarithm form. During runtime, the values provided to the prover are then used to generate the zero-knowledge proof. In addition to eliminating redundancy between proofs of different relations in the program, this technique also allows our language to easily add new types of proofs as they become available. To add any proof that can be broken down into this discrete logarithm form, we need to add only a translation function and a rule in the grammar for how we would like to specify this proof in a program, and the rest of the work will be handled by our existing framework.

7.5.2 Multi-exponentiation

The computational performance of many cryptographic protocols, especially those used by our library, is often dominated by the need to perform many modular exponentiation operations on large integers. These operations typically involve the use of systems parameters as bases, with exponents chosen at random or provided as private inputs (e.g., Pedersen commitments, which require computation of $g^x \cdot h^r$, where g and h are publicly known). Algorithms for simultaneous multiple exponentiation allow the result of multi-base exponentiations such as these to be computed without performing each intermediate exponentiation individually; an overview of these protocols can be found in Section 14.6 of Menezes et al. [126].

Our interpreter leverages the descriptions of mathematical expressions in ZKPDL programs to recognize when fixed-base exponentiation operations occur, allowing it to precompute lookup tables at compile time that can speed up these computations dramatically. In addition to single-table multi-exponentiation techniques (i.e., the 2^w-ary method [126]), we offer programmers who expect to run the same protocol many times the ability to take advantage of time/space tradeoffs by generating large lookup tables of precomputed powers. This allows a programmer to choose parameters that balance the memory requirements of the interpreter against the need for fast exponentiation.

For single-base exponentiation, we employ window-based precomputation techniques similar to those used by PBC [120] to cache powers of fixed bases. For multi-base exponentiation of k exponents, we currently extend the 2^w-ary method to store 2^{kw}-sized lookup tables for each w-bit window of the expected exponent length, so that multi-exponentiations on exponents of length n require only n/w multiplications of stored values. While we are also evaluating other algorithms offering similar time-space tradeoffs, we demonstrate the performance gains afforded by these techniques later in Table 7.1.

7.5.3 Interpreter caching

We also cache the parsed, compiled environments of ZKPDL programs when they are first run. Because we accept system parameters at compile time, we are able to evaluate and propagate any subexpressions made up of fixed constants and perform exponentiation precomputations before these expressions are fully evaluated at runtime. Even without the use of large tables for fixed-based exponentiation, this optimization proves useful when repeated executions of the same program must be performed; e.g., for a bank dealing with e-coin deposits. In this case, a bank must invoke the interpreter for each coin deposited; looking ahead to Table 7.1 we see that, on average, this operation takes the bank 83ms. If our program were re-parsed each time, it would take an extra 10ms, as opposed to the fraction of a millisecond required to load a cached interpreter environment, saving the bank approximately 10% of computation time per transaction by avoiding parsing overhead.

7.6 Sample programs

Using our language, we have written programs for a wide variety of cryptographic primitives, including blind signatures [40], verifiable encryption [43], and endorsed e-cash [42]. In the following sections, we provide our programs for these three primitives; in addition, performance benchmarks for all of them can be found at the end of the section.

cl-recipient-proof.zkp

```
computation:
  given:
    group: pkGroup = <fprime, gprime[1:L+k], hprime>
    exponents in pkGroup: x[1:L]
    integers: stat, modSize
  compute:
    random integer in [0,2^(modSize+stat)): vprime
    C := hprime^vprime * for(i, 1:L, *, gprime_i^x_i)

proof:
  given:
    group: pkGroup = <fprime, gprime[1:L+k], hprime>
    group: comGroup = <f, g, h, h1, h2>
    element in pkGroup: C
    elements in comGroup: c[1:L]
    for(i, 1:L, commitment to x_i: c_i=g^x_i*h^r_i)
    integer: l_x
  prove knowledge of:
    integers: x[1:L]
    exponents in comGroup: r[1:L]
    exponent in pkGroup: vprime
  such that:
    for(i, 1:l, range: (-(2^l_x-1)) <= x_i < 2^l_x)
    C = hprime^vprime * for(i, 1:L, gprime_i^x_i)
    for(i, 1:L, c_i = g^x_i * h^r_i)
```

Figure 7.5: CL signatures in ZKPDL, phase one: partial signature.

cl-issuer-proof.zkp

```
computation:
  given:
    group: pkGroup = <f, g[1:L+k], h>
    element in pkGroup: C
    exponents in pkGroup: x[1:k+L]
    integers: stat, modSize, lx
  compute:
    random integer in [0,2^(modSize+lx+stat)): vpp
    random prime of length lx+2: e
    einverse := 1/e
    A := (f*C*h^vpp * for(i,L+1:k+L,*,g_i^x_i))^einverse

proof:
  given:
    group: pkGroup = <f, g[1:L+k], h>
    elements in pkGroup: A, C
    exponents in pkGroup: e, vpp, x[L+1:k]
  prove knowledge of:
    exponents in pkGroup: einverse
  such that:
    A = (f*C*h^vpp * for(i,L+1:k+L,*,g_i^x_i))^einverse
```

Figure 7.6: CL signatures in ZKPDL, phase two: issuer proof.

cl-possession-proof.zkp

```
computation:
  given:
    group: pkGroup = <fprime, gprime[1:L+k], hprime>
    element in pkGroup: A
    exponents in pkGroup: e, v, x[1:L]
    integers: modSize, stat
  compute:
    random integers in [0,2^(modSize+stat)): r, r_C
    vprime := v + r*e
    Aprime := A * hprime^r
    C := h^r_C * for(i, 1:L, *, gprime_i^x_i)
    D := for(i, L+1:L+k, *, gprime_i^x_i)
    fCD := f * C * D

proof:
  given:
    group: pkGroup = <fprime, gprime[1:L+k], hprime>
    group: comGroup = <f, g, h, h1, h2>
    elements in pkGroup: C, D, Aprime, fCD
    elements in comGroup: c[1:L]
    for(i, 1:L, commitment to x_i: c_i=g^x_i*h^r_i)
    exponents in pkGroup: x[L+1:L+k]
    integer: l_x
  prove knowledge of:
    integers: x[1:L]
    exponents in comGroup: r[1:L]
    exponents in pkGroup: e, vprime, r_C
  such that:
    for(i, 1:L, range: (-(2^l_x - 1)) <= x_i < 2^l_x)
    C = hprime^r_C * for(i, 1:l, *, gprime_i^x_i)
    for(i, 1:L, c_i = g^x_i * h^r_i)
    fCD = (Aprime^e) * hprime^(r_C - vprime)
```

Figure 7.7: CL signatures in ZKPDL, phase three: proof of signature possession.

7.6.1 CL signatures

Using our language, we have implemented the blind signature scheme due to Camenisch and Lysyanskaya [40]; as we will see in Section 7.6.3, CL signatures are integral to endorsed e-cash. Briefly, a *blind signature*, as introduced by Chaum [45], enables a signature issuer to sign a message without learning the contents of the message. A CL signature works in two main phases: an issuing phase, in which a user actually obtains the signature, and a proving phase, in which the user is able to prove (in zero-knowledge) to other users that he does in fact possess a valid CL signature.

The issuing phase is a one-round interaction between the recipient and the issuer, at the end of which the recipient obtains the blind signature on her message(s). Because the protocol is interactive, we present one program for each stage of the protocol. At the end of this first stage, implemented in Figure 7.5, the signature issuer will return a *partial signature* to the recipient, who will then use this signature to compute the full signature on the hidden message.

Next, the issuer must prove the partial signature is computed correctly, as in the program shown in Figure 7.6. Once the recipient obtains the partial signature, she can unblind it to obtain a full signature; this step completes the issuing phase.

Finally, in Figure 7.7, the owner of a CL signature needs a way to prove that she has a signature, without revealing either the signature or the values. To accomplish this, the prover first randomizes the CL signature and then attaches a zero-knowledge proof of knowledge that the randomized signature corresponds to the original signature on the committed message.

7.6.2 Verifiable encryption

Briefly, verifiable encryption consists of a ciphertext under the public key of some trusted third party (in our case, the arbiter) and a zero-knowledge proof that the values inside the ciphertext satisfy some relation; this pair is often referred to as a *verifiable escrow*. Our implementation of verifiable encryption, shown in Figure 7.8, is based on the construction of Camenisch and Shoup [43]. 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 then verify that the encrypted values correspond to the opening of the commitment.

7.6.3 E-cash

Electronic cash, or e-cash for short, was first introduced by Chaum [45] and can be thought of as the electronic equivalent of cash; i.e., an electronic currency that preserves users' anonymity, as opposed to electronic checks [48] or credit cards. We implement endorsed e-cash, due to Camenisch, Lysyanskaya, and Meyerovich [42] (which is an extension of compact e-cash [36]), for two main reasons. Our first reason is that an endorsed e-coin can be split up into two parts, its endorsement and an unendorsed component; only with both of these parts can the coin be considered complete. As we will see in Section 7.8.2, this property enables efficient fair exchange. The second reason for choosing endorsed e-cash is that it is *offline*, which means the bank does not need to be active in every transaction; this significantly reduces the burden placed on the bank. Although the bank does not check the coin in every interaction, endorsed e-cash has the property that double-spenders (i.e., users who try to spend the same coin twice) can be caught by the bank at the time of deposit

verifiable-encryption.zkp

```
computation:
  given:
    group: secondGroup = <g[1:m], h>
    group: RSAGroup
    modulus: N
    group: G
    group: cashGroup = <f_3, gprime, hprime, f_1, f_2>
    exponents in G: x[1:m]
    elements in G: u[1:m], v, w
  compute:
    random integer in [0,N/4): s
    random exponents in secondGroup: r[1:m]
    for(i, 1:m, c_i := g_1^x_i * g_2^r_i)
    Xprime := for(i, 1:m, *, g_i^x_i) * h^s
    vsquared := v^2
    wsquared := w^2
    for(i, 1:m, usquared_i := u_i^2)

proof:
  given:
    group: secondGroup = <g[1:m], h>
    group: G
    group: RSAGroup
    modulus: N
    group: cashGroup = <f_3, gprime, hprime, f_1, f_2>
    element in cashGroup: X
    elements in secondGroup: Xprime, c[1:m]
    for(i,1:m,commitment to x_i: c_i=g_1^x_i*g_2^r_i)
    elements in G: a[1:m], b, d, e, f, usquared[1:m],
                   vsquared, wsquared
  prove knowledge of:
    integers: x[1;M], r
    exponent in G: hash
    exponents in secondGroup: r[1:m], s
  such that:
    for(i, 1:m, range: -N/2 + 1 <= x_i < N/2)
    vsquared = f^(2*r)
    wsquared = (d * e^hash)^(2*r)
    for(i, 1:m, usquared_i = b^(2*x_i) * a_i^(2*r))
    X = for(i, 1:m, *, f_i^x_i)
    Xprime = for(i, 1:m, *, g_i^x_i) * h^s
```

Figure 7.8: ZKPDL Implementation of verifiable encryption [43].

Program type	Prover (ms)		Verifier (ms)		Proof size	Cache	Multi-exps	
	With cache	Without	With cache	Without	(bytes)	size (MB)	Prover	Verifier
DLR proof	3.07	3.08	1.26	1.25	511	0	2	1
Multiplication proof	2.03	4.07	1.66	2.32	848	33.5	8	2
Range proof	36.36	74.52	21.63	31.54	5455	33.5	31	11
CL recipient proof	119.92	248.31	70.76	112.13	19189	134.2	104	39
CL issuer proof	7.29	7.38	1.73	1.73	1097	0	2	1
CL possession proof	125.89	253.17	78.19	117.67	19979	134.2	109	40
Verifiable encryption	416.09	617.61	121.87	162.77	24501	190.2	113	42
Coin	134.37	271.34	83.01	121.83	22526	223.7	122	45

Table 7.1: Time (in milliseconds) and size (in bytes) required for various zero-knowledge proofs, averaged over twenty runs. Timings are considered from both the prover and verifier sides, as are the number of multi-exponentiations, and are considered both with and without caching for fixed-based exponentiations; the size of the cache is also measured (in megabytes). As we can see, using caching results on average in a 48% speed improvement for the prover, and a 31% improvement for the verifier.

and punished accordingly. Because e-cash is meant to preserve privacy, however, a user is also guaranteed that unless she double spends a coin, her identity will be kept secret.

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. This can be accomplished using the simple program in Figure 7.9.

Once the bank has verified this proof, the user and the bank will run a protocol to obtain a CL signature (using the programs we saw in Section 7.6.1) 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 one of her coins, she splits it up into its unendorsed part and the endorsement. She then sends the unendorsed component to a merchant and proves it is valid. If the merchant then sends her what she wanted to buy, she will follow up with the endorsement to complete the coin and the transaction is complete. The program shown in Figure 7.10 is used to prove the validity of a coin.

7.7 Performance of ZKPDL

Here we measure the communication and computational resources used by our system when running each of the programs above. The benchmarks presented in Table 7.1 were collected on a MacBook Pro with a 2.53GHz Intel Core 2 Duo processor and 4GB of RAM running OS X 10.6; we therefore expect that these results will reflect those of a typical home user with no special cryptographic hardware support.

As for speed, caching exponents of fixed bases results in a significant performance increase, making it an important optimization for applications that require repeated protocol executions. The only caveat is that the exponent cache required for complex protocols can grow to hundreds of megabytes (using faster-performing parameters), and so our library allows users to choose whether to use caching, and if so how much of the cache should be used by this optimization.

The time taken for the higher-level protocols provides a clear view of the complexity of

user-id-proof.zkp

```
proof:
  given:
    group: cashGroup = <f,g,h,h1,h2>
    elements in cashGroup: A, pk_u
      commitment to sk_u: A = g^sk_u * h^r_u
  prove knowledge of:
    exponents in cashGroup: sk_u, r_u
  such that:
    pk_u = g^sk_u
    A = g^sk_u * h^r_u
```

Figure 7.9: Proof of user identity to bank, in ZKPDL.

coin-proof.zkp

```
computation:
  given:
    group: cashGroup = <f, g, h, h1, h2>
    exponents in cashGroup: s, t, sk_u
    integer: J
  compute:
    random exponents in cashGroup: r_B, r_C, r_D, x1, x2, r_y, R
    alpha := 1 / (s + J)
    beta := 1 / (t + J)
    C := g^s * h^r_C
    D := g^t * h^r_D
    y := h1^x1 * h2^x2 * f^r_y
    B := g^sk_u * h^r_B
    S := g^alpha * g^x1
    T := g^sk_u * (g^R)^beta * g^x2

proof:
  given:
    group: cashGroup = <f, g, h, h1, h2>
    elements in cashGroup: y, S, T, B, C, D
    commitment to sk_u: B = g^sk_u * h^r_B
    commitment to s: C = g^s * h^r_C
    commitment to t: D = g^t * h^r_D
    integer: J
  prove knowledge of:
    exponents in cashGroup: x1, x2, r_y, sk_u, alpha, beta, s, t, r_B, r_C, r_D, R
  such that:
    y = h1^x1 * h2^x2 * f^r_y
    S = g^alpha * g^x1
    T = g^sk_u * (g^R)^beta * g^x2
    g = (g^J * C)^alpha * h^(-r_C / (s+J))
    g = (g^J * D)^beta * h^(-r_D / (t+J))
```

Figure 7.10: Coin validity proof in ZKPDL.

each protocol. 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 and five more discrete log relations.

Table 7.1 also shows that verifiable encryption is by far the biggest bottleneck, requiring almost three times as much time to compute as any other step. As seen in the program in Section 7.6.2, 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 [149]. Because the values used in our verifiable encryption program are much larger than the ones used in CL signatures (about 1024 vs. 160 bits, to get 80-bit security for both), this decomposition often takes considerably more time 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 wouldn't be of any use.

A final observation on computational performance is that proving possession of a CL signature completely dominates the time required to prove the validity of a coin, since the timings for the two proofs are within milliseconds. This suggests that the only way to significantly improve the performance of e-coins and verifiable encryption would be to develop more efficient techniques for range proofs (which has in fact been the subject of some recent cryptographic research [91, 33, 160]).

In terms of proof size, range proofs are much larger than proofs for discrete logarithms or multiplication. This is to be expected, as translating a range proof into discrete logarithm form (as described in Section 7.5) requires eleven equations, whereas a single DLR proof requires only one, and a multiplication proof requires two.

7.8 Implementation of Cashlib

Using the primitives described in the previous section, we wrote a cryptographic library designed for optimistic fair exchange protocols. Fair exchange [56] involves a situation in which a buyer wants to make sure that she doesn't pay a merchant unless she gets what she is buying, while the merchant doesn't want to give away his goods unless he is guaranteed to be paid. It is known that fair exchange cannot be done without a trusted third party [140], but *optimistic* fair exchange [7, 10] describes the cases in which the trusted third party has to get involved only in the case of a dispute.

The library was written in C++ and consists of approximately 11000 lines of code in addition to the interpreter. A previous version of the library in which all the protocols and proofs were hand-coded (i.e., the interpreter was not used) consisted of approximately 20000 lines of code, meaning that the use of roughly 400 lines of ZKPDL was able to replace 9000 lines of our original C++ code (and, as we will see, make our operations more efficient as well).

7.8.1 Endorsed e-cash

A description of endorsed e-cash can be found in Section 7.6.3; 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 verify only when

the same public key of the bank is used, and thus the merchant can check for himself the denomination of the coin.

The program in Section 7.6.3 also reflects our decision to 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 Table 7.1 for a reminder), this optimization resulted in coins that were 20% smaller and 21% 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.

7.8.2 Buying and Bartering

Our library implements two efficient optimistic fair exchange protocols for use with e-cash. The first is the *buy* protocol for exchanging a coin with a file (described in Chapter 3), and the improved *barter* protocol of Küpçü and Lysyanksaya [107] 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 (described in Chapters 3 and 6). 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 7.7, 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, our FairTrader implementation from Chapter 6 depends on both protocols. Peers who have nothing to offer but would still like to download can offer to buy the files, while peers who would like only 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 always be used 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. This modification, as well as outlines of both the protocols, can be seen in Figure 7.11. We further modified both protocols to let them exchange multiple blocks at once, so that one block of the fair exchange protocol might correspond to multiple blocks of the underlying file.

We give an overview of each protocol below, with the optimizations we have added. We have also implemented the trusted third parties (the bank and the arbiter) necessary for e-cash and fair exchange.

Buying

The modified buy protocol is depicted on the left in Figure 7.11, although we also allow for the users to participate in the original buy protocol (in which the messages appear in a slightly different order). To initiate the modified buy protocol, the buyer sends a "setup" message, which consists of an unendorsed coin and a verifiable escrow on its corresponding endorsement. Upon receiving this message, the seller will use the programs in Section 7.6 to check the validity of the coin and the escrow. If these proofs verify, the seller will proceed by sending back an encrypted version of his file (or file block). Upon receiving this ciphertext, the buyer will store it (and a Merkle hash of it, for use with the arbiter in case the protocol goes wrong later on) and send back a contract, which consists of a hash of the seller's file and some session information. The seller will check this contract and, if satisfied with the details of the agreement, send back its decryption key. The buyer can then use this key to decrypt the ciphertext it received in the second message of the protocol. If the decryption is successful, the buyer will send back his endorsement on the coin. If in these last steps either party is unsatisfied (for example, the file does not decrypt or the endorsement isn't valid for the coin from the setup message), they can proceed to contact the arbiter and run resolution protocols (Section 3.3.3).

Bartering

This protocol is depicted on the right in Figure 7.11; because the first two messages of the barter protocol (the setup message and the encrypted data) are identical to those in the buy protocol described in the previous section, we do not describe them again here and instead jump directly to the third message. Because bartering involves an exchange of data, the initiator will respond to the receipt of the ciphertext with a ciphertext of her own, corresponding to an encryption of her file. She will also send a contract, which is similar to the buy contract but also contains hash information for her file. The responder will then check this contract as the seller did in the buy protocol, and if satisfied with the agreement will send back his decryption key. If the ciphertext decrypts correctly (i.e., decrypts to the file described in the contract) then the initiator can respond in turn with her own decryption key. If this decryption key is also valid, both parties have successfully obtained the desired files and the barter protocol can be considered complete. If neither party had to contact the arbiter (for similar reasons as in the buy protocol; i.e., a file did not decrypt correctly) then they are free to engage in future barter protocols without the overhead of an additional setup message. Otherwise, they need to resolve with the arbiter [107].

7.9 Performance of Cashlib

In Table 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 7.7.

The numbers in Table 7.2 clearly demonstrate our earlier observation that bartering is considerably more efficient than buying, both in terms of computation and communication overhead. The setup message for both buying and bartering takes about 600ms 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).

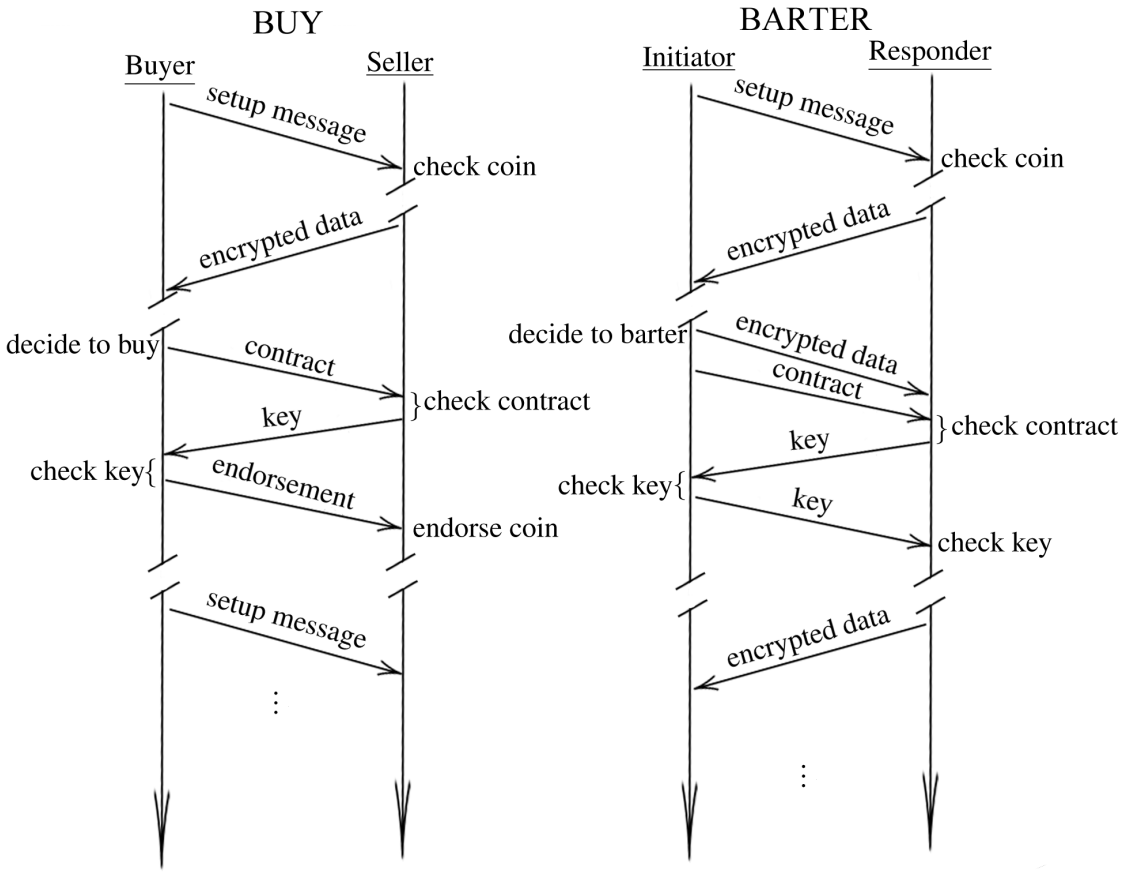


Figure 7.11: An outline of both the buy and barter protocols. Until the decision to buy or barter, the two protocols are identical; the main difference is that in a buy protocol, the setup message must be sent for each file exchange, which results in linear efficiency loss compared to bartering.

Operation	Time (ms)	"Naïve" time (ms)	Size (B)
Withdraw (user)	126.35	290.79	20093
Withdraw (bank)	83.36	140.02	1167
Deposit (bank)	82.11	128.36	22526
Buying a block (buyer)	628.49	901.04	47286
Buying a block (seller)	211.89	275.94	203
Barter setup message	608.29	881.32	46934
Checking setup message	210.61	276.98	n/a
Barter after setup (initiator)	18.02	18.28	1280
Barter after setup (responder)	1.11	1.18	204

Table 7.2: Average time required and network overhead, in milliseconds and bytes respectively, for each e-cash protocol implemented by Cashlib. The timings were averaged over twenty runs, and caching and compression optimizations were used. For the naïve timings, an older version of the library was used, which uses some multi-exponentation optimization techniques but not the interpreter; we can see a clear improvement when using ZKPDL. Parameters were used to provide a security level of 80 bits (160-bit SHA-1 hashing, 128-bit AES encryption, 1024-bit RSA moduli, and 1024-bit DSA signatures).

In addition, we consider the same protocols run using a previous "naïve" version of our library, which provided the same e-cash API and employed some multi-exponentation optimizations, but did not use ZKPDL. Using the optimizations available to the interpreter is considerably faster over our previous approach, meaning that our interpreter has not only made developing our protocols more convenient, but has also helped to improve efficiency.

Chapter 8

Conclusion and future work

Recall, from Chapter 1, the primary assertion of this thesis:

> E-cash accounting techniques are practical and well-suited for providing fairness, robustness, and better long-term incentives in decentralized systems.

Chapter 3 presents a design describing how e-cash accounting techniques could be applied to distributed system applications such as file-sharing, distributed storage, computation outsourcing, and onion routing. Key components in such a currency-based approach include the use of *contracts* which describe the resources being exchanged, and *fair exchange protocols* for buying and bartering those resources.

Chapter 4 provides greater detail on how users in a computation outsourcing system could be incentivized to behave correctly. It found that by farming the same jobs to multiple workers, and by penalizing users for misbehavior, the impact of selfish or malicious workers can be acceptably mitigated.

Chapter 5 provides an essential primitive for the use of currency in distributed storage networks: a system for creating *dynamic proofs of data possession* that guarantee to a buyer of storage services that her remote data remains intact and unmolested. This enables the buyer to erase her local copy of the data, retaining only a constant-sized piece of metadata, and allows her to to issue any number of challenges to the storage provider to ensure her data is intact. These proofs also allow her to insert, modify, or delete certain blocks of her remote data, and update her local metadata, without having to retrieve the entire dataset from the provider. Section 5.7 demonstrates that these proofs are computationally inexpensive to compute, are relatively small, and have direct applications to storage systems with extensive metadata, such as version-controlled file systems.

Chapter 6 elaborates on the design presented in Chapter 3 and presents the implementation of FairTrader, a peer-to-peer file-sharing system based on BitTorrent. It describes practical measures that such a system must take in order to mitigate the cryptographic overhead required by e-cash accounting. The seeding incentives provided by FairTrader's design have much in common with ratio-tracking techniques deployed in many BitTorrent communities today, described in Section 2.2.6.

Finally, Chapter 7 presents an implementation approach and software library that makes e-cash practical for use in network systems. Significant performance gains are reaped by decoupling the description of protocols involving zero-knowledge proofs from their implementation, through the use of a simple description language and focused optimization efforts in the language's runtime engine. These methods could be expanded to describe a wider range

of cryptographic protocols in the future, such as those which use pairing-based cryptography. Further optimization efforts still offer promise: for example, multi-core architectures could be exploited to lower e-cash transaction latency.

One area left largely unconsidered by this thesis is the application of e-cash to anonymous or privacy-preserving systems, such as Tor [66]. Like file-sharing systems, onion routing networks also suffer from fairness and free-riding problems, but it is difficult to develop a reputation or incentive system that rewards users with better service for forwarding other users' traffic without leaking information that threatens their privacy [64]. While users of e-cash are anonymous from the bank's viewpoint, and coins cannot identify a buyer to the seller, any deployment of currency-based accounting on an anonymous network would have to consider possible side-channel attacks related to communication with the bank and other trusted third parties, from the viewpoint of an adversary intent on linking traffic between users. A design that provided forwarding incentives by placing e-cash at the core of a system like Tor might consider ways to mask or batch traffic related to e-coin transactions to prevent making traffic analysis attacks more effective.

Still, systems such as BitTorrent, which do not provide strong privacy guarantees (as a tracker generally is employed that can record peer activity), might find enhanced privacy due to the use of e-cash accounting as a replacement for reputation systems or user-based ratio-tracking systems like those currently deployed today (Section 2.2.6). Future peer-to-peer system designers could benefit from e-cash incentives and remain confident that currency might be easily adapted for their application without threatening user privacy. It is this author's hope that in the future, as the computational power of the average user's computer increases, and as concerns for user privacy in peer-to-peer systems grow, that these e-cash accounting protocols and techniques might be of use to future system designers in need of a fair, fungible, and secure incentive system.

Bibliography

[1] A. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J. Martin, and C. Porth. BAR fault tolerance for cooperative services. *ACM SIGOPS Operating Systems Review*, 39(5):45–58, 2005.

[2] J. B. Almeida, E. Bangerter, M. Barbosa, S. Krenn, A.-R. Sadeghi, and T. Schneider. A certifying compiler for zero-knowledge proofs of knowledge based on sigma-protocols. In *ESORICS 2010*, 2010.

[3] K. G. Anagnostakis and M. B. Greenwald. Exchange-based incentive mechanisms for peer-to-peer file sharing. In *ICDCS '04: Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, 2004.

[4] A. Anagnostopoulos, M. Goodrich, and R. Tamassia. Persistent Authenticated Dictionaries and Their Applications. *ISC*, pages 379–393, 2001.

[5] N. Andrade, M. Mowbray, A. Lima, G. Wagner, and M. Ripeanu. Influences on cooperation in BitTorrent communities. In *P2PECON '05: Proceeding of the 2005 ACM SIGCOMM workshop on Economics of peer-to-peer systems*, 2005.

[6] E. Androulaki, M. Raykova, S. Srivatsan, A. Stavrou, and S. Bellovin. PAR: payment for anonymous routing. In *Privacy Enhancing Technologies Symposium (PETS)*, volume 5134 of *Lecture Notes in Computer Science*, pages 219–236. Springer-Verlag, 2008.

[7] N. Asokan, V. Shoup, and M. Waidner. Optimistic fair exchange of digital signatures. *IEEE Journal on Selected Areas in Communications*, 18(4):591–610, 2000.

[8] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song. Provable data possession at untrusted stores. In *ACM Conference on Computer and Communications Security*, pages 598–609, 2007.

[9] G. Ateniese, R. D. Pietro, L. V. Mancini, and G. Tsudik. Scalable and efficient provable data possession. SecureComm, 2008.

[10] G. Avoine and S. Vaudenay. Optimistic fair exchange based on publicly verifiable secret sharing. In *ACISP*, volume 3108 of *Lecture Notes in Computer Science*, pages 74–85. Springer-Verlag, 2004.

[11] E. Bangerter, S. Barzan, S. Krenn, A.-R. Sadeghi, T. Schneider, and J.-K. Tsay. Bringing zero-knowledge proofs of knowledge to practice. In *17th International Workshop on Security Protocols*, 2009.

[12] E. Bangerter, J. Camenisch, S. Krenn, A.-R. Sadeghi, and T. Schneider. Automatic generation of sound zero-knowledge protocols. Cryptology ePrint Archive, Report 2008/471, 2008. `http://eprint.iacr.org/2008/471`.

[13] F. Bao, R. Deng, and W. Mao. Efficient and practical fair exchange protocols with off-line ttp. In *IEEE Symposium on Security and Privacy*, pages 77–85. IEEE Computer Society Press, 1998.

[14] M. Barbosa, R. Noad, D. Page, and N. Smart. First steps toward a cryptography-aware language and compiler. Cryptology ePrint Archive, Report 2005/160, 2005. `http://eprint.iacr.org/2005/160`.

[15] S. A. Baset and H. Schulzrinne. An analysis of the Skype peer-to-peer Internet telephony protocol. In *Proc. of the INFOCOM '06*, Barcelona, Spain, April 2006.

[16] M. Belenkiy, J. Camenisch, M. Chase, M. Kohlweiss, A. Lysyanskaya, and H. Shacham. Delegatable anonymous credentials. In *Proceedings of Crypto 2009*, volume 5677 of *Lecture Notes in Computer Science*, pages 108–125. Springer-Verlag, 2009.

[17] M. Belenkiy, M. Chase, C. Erway, J. Jannotti, A. Küpçü, and A. Lysyanskaya. Incentivizing outsourced computation. In *NetEcon*, pages 85–90, 2008.

[18] M. Belenkiy, M. Chase, C. Erway, J. Jannotti, A. Küpçü, A. Lysyanskaya, and E. Rachlin. Making P2P accountable without losing privacy. In *WPES*, pages 31–40. ACM, 2007.

[19] M. Belenkiy, M. Chase, M. Kohlweiss, and A. Lysyanskaya. Non-interactive anonymous credentials. In *Proceedings of the 5th Theory of Cryptography Conference (TCC)*, pages 356–374, 2008.

[20] M. Bellare and O. Goldreich. On defining proofs of knowledge. In *Proceedings of Crypto 1992*, volume 740 of *Lecture Notes in Computer Science*, pages 390–420. Springer-Verlag, 1992.

[21] M. Bellare and P. Rogaway. Random oracles are practical: a paradigm for designing efficient protocols. In *ACM Conference on Computer and Communications Security (CCS) 1993*, pages 62–73, 1993.

[22] A. Ben-David, N. Nisan, and B. Pinkas. FairplayMP: a system for secure multi-party computation. In *ACM Conference on Computer and Communications Security (CCS) 2008*, pages 257–266, 2008.

[23] A. R. Bharambe, C. Herley, and V. N. Padmanabhan. Analyzing and improving a BitTorrent network's performance mechanisms. In *Proc. IEEE INFOCOM*, Barcelona, Spain, Mar. 2006.

[24] P. Bichsel, C. Binding, J. Camenisch, T. Groß, T. Heydt-Benjamin, D. Sommer, and G. Zaverucha. Cryptographic protocols of the identity mixer library, v. 1.0. IBM Research Report RZ3730, 2009.

[25] A. Blanc, Y.-K. Liu, and A. Vahdat. Designing incentives for peer-to-peer routing. In *Proc. IEEE INFOCOM*, Miami, FL, Mar. 2005.

[26] B. Blanchet and D. Pointcheval. Automated security proofs with sequences of games. In *Proceedings of Crypto 2006*, volume 4117 of *Lecture Notes in Computer Science*, pages 537–554. Springer-Verlag, 2006.

[27] M. Blum, A. de Santis, S. Micali, and G. Persiano. Non-interactive zero-knowledge. *SIAM Journal of Computing*, 20(6):1084–1118, 1991.

[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, B. Lynn, and H. Shacham. Short signatures from the weil pairing. In *ASIACRYPT*, 2001.

[31] Boost. The Boost C++ libraries. `http://www.boost.org`.

[32] F. Boudot. Efficient proofs that a committed number lies in an interval. In *Proceedings of Eurocrypt 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 431–444. Springer-Verlag, 2000.

[33] J. Camenisch, R. Chaabouni, and abhi shelat. Efficient protocols for set membership and range proofs. In *Proceedings of Asiacrypt 2008*, pages 234–252, 2008.

[34] J. Camenisch and E. V. Herreweghen. Design and implementation of the Idemix anonymous credential system. In *ACM Conference on Computer and Communications Security (CCS) 2002*, pages 21–30. ACM, 2002.

[35] J. Camenisch, S. Hohenberger, M. Kohlweiss, A. Lysyanskaya, and M. Meyerovich. How to win the clone wars: efficient periodic n-times anonymous authentication. In *ACM Conference on Computer and Communications Security (CCS) 2006*, pages 201–210, 2006.

[36] J. Camenisch, S. Hohenberger, and A. Lysyanskaya. Compact e-cash. In *Proceedings of Eurocrypt 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 302–321. Springer-Verlag, 2005.

[37] J. Camenisch, S. Hohenberger, and M. Ø. Pedersen. Batch verification of short signatures. In *Proceedings of Eurocrypt 2007*, volume 4515 of *Lecture Notes in Computer Science*, pages 246–263. Springer-Verlag, 2007.

[38] J. Camenisch and A. Lysyanskaya. An efficient system for non-transferable anonymous credentials with optional anonymity revocation. In *Proceedings of Eurocrypt 2001*, volume 2045 of *Lecture Notes in Computer Science*, pages 93–118. Springer-Verlag, 2001.

[39] J. Camenisch and A. Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In M. Yung, editor, *Advances in Cryptology — CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 61–76. Springer Verlag, 2002.

[40] J. Camenisch and A. Lysyanskaya. A signature scheme with efficient protocols. In *Proceedings of SCN 2002*, volume 2576 of *Lecture Notes in Computer Science*, pages 268–289. Springer-Verlag, 2002.

[41] J. Camenisch and A. Lysyanskaya. Signature schemes and anonymous credentials from bilinear maps. In *Proceedings of Crypto 2004*, volume 3152 of *Lecture Notes in Computer Science*, pages 56–72. Springer-Verlag, 2004.

[42] J. Camenisch, A. Lysyanskaya, and M. Meyerovich. Endorsed e-cash. In *IEEE Symposium on Security and Privacy*, pages 101–115, 2007.

[43] J. Camenisch and V. Shoup. Practical verifiable encryption and decryption of discrete logarithms. In *Proceedings of Crypto 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 126–144. Springer-Verlag, 2003.

[44] D. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–88, Feb. 1981.

[45] D. Chaum. Blind signatures for untraceable payments. In *Proceedings of Crypto 1982*, Lecture Notes in Computer Science, pages 199–203. Springer-Verlag, 1982.

[46] D. Chaum. Blind signature systems. In *CRYPTO '83*, pages 153–156. Plenum, 1983.

[47] D. Chaum. Security without identification: transaction systems to make big brother obsolete. *Communications of the ACM*, 28(10):1030–1044, 1985.

[48] D. Chaum, B. den Boer, E. van Heyst, S. F. Mjølsnes, and A. Steenbeek. Efficient offline electronic checks (extended abstract). In *Proceedings of Eurocrypt 1989*, pages 294–301, 1989.

[49] D. Chaum and T. P. Pedersen. Transferred cash grows in size. In *Proceedings of Eurocrypt 1992*, volume 658 of *Lecture Notes in Computer Science*, pages 390–407. Springer-Verlag, 1992.

[50] X. Chen, Y. Jiang, and X. Chu. Measurements, analysis and modeling of private trackers. In *Peer-to-Peer Computing (P2P), 2010 IEEE Tenth International Conference on*, pages 1 –10, aug. 2010.

[51] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. PlanetLab: an overlay testbed for broad-coverage services. *SIGCOMM Comput. Commun. Rev.*, 33:3–12, July 2003.

[52] B. N. Chun, P. Buonadonna, A. AuYoung, C. Ng, D. C. Parkes, J. Shneidman, A. C. Snoeren, and A. Vahdat. Mirage: A microeconomic resource allocation system for sensornet testbeds. In *Proceedings of the 2nd IEEE Workshop on Embedded Networked Sensors*, 2005.

[53] D. E. Clarke, S. Devadas, M. van Dijk, B. Gassend, and G. E. Suh. Incremental multiset hash functions and their application to memory integrity checking. In *ASIACRYPT*, pages 188–207, 2003.

[54] B. Cohen. The BitTorrent protocol specification. `http://www.bittorrent.org/beps/bep_0003.html`.

[55] B. Cohen. Incentives build robustness in BitTorrent. In *Proc. 2th International Workshop on Peer-to-Peer Systems (IPTPS)*, Berkeley, CA, Feb. 2003.

[56] B. Cox, J. Tygar, and M. Sirbu. NetBill security and transaction protocol. In *Proceedings of the 1st Usenix Workshop on Electronic Commerce*, pages 77–88, 1995.

[57] J. Daemen and V. Rijmen. *Rijndael: AES – The Advanced Encryption Standard*. Springer-Verlag, 2002.

[58] A. Daly and W. Marnane. Efficient architectures for implementing montgomery modular multiplication and RSA modular exponentiation on reconfigurable logic. In *FPGA '02: Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*, pages 40–49, New York, NY, USA, 2002. ACM Press.

[59] I. Damgård. On sigma protocols. `http://www.daimi.au.dk/ivan/Sigma.pdf`.

[60] I. Damgård. Payment systems and credential mechanism with provable security against abuse by individuals. In *Proceedings of Crypto 1988*, volume 403 of *Lecture Notes in Computer Science*, pages 328–335. Springer-Verlag, 1988.

[61] I. Damgård, K. Dupont, and M. Ø. Pedersen. Unclonable group identification. In *Proceedings of Eurocrypt 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 555–572. Springer-Verlag, 2006.

[62] I. Damgård and E. Fujisaki. A statistically-hiding integer commitment scheme based on groups with hidden order. In *Proceedings of Asiacrypt 2002*, volume 2501 of *Lecture Notes in Computer Science*, pages 125–142. Springer-Verlag, 2002.

[63] I. Damgård, J. Groth, and G. Salomonsen. The theory and implementation of an electronic voting system. In *Proceedings of Secure Electronic Voting (SEC)*, pages 77–100, 2003.

[64] R. Dingledine, N. Mathewson, and P. Syverson. Reputation in P2P anonymity systems. In *P2PECON '03: Proceedings of Workshop on Economics of Peer-to-Peer Systems*, June 2003.

[65] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[66] R. Dingledine, N. Mathewson, and P. Syverson. Tor: the second-generation onion router. 2004.

[67] Distributed.net. `http://www.distributed.net`.

[68] Y. Dodis, S. Vadhan, and D. Wichs. Proofs of retrievability via hardness amplification. In O. Reingold, editor, *Theory of Cryptography Conference (TCC)*, volume 5444 of *Lecture Notes in Computer Science*. Springer, 2009.

[69] J. Douceur. The Sybil attack. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, 2002.

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

[71] C. C. Erway, A. Küpçü, C. Papamanthou, and R. Tamassia. Dynamic provable data possession. In E. Al-Shaer, S. Jha, and A. D. Keromytis, editors, *ACM Conference on Computer and Communications Security*, pages 213–222. ACM, 2009.

[72] B. Fan, D.-M. Chiu, and J. C. S. Lui. The delicate tradeoffs in BitTorrent-like file sharing protocol design. In *ICNP '06: Proceedings of the 14th IEEE International Conference on Network Protocols*, November 2006.

[73] U. Feige, D. Lapidot, and A. Shamir. Multiple non-interactive zero-knowledge proofs based on a single random string. In *Proceedings of 31st Symposium on Theory of Computing (STOC)*, pages 308–317, 1990.

[74] A. L. Ferrara, M. Green, S. Hohenberger, and M. Ø. Pedersen. Practical short signature batch verification. In *Proceedings of CT-RSA*, pages 309–324, 2009.

[75] A. Fiat and A. Shamir. How to prove yourself: practical solutions to identification and signature problems. In *Proceedings of Crypto 1986*, volume 263 of *Lecture Notes in Computer Science*, pages 186–194. Springer-Verlag, 1986.

[76] E. J. Friedman, J. Y. Halpern, and I. A. Kash. Efficiency and nash equilibria in a scrip system for P2P networks. In *EC '06: Proceedings of the 7th ACM conference on Electronic commerce*, pages 140–149, 2006.

[77] E. J. Friedman and P. Resnick. The social cost of cheap pseudonyms. *Journal of Economics and Management Strategy*, 10(2):173–199, 2001.

[78] E. Fujisaki and T. Okamoto. Statistical zero knowledge protocols to prove modular polynomial relations. In *Proceedings of Crypto 1997*, volume 1294 of *Lecture Notes in Computer Science*, pages 16–30. Springer-Verlag, 1997.

[79] D. L. Gazzoni and P. S. L. M. Barreto. Demonstrating data possession and uncheatable data transfer. Cryptology ePrint Archive, Report 2006/150, 2006.

[80] R. Gennaro, C. Gentry, and B. Parno. Non-interactive verifiable computing: outsourcing computation to untrusted workers. Cryptology ePrint Archive, Report 2009/547, 2009. `http://eprint.iacr.org/2009/547`.

[81] GMP. The GNU MP Bignum library. `http://gmplib.org`.

[82] O. Goldreich, S. Goldwasser, and S. Micali. How to construct random functions (extended abstract). In *Proceedings of 25th Symposium on the Foundations of Computer Science (FOCS)*, pages 464–479, 1984.

[83] 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 the ACM*, 38(3):691–729, 1991.

[84] D. M. Goldschlag, M. G. Reed, and P. F. Syverson. Onion routing for anonymous and private internet connections. *Communications of the ACM*, 42(2):84–88, February 1999.

[85] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. In *Proceedings of 17th Symposium on the Theory of Computing (STOC)*, pages 186–208, 1985.

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

[87] P. Golle, K. Leyton-Brown, and I. Mironov. Incentives for sharing in peer-to-peer networks. In *Proc. of the 2001 ACM Conference on Electronic Commerce*, 2001.

[88] M. T. Goodrich, C. Papamanthou, R. Tamassia, and N. Triandopoulos. Athos: Efficient authentication of outsourced file systems. In *ISC*, pages 80–96, 2008.

[89] 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.

[90] GreedyTorrent. `http://www.greedytorrent.com/`.

[91] J. Groth. Non-interactive zero-knowledge arguments for voting. In *ACNS*, volume 3531 of *Lecture Notes in Computer Science*, pages 467–482. Springer-Verlag, 2005.

[92] T. Heydt-Benjamin, H.-J. Chae, B. Defend, and K. Fu. Privacy for public transportation. In *Privacy Enhancing Technologies Symposium (PETS)*, pages 1–19, 2006.

[93] J. Ioannidis, S. Ioannidis, A. D. Keromytis, and V. Prevelakis. Fileteller: Paying and getting paid for file storage. In *Financial Cryptography, 6th International Conference*, 2002.

[94] A. Iosup, P. Garbacki, J. Pouweise, and D. Epema. Correlating topology and path characteristics of overlay networks and the internet. In *Proceedings of CCGRID 2006*, 2006.

[95] Ipoque. Internet study 2008/2009. `http://www.ipoque.com/resources/internet-studies/internet-study-2008_2009`.

[96] D. Irwin, J. Chase, L. Grit, and A. Yumerefendi. Self-recharging virtual currency. In *P2PECON '05: Proceeding of the 2005 ACM SIGCOMM workshop on Economics of peer-to-peer systems*, 2005.

[97] N. Ishida, S. Matsuo, and W. Ogata. Divisible voting scheme. In *Proceedings of ISC 2003*, volume 2851 of *Lecture Notes in Computer Science*, pages 137–150. Springer-Verlag, 2003.

[98] M. Jakobsson. Ripping coins for a fair exchange. In *Advances in Cryptology — EUROCRYPT '95*, volume 921, pages 220–230. Springer Verlag, 1995.

[99] A. Juels and B. S. Kaliski. PORs: Proofs of retrievability for large files. In *ACM Conference on Computer and Communications Security*, pages 584–597, 2007.

[100] S. Jun and M. Ahamad. Incentives in BitTorrent induce free riding. In *P2PECON '05: Proceeding of the 2005 ACM SIGCOMM workshop on Economics of peer-to-peer systems*, pages 116–121, 2005.

[101] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable secure file sharing on untrusted storage. *FAST*, pages 29–42, 2003.

[102] I. A. Kash, E. J. Friedman, and J. Y. Halpern. Optimizing scrip systems: Efficiency, crashes, hoarders, and altruists. In *EC '07: Proceedings of the 8th ACM conference on Electronic commerce*, 2007.

[103] I. A. Kash, J. K. Lai, A. Zohar, and H. Zhang. Economics of BitTorrent communities. `http://people.seas.harvard.edu/~kash/papers/communities.pdf`.

[104] J. Kleinberg and P. Raghavan. Query incentive networks. In *FOCS '05: Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science*, 2005.

[105] D. Kravets. Seeking world record, pirate bay claims 22 million users. *Wired Magazine*, November 2008. `http://blog.wired.com/27bstroke6/2008/11/pirate-bay-seek.html`.

[106] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: an architecture for global-scale persistent storage. *SIGPLAN Not.*, 35(11):190–201, 2000.

[107] A. Küpçü and A. Lysyanskaya. Usable optimistic fair exchange. In *Proceedings of CT-RSA 2010*, volume 5985 of *Lecture Notes in Computer Science*, pages 252–267. Springer-Verlag, 2010.

[108] K. Lai, L. Rasmusson, E. Adar, L. Zhang, and B. Huberman. Tycoon: An implementation of a distributed, market-based resource allocation system. *Multiagent and Grid Systems*, 1(3):169–182, 2005.

[109] B. Laurie and B. Clifford. Stupid: a meta-language for cryptography, 2010. `http://code.google.com/p/stupid-crypto`.

[110] A. Legout, N. Liogkas, E. Kohler, and L. Zhang. Clustering and sharing incentives in BitTorrent systems. In *Proc. of ACM SIGMETRICS'07*, June 2007.

[111] D. Levin, K. LaCurts, N. Spring, and B. Bhattacharjee. BitTorrent is an auction: analyzing and improving BitTorrent's incentives. In *Proceedings of SIGCOMM 2008*, pages 243–254, 2008.

[112] J. Lewis and B. Martin. Cryptol: high assurance, retargetable crypto development, and validation. In *Proceedings of Military Communications Conference 2003*, pages 820–825, 2003.

[113] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Dynamic authenticated index structures for outsourced databases. In *SIGMOD*, pages 121–132, 2006.

[114] H. Li, A. Clement, E. Wong, J. Napper, I. Roy, L. Alvisi, and M. Dahlin. BAR gossip. In *Proceedings of the 2006 USENIX Operating Systems Design and Implementation (OSDI)*, Nov. 2006.

[115] J. Li, M. Krohn, D. Mazieres, and D. Shasha. Secure Untrusted Data Repository (SUNDR). *OSDI*, pages 121–136, 2004.

[116] N. Liogkas, R. Nelson, E. Kohler, and L. Zhang. Exploiting BitTorrent for fun (but not profit). In *Proc. 5th International Workshop on Peer-to-Peer Systems (IPTPS)*, Santa Barbara, CA, Feb. 2006.

[117] H. Lipmaa. On Diophantine complexity and statistical zero-knowledge arguments. In *Proceedings of Asiacrypt 2003*, volume 2894 of *Lecture Notes in Computer Science*, pages 398–415. Springer-Verlag, 2003.

[118] H. Lipmaa, N. Asokan, and V. Niemi. Secure vickrey auctions without threshold trust. In *Proceedings of Financial Cryptography 2002*, volume 2357 of *Lecture Notes in Computer Science*, pages 87–101. Springer-Verlag, 2002.

[119] T. Locher, P. Moor, S. Schmid, and R. Wattenhofer. Free Riding in BitTorrent is Cheap. In *5th Workshop on Hot Topics in Networks (HotNets), Irvine, California, USA*, November 2006.

[120] B. Lynn. *On the Implementation of Pairing-Based Cryptography*. PhD thesis, Stanford University. `http://crypto.stanford.edu/pbc`.

[121] U. Maheshwari, R. Vingralek, and W. Shapiro. How to build a trusted database system on untrusted storage. In *OSDI*, pages 10–26, Berkeley, CA, USA, 2000. USENIX Association.

[122] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay - a secure two-party computation system. In *Proceedings of USENIX Security 2004*, pages 287–302, 2004.

[123] S. Marti and H. Garcia-Molina. Identity crisis: Anonymity vs. reputation in P2P systems. In *P2P '03: Proceedings of the 3rd International Conference on Peer-to-Peer Computing*, 2003.

[124] S. Marti and H. Garcia-Molina. Taxonomy of trust: Categorizing P2P reputation systems. *Computer Networks*, 50(4):472–484, 2006.

[125] S. Meiklejohn, C. C. Erway, A. Küpçü, T. Hinkle, and A. Lysyanskaya. ZKPDL: a language-based system for efficient zero-knowledge proofs and electronic cash. In *Proceedings of USENIX Security 2010*, pages 193–206, 2010.

[126] A. J. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

[127] R. Merkle. A digital signature based on a conventional encryption function. In *Advances in Cryptology — CRYPTO '87*, pages 269–278, 1987.

[128] M. Meulpolder, L. D'Acunto, M. Capotă, M. Wojciechowski, J. A. Pouwelse, D. H. J. Epema, and H. J. Sips. Public and private BitTorrent communities: a measurement study. In *Proceedings of the 9th international conference on Peer-to-peer systems*, IPTPS'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[129] N. Michalakis, R. Soulé, and R. Grimm. Ensuring content integrity for untrusted peer-to-peer content distribution networks. In *Proceedings of the 4th USENIX conference on Networked Systems Design and Implementation*, NSDI'07, Berkeley, CA, USA, 2007. USENIX Association.

[130] G. Miller. Riemann's hypothesis and tests for primality. In *ACM Symposium on Theory of Computing*, pages 234–239, 1975.

[131] D. Molnar. The seti@home problem. *ACM Crossroads*, Sep 2000.

[132] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen. Ivy: A Read/Write Peer-to-Peer File System. *OSDI*, pages 31–44, 2002.

[133] A. Nandi, T.-W. J. Ngan, A. Singh, P. Druschel, and D. S. Wallach. Scrivener: Providing incentives in cooperative content distribution systems. In *Proceedings of the ACM/IFIP/USENIX 6th International Middleware Conference (Middleware 2005)*, Grenoble, France, Nov. 2005.

[134] M. Naor and K. Nissim. Certificate revocation and certificate update. In *USENIX Security*, pages 17–17, 1998.

[135] M. Naor and G. N. Rothblum. The complexity of online memory checking. In *FOCS*, pages 573–584, 2005.

[136] M. Naor and M. Yung. Universal one-way hash functions and their cryptographic applications. In *Proceedings of 21st Symposium on Theory of Computing (STOC)*, pages 33–43, 1989.

[137] L. Nguyen and R. Safavi-Naini. Dynamic k-times anonymous authentication. In *ACNS*, volume 3531 of *Lecture Notes in Computer Science*, pages 318–333. Springer-Verlag, 2005.

[138] A. Oprea, M. Reiter, and K. Yang. Space-Efficient Block Storage Integrity. *NDSS*, 2005.

[139] J. Ousterhout. Tcl/tk. `http://www.tcl.tk/`.

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

[141] C. Papamanthou and R. Tamassia. Time and space efficient algorithms for two-party authenticated data structures. In *ICICS*, pages 1–15, 2007.

[142] C. Papamanthou, R. Tamassia, and N. Triandopoulos. Authenticated hash tables. In *ACM Conference on Computer and Communications Security*, pages 437–448, 2008.

[143] T. Parr. ANTLR parser generator. `http://www.antlr.org`.

[144] Patch-free-processing. `http://web.archive.org/web/20070207064618/http://home.hccnet.nl/a.alfred/p-free-p1pfp.html`.

[145] T. P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In J. Feigenbaum, editor, *Advances in Cryptology – CRYPTO '91*, volume 576 of *Lecture Notes in Computer Science*, pages 129–140. Springer Verlag, 1992.

[146] M. Piatek, T. Isdal, T. Anderson, A. Krishnamurthy, and A. Venkataramani. Do incentives build robustness in BitTorrent? In *Proceedings of the 4th USENIX conference on Networked Systems Design and Implementation*, NSDI'07, Berkeley, CA, USA, 2007. USENIX Association.

[147] M. Piatek, T. Isdal, A. Krishnamurthy, and T. Anderson. One hop reputations for peer to peer file sharing workloads. In *Proceedings of NSDI 2008*, pages 1–14, 2008.

[148] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.

[149] M. Rabin and J. Shallit. Randomized algorithms in number theory. *Communications on Pure and Applied Mathematics*, 39(1):239–256, 1986.

[150] A. Ramachandran, A. D. Sarma, and N. Feamster. Bitstore: an incentive-compatible solution for blocked downloads in BitTorrent. In *Proceedings of NetEcon + IBC 2007*, 2007.

[151] A. Rapoport. Prisoner's dilemma - recollections and observations. *Game Theory as a Theory of Conflict Resolution*, 1974.

[152] O. Regev and N. Nisan. The POPCORN market. Online markets for computational resources. *Decision Support Systems*, 28(1-2):177–189, 2000.

[153] M. Reiter, X. Want, and M. Wright. Building reliable mix networks with fair exchange. In *Applied Cryptography and Network Security: Third International Conference*, pages 378–392. Lecture Notes in Computer Science, June 2005.

[154] L. Rizzo. Dummynet. `www.iet.unipi.it/~luigi/dummynet`.

[155] R. Rodrigues and P. Druschel. Peer-to-peer systems. *Commun. ACM*, 53:72–82, October 2010.

[156] Rosetta@home. `http://boinc.bakerlab.org/rosetta/`.

[157] Samba. Samba.org CVS repository. `http://cvs.samba.org/cgi-bin/cvsweb/`.

[158] Sandvine. Fall 2010 global internet phenomena report. `http://www.sandvine.com/news/global_broadband_trends.asp`.

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

[160] B. Schoenmakers. Interval proofs revisited. In *International Workshop on Frontiers in Electronic Elections*, 2005.

[161] T. Schwarz and E. Miller. Store, Forget, and Check: Using Algebraic Signatures to Check Remotely Administered Storage. *ICDCS*, page 12, 2006.

[162] 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.

[163] Seti@home. `http://setiathome.berkeley.edu`.

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

[165] M. Sirivianos, J. H. Park, X. Yang, and S. Jarecki. Dandelion: cooperative content distribution with robust incentives. In *Proc. USENIX 2007 Annual Technical Conference*, June 2007.

[166] M. Stonebraker, P. M. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu. Mariposa: A wide-area distributed database system. *VLDB Journal: Very Large Data Bases*, 5(1):48–63, 1996.

[167] D. Stutzbach and R. Rejaie. Understanding churn in peer-to-peer networks. In *Proceedings of IMC 2006*, pages 189–202, 2006.

[168] R. Tamassia. Authenticated data structures. In G. D. Battista and U. Zwick, editors, *ESA*, volume 2832 of *Lecture Notes in Computer Science*, pages 2–5. Springer, 2003.

[169] R. Tamassia and N. Triandopoulos. Computational bounds on hierarchical data processing with applications to information security. In L. Caires, G. F. Italiano, L. Monteiro, C. Palamidessi, and M. Yung, editors, *ICALP*, volume 3580 of *Lecture Notes in Computer Science*, pages 153–165. Springer, 2005.

[170] R. Thommes and M. Coates. BitTorrent fairness: analysis and improvements. In *WITSP '05: Proceedings of the 4th Workshop on the Internet, Telecommunications and Signal Processing*, December 2005.

[171] truXoft Calibrating BOINC Core Client. `http://boinc.truxoft.com/core-cal.htm`.

[172] V. Vishnumurthy, S. Chandrakumar, and E. G. Sirer. Karma: A secure economic framework for P2P resource sharing. In *Proc. of Workshop on the Economics of Peer-to-Peer Systems*, 2003.

[173] C. Waldspurger, T. Hogg, B. Huberman, J. Kephart, and W. Stornetta. Spawn: A Distributed Computational Economy. *IEEE Transactions on Software Engineering*, 18(2):103–117, 1992.

[174] M. Walfish, J. Zamfirescu, H. Balakrishnan, D. Karger, and S. Shenker. Distributed quota enforcement for spam control. In *Proc. 3rd USENIX/ACM Symposium on Networked Systems Design and Implementation*, May 2006.

[175] B. Wilcox-O'Hearn. Experiences deploying a large-scale emergent network. In *Proc. 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, Cambridge, MA, Mar. 2002.

[176] H. Yu, M. Kaminsky, P. B. Gibbons, and A. D. Flaxman. SybilGuard: Defending against sybil attacks via social networks. *IEEE/ACM Trans. Networking*, 16(3):576–589, 2008.

[177] B. Zhu and S. Jajodia. Building trust in peer-to-peer systems: a review. *International Journal of Security and Networks*, 1(1/2):103–112, 2006.