Usable Optimistic Fair Exchange

Alptekin Küpçü and Anna Lysyanskaya
Brown University
Providence, RI, USA
{kupcu,anna}@cs.brown.edu

Abstract

Fairly exchanging digital content is an everyday problem. It has been shown that no fair exchange can be done without a trusted third party. Yet, even with a trusted party, it is still non-trivial to come up with an efficient solution, especially one that can be used in a p2p file sharing system with a high volume of data exchanged.

We provide an efficient optimistic fair exchange (barter) mechanism for exchanging digital files, where receiving a payment in return to a file (buying) is also considered fair. The exchange is optimistic, removing the need for a trusted party's involvement unless a dispute occurs. Previous protocols used an expensive primitive called verifiable encryption for every exchange. Our protocol's unique property is that, we use this expensive primitive only once between peers, and when used for exchanging multiple files, the efficiency of our protocol is comparable to a very simple (unfair) file exchange, while being provably secure and privacy respecting (when payments are made using e-cash).

1 Introduction

Fairly exchanging digital content is an everyday problem. A fair exchange scenario commonly involves Alice and Bob. Alice has something that Bob wants, and Bob has something that Alice wants. A fair exchange protocol guarantees that at the end either each of them obtains what (s)he wants, or neither of them does (see Micali's exposition [35] for more details and examples).

In this paper, we consider a general file exchange (bartering) scenario, inspired by BitTorrent [21] peer-to-peer file sharing protocol. Alice has several files (BitTorrent blocks) of interest to Bob, and Bob has several files (blocks) of interest to Alice. They do not know ahead of time how many or which files they will end up exchanging. They want to perform a fair exchange: Alice should get Bob's file (block) if and only if Bob gets Alice's file (block). In our scenario, we are assuming that Alice/Bob will be equally happy to get a payment in return to her/his file. Thus, exchanging a file/block with a payment (buying) is also considered fair.

Previous Work: It is well-known that **a fair exchange protocol is impossible without a trusted third party** (TTP) [38] that ensures that Alice cannot take advantage of Bob, and vice versa. Without loss of generality, Alice will have to send the last message of the protocol, and we want to protect Bob in case she chooses not to do so. From now on, we will refer to this trusted third party as the *Arbiter*. Without an arbiter, gradual release type of protocols where parties send pieces to each other in rounds can provide only weaker forms of fairness, and are much less efficient [11, 13].

Luckily, the impossibility result [38] *does not require that the Arbiter be involved in each transaction, but simply that the Arbiter exists*. If Alice and Bob are both well-behaved, there is no need for the Arbiter to do anything (or even know an exchange took place). Micali [34], Asokan,

Schunter, Waidner [2], and Asokan, Shoup and Waidner [4, 3] investigated this *optimistic* fair exchange scenario in which the Arbiter gets involved only in case of a dispute. Two such protocols [4, 28] were analyzed in [41] (see also [7]).

Asokan, Shoup and Waidner (ASW) gave the first provably secure and completely fair optimistic exchange protocol for exchanging *digital signatures* [4]. Later on, Belenkiy *et al.* [8] gave a protocol for buying digital content in exchange for e-cash, building on top of the ASW protocol. They provided an optimization for the Arbiter so that, unlike in the ASW protocol, the amount of work that the Arbiter is required to do depends only logarithmically on the size of the file. We will make use of their optimization, as defined in Appendix A. They also assume there is an additional TTP (which we call the *Tracker*) that provides a means of verification that the file actually contains the right content (*e.g.,* using hashes). Such entities certifying the hashes exist in current BitTorrent systems [21].

Belenkiy *et al.* [8] used e-cash (introduced by Chaum [19]), in particular, endorsed e-cash [17] in their constructions. The reason is that other forms of payments (signatures or electronic checks used in [4, 32]) do not provide any privacy. In our protocols, any form of payment can be employed, but we will also use endorsed e-cash as our sample instantiation since it is efficient and anonymous. See Section 3.6 and Appendix C for more discussion on employing different payment systems.

Contributions: We present the most efficient fair exchange known to us, where **the efficiency is comparable to a simple unfair exchange if performed multiple times between same pair of users**. Using the Belenkiy *et al.* barter protocol [8], n files/blocks can be exchanged using $n/2$ transactions, each of which requires a costly step involving a verifiable escrow and an e-coin. Our contribution is a very efficient fair exchange protocol where this can be done using only *one* (or *two* if we do not want to employ timeouts) step in total that involves the same expensive primitives (verifiable escrow and payment). This means that, with no (*i.e.* negligible) efficiency loss, our fair exchange protocol can be used to exchange files instead of the unfair protocol currently used by BitTorrent or similar file sharing protocols. We stress the fact that the timeouts used for dispute resolution purposes in one of our protocols can be very large (*e.g.,* one day or week) to allow for unexpected situations in which the participants act honestly (*e.g.,* network failure), and thus require very loose synchronization (*e.g.,* one hour difference), and users can freely participate in other exchanges without waiting for the timeout.

We take the idea of using verifiable escrow from ASW [4], and the subprotocols of Belenkiy *et al.* [8] that increase the efficiency of the Arbiter and the Tracker (see Appendix A). The Arbiter does *absolutely no work* in our protocols, as long as no dispute occurs. Our protocols can make use of any type of payments, but we will show an instantiation using e-cash since it also provides privacy. Our performance evaluation numbers will use endorsed e-cash [17] as the payment mechanism. Note that other (non-anonymous) forms of payments (*e.g.,* electronic checks) will be more efficient.

Our additional contribution is definitional. We give **a general definition for fair exchange** of digital content (not just digital signatures) *provided that it can be verified using some verification algorithm*. We then prove our protocol's security based on that definition. We sum up the most important properties of our protocols below.

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

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

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

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

We discuss the efficiency of our protocols and our initial implementation results in Sections 3.4 and 4.2. Discussion of limitations and future work can be found in Appendix D.

2 Definitions

Barter is an exchange of two items, which are digital files in our case. We assume that the reader is familiar with encryption and signature schemes, and hash functions. Further required definitions and notation are given below, although partially, omitting the details not necessary for understanding this paper.

2.1 Notation

An escrow is a ciphertext under the public key of some trusted third party (TTP). A *verifiable* escrow [4, 18, 15] means that the contents of the ciphertext can be verified by the recipient to satisfy some relation (therefore stating that the ciphertext contains the expected content). A contract (a.k.a. label, condition, or tag) attached to such a ciphertext defines the conditions under which the TTP should decrypt and give away the encrypted secret [42]. The label is public and it is integrated with the ciphertext in a such way that it cannot be modified. We will use $E_{Arb}(a; b)$ to denote an escrow of the secret a under the Arbiter's public key, with the contract b. The contract and the secret can contain multiple values. Similarly, $VE_{Arb}(a; b)$ will denote a verifiable escrow.

Any payment protocol that can efficiently be verifiably escrowed and is secure can be used in our protocols. Furthermore, if privacy is desired, the payments should be anonymous as in e-cash [19]. We provide an instantiation using endorsed e-cash [17] (which is an extension of compact e-cash [16]), since it satisfies all these requirements. Endorsed e-cash splits a coin into an unendorsed coin (denoted $coin'$) and endorsement (denoted *end*). One can think of $coin'$ as an encrypted coin and *end* as the key. One can check if the endorsement *end* in a given verifiable escrow [18] matches the given unendorsed coin $coin'$ (without learning the endorsement *end*). Furthermore, given only the unendorsed part $coin'$, no other party (except the owner) can come up with a valid endorsement *end*. Endorsed e-cash moreover has the ability to catch *double-spenders*. Hence, if one uses two different $coin'$, *end* pairs trying to spend the same coin twice, (s)he will be caught and punished. Note that if a party tries to deposit the same coin twice (using the same $coin'$, *end* pair), the operation can easily be denied by checking against a list of past transactions. Lastly, only matching $coin'$, *end* pairs can be linked, unendorsed coins and endorsements prepared for different exchanges remain unlinkable.

Wherever used, K_P will denote a symmetric key of party P, generated through an encryption scheme's key generation algorithm. We let $c = Enc_K(f)$ denote that the ciphertext c is an encryption of the plaintext f under the symmetric key K. Similarly, $f = Dec_K(c)$ will denote that the plaintext f is the decryption of the ciphertext c under the symmetric key K. Our protocol can make use of any secure symmetric encryption scheme (see the book by Katz and Lindell [31] for definitions and constructions).

Let pk_P and sk_P denote public and secret keys for party P (they can denote the keys for any asymmetric scheme, like a public-key encryption or a signature scheme). Then $sign_{sk}(x)$ will denote a signature on x under the secret key sk which can be verified using the corresponding public key pk. Our protocol can make use of any secure public-key encryption scheme [23, 26] and any secure signature scheme [29].

Furthermore, let H_k be a family of (universal one-way) hash functions [36], where k is the security parameter, and uniformly choose a hash function $hash \leftarrow H_k$ from the family. Then, $h_x = hash(x)$ will denote that h_x is the hash of x under the hash function $hash$. We now introduce a definition we frequently use in the paper.

Definition 1. *We say that a key K **decrypts correctly**, or is the **correct key** with respect to a plaintext hash h_f and a ciphertext c, if the plaintext $f' = Dec_K(c)$ has the property $hash(f') = h_f$.*

ASW [4], Belenkiy *et al.* [8] and BitTorrent [21] all assume existence of a TTP called the *Tracker*[1]. In general, the Tracker provides a *verification algorithm* to Alice. She trusts that any input accepted by that verification algorithm will be the content she wants (similar for Bob). So, the exchange takes place on materials which are accepted by the algorithms given by the Tracker to the participants.

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

2.2 (Optimistic) Fair Exchange

In this section we will give a general definition of fair exchange. Unlike in ASW, our definitions will not be specific to signature exchange, and we will consider the general case of multiple exchanges. Furthermore, we separate and clearly define the roles of all trusted parties. While providing models and definitions for a general framework of (optimistic) fair exchange applicable to a broad range of protocols, we will also show its extensions to our case. We start by defining the model we use.

Model: The model is adapted from the ASW definition [4], with clarifications and generalizations. There are three players; *Alice* and *Bob* exchanging two digital materials, and the *Arbiter*[2] for conflict resolution. All players are assumed to be polynomial time interactive Turing machines. We make no assumption about the underlying network capability.[3] Any message that does not confirm with the protocol specification will be discarded by the honest parties. Any input which does not verify according to the protocol will be resolved as stated by the protocol or the protocol will be aborted if no resolution is applicable. It is important that the Arbiter resolves conflicts on the same exchange *atomically*.[4] Thus, it will only interact with either Alice or Bob at any given time instance, until that interaction ends as specified by the protocol.[5] Sensitive communication (*e.g.,* exchange of decryption keys for files) will be carried out over a secure (and possibly authenticated) channel (*e.g.,* SSL can be used to connect to the Arbiter, a secure key exchange with no public key infrastructure can be used for the communication between Alice and Bob).

[1] Called just TTP in ASW.

[2] One of the TTPs in ASW.

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

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

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

For protocols using a *timeout*[6], we assume that the adversary cannot prevent the honest party from reaching the Arbiter before the timeout. If no timeouts are defined, we assume the adversary cannot prevent the honest party from reaching the Arbiter eventually. Hence, the honest party is assumed to be able to reach the Arbiter as defined by the protocol. Even with timeouts, this is not an unrealistic assumption since our timeouts can be large (*e.g.,* one day or week).

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

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

Definition 2. *Let* SP *denote the security parameters of the system (e.g., key lengths of the primitives used). Let* PP *denote all the public values in the system, including* SP*, public keys of the trusted parties, and possibly some public parameters. Let* $\mathsf{PPGen}(\mathsf{SP})$ *be the randomized procedure which generates the public values given the security parameters. Then, define our* $\mathsf{PP} = (pk_{arb}, pk_{bank}, pk_{tracker}, timeout, \mathsf{SP}$*, and additional parameters for primitives used).*

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

Assume that the content to be exchanged and associated verification algorithms are output by a generation algorithm $(f_H^{(1)}, V_{f_H^{(1)}}, pub_{f_H^{(1)}}, .., f_H^{(n)}, V_{f_H^{(n)}}, pub_{f_H^{(n)}}) \leftarrow \mathsf{Gen}(\mathsf{SP})$ that takes the security parameters as input and outputs some content to be exchanged, with associated verification algorithms, and possibly some public information about the content. This procedure involves a party H and the Tracker.

Definition 3. *Content and verification algorithms are secure if* $\forall$ *PPT adversaries* $\mathcal{A}$ *and* $\forall$ *auxiliary inputs* $z \in \{0,1\}^{poly(\mathsf{SP})}$ *we have (over the randomness of the generation algorithms, the adversary, and possibly the verification algorithms)*

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

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

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

[8] Alice and Bob both trust the Tracker that whatever files that can pass this verification algorithms will be the content they would like.

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

To provide evidence on the generality and applicability of our definition, we present several example verification algorithms for various tasks. For example, a file verification can be performed using hashes. So, each verification algorithm $V_{f_A^{(i)}}$ for Alice's file[10] $f_A^{(i)}$ contains the definition of hash function used –*hash*–[11], and the hash value $h_{f_A^{(i)}} = hash(f_A^{(i)})$. The ith verification algorithm computes the hash of the given input according to the description of the hash function, and accepts it if and only if the computed hash matches $h_{f_A^{(i)}}$. The security of each verification algorithm is proved using the security of universal one-way hash functions [36] in Appendix B. As another example, consider the ASW signature exchange protocol, in which each verification algorithm contains the signature scheme's description[11], the signature public key of Alice pk_A[11], and the message m_i to be signed. When it receives a signature as input, the ith verification accepts the signature if and only if it is a valid signature on message m_i under the public key pk_A using the signature scheme. As yet another example, an e-coin verification algorithm can take a coin to verify, and use the Bank's public key while verifying the non-interactive proofs given. Such an algorithm is a part of the specification of every e-cash scheme (*e.g.,* see [17, 16]). Verifiable encryption schemes (*e.g.,* [18]) and, in general, proof systems also specify a verification algorithm in their definitions. Such algorithms can be used directly in a fair exchange protocol, satisfying our definition as long as they are secure according to definition 3.

To summarize, in the setup phase, public values are generated using $\mathsf{PPGen}(\mathsf{SP})$, and Alice and Bob both use the $\mathsf{Gen}(\mathsf{SP})$ procedure to come up with their contents and verifications (jointly with the Tracker). The public information leaked from the generation procedure can be used by the adversary. Even though we abstracted out $\mathsf{PPGen}(\mathsf{SP})$ and $\mathsf{Gen}(\mathsf{SP})$ as two procedures, in reality their implementation can be broken down further into multiple procedures. $\mathsf{PPGen}(\mathsf{SP})$ can be realized as the Arbiter, the Tracker, and the Bank all generating their public keys independently and publishing them. $\mathsf{Gen}(\mathsf{SP})$ can have different components. The files may be created by some content generator (*e.g.,* movie distributor, Alice herself, etc.), and the verification algorithm may be generated by the Tracker, jointly with the content generator. The Tracker as a trusted entity needs to be involved in the generation of verification algorithms since the setup phase needs to be trusted. From now on, we assume the content and verification algorithms used are secure.

Definition 4. ***Fair Exchange Protocol****: A fair exchange protocol is composed of three interactive algorithms: Alice running algorithm A, Bob running algorithm B, and the Arbiter running the trusted algorithm T. The content and verification algorithms used need to be secure according to definition 3. The security of the exchange is then defined in terms of completeness and fairness.*

Completeness for a (non-optimistic) fair exchange states that the interactive run of A, B and T by *honest parties* results in A getting f_B and B getting f_A (assuming an ideal network):

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

[9] For example, if movies are being exchanged, a lot of information is publicly available about such a movie file, such as actors, length, and release date.

[10] Files are generated from a distribution with sufficient entropy.

[11] Possibly different for each verification algorithm

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

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

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

Fairness states that at the end of the protocol, either Alice and Bob both get content that passes the verification algorithms given to them, or neither Alice nor Bob gets anything that passes the verification, in each of the n exchanges. This definition is easy to satisfy using a (non-optimistic) fair exchange protocol since Alice and Bob can both hand their files to the Arbiter, and then the Arbiter can send Bob's files to Alice and Alice's files to Bob, if they pass respective verifications. Thus, below, we will define the more interesting case, the *fairness for an optimistic fair exchange.* It is important to note that the ASW definition of fairness applies only to a single exchange, whereas our definition covers polynomially many exchanges.

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

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

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

Fairness Game: There are three types of interaction in our fairness game. Type 1 interactions are between H and $\mathcal{A}$. Type 2 interactions are between H and T. Type 3 interactions are between $\mathcal{A}$ and T.[12] The adversary can arbitrarily interleave type $1, 2, 3$ interactions, but cannot prevent type 2 interactions from happening until the timeout if timeouts are used, or eventually otherwise. The

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

game ends when the honest party H produces its final output (including aborts and resolutions) in all the started protocols. Without loss of generality, we assume both parties want to exchange different content in different exchanges ($\forall i \neq j \quad f_H^{(i)} \neq f_H^{(j)}$ and $f_\mathcal{A}^{(i)} \neq f_\mathcal{A}^{(j)}$ and $\forall i, j \quad f_H^{(i)} \neq f_\mathcal{A}^{(j)}$).[13]

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

Our protocol with payments has a very straightforward generalization of the fairness property. We will abuse the notation now and say that the verification algorithm provided by the Tracker in the setup phase has two parts. One is the file verification denoted $V_{f_H^{(i)}}$, and the other is the payment verification denoted $V_{c_H^{(i)}}$. Then, we can define our fairness condition as $\forall i : 1..n \quad [\exists x \in X : V_{f_\mathcal{A}^{(i)}}(x) = accept \Rightarrow (\exists y \in Y : V_{f_H^{(i)}}(y) = accept \quad XOR \quad \exists y \in Y : V_{c_H^{(i)}}(y) = accept)] \wedge [\exists y \in Y : V_{f_H^{(i)}}(y) = accept \Rightarrow (\exists x \in X : V_{f_\mathcal{A}^{(i)}}(x) = accept \quad XOR \quad \exists x \in X : V_{c_\mathcal{A}^{(i)}}(x) = accept)]$ stating that if either party obtains the other party's file, the other party obtains either the file or payment of that party. We believe that a broad range of optimistic fair exchange protocols can adapt the definition above using straightforward extensions whenever necessary.

TIMELY RESOLUTION: Lastly, as pointed out by ASW [4], an optimistic fair exchange protocol must provide timely resolution: Alice and Bob must be able to have disputes resolved within a finite and limited time. In our protocol without timeouts, resolution is immediate. In our protocol with timeouts, we guarantee resolution at the timeout (which is finite and fixed). We furthermore show that timeouts do not render our system less usable (Alice and Bob can freely participate in other exchanges without waiting for the timeout), and so in general we can use our more efficient protocol with timeouts.

We now present two different barter protocols, one that employs timeouts (Section 3), and one that does not (Section 4). Both of our protocols are $O(n)$ times more efficient than previous protocols [4, 3, 2, 5, 35, 8, 17], and almost as efficient as an unfair exchange, when n files or blocks are exchanges, while still being provably fair.

3 Efficient Optimistic Barter Protocol

3.1 Barter with Timeouts

We will show a particular instantiation of our protocol, and then point out how to generalize it easily, in Section 3.6. Before the protocol begins (this is the setup phase), we assume Alice has withdrawn an e-coin from the Bank. Any time Alice and Bob wants to exchange two files, Alice generates her fresh key K_A and Bob generates his fresh key K_B for a symmetric encryption scheme. Alice and Bob both have their files (f_A, f_B), have the encrypted versions of their files $(c_A = Enc_{K_A}(f_A), c_B = Enc_{K_B}(f_B))$, have the hashes of their files and encryptions (Alice has $h_{f_A} = hash(f_A), h_{c_A} = hash(c_A)$, and Bob has $h_{f_B} = hash(f_B), h_{c_B} = hash(c_B)$). Besides, the Tracker provides them with the respective verification algorithms: Alice gets h_{f_B}, Bob gets h_{f_A}.[14]

[13] If the honest party already has the adversary's file, it does not make sense to exchange it, even though the exchange will be trivially fair due to the completeness property. If the adversary already has the honest party's file, then there is no hope for fairness since the adversary can just abort the protocol but he already has the file. Similar arguments hold for exchanging the same file multiple times.

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

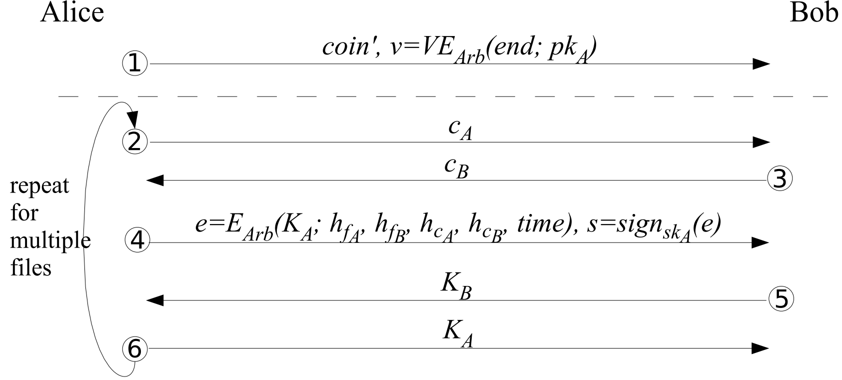

Figure 1: Our Barter Protocol with Timeouts

Everyone uses the same time zone (*e.g.,* GMT), and the *timeout* is a globally known parameter. If anything goes wrong prior to step 5 (no resolution protocol is applicable), the protocol will be aborted. The protocol proceeds as follows (summarized in Figure 1):

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

2. Alice sends Bob ciphertext c_A of her file.[15] Bob calculates $h_{c_A} = hash(c_A)$.

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

4. Alice sends Bob an escrow $e = E_{Arb}(K_A; h_{f_A}, h_{f_B}, h_{c_A}, h_{c_B}, time)$ and her signature $s = sign_{sk_A}(e)$ on that escrow. The escrow e should encrypt a key and should be labeled with four hash values $h_{f_A}, h_{f_B}, h_{c_A}, h_{c_B}$, and a *time* value. If any of the hash values do not match Bob's knowledge of those values, or if the *time* value is deviated too much from Bob's knowledge of the time (*e.g.,* almost one timeout difference), then Bob aborts.[16] Moreover, if the signature s on the escrow e does not verify with the public key pk_A sent in step 1 as part of the verifiable escrow v, Bob aborts the protocol.

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

6. Alice sends Bob her key K_A. Bob checks if the key K_A decrypts the ciphertext c_A correctly. If not, he runs *BobResolve*; he must do so before the timeout.[17]

Once step 1 is completed, **cheap steps 2-6 can be repeated to exchange more files**, as long as no dispute occurs. Alice and Bob need not know beforehand how many or which files/blocks to exchange. Below we present the resolution protocols in case of a dispute between Alice and Bob. *The Arbiter never gets involved in a transaction unless there is a dispute.*

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

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

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

3.2 BobResolve

Bob needs to contact the Arbiter before the timeout for resolution (current time $<$ *time* in escrow e + *timeout*), since otherwise the Arbiter is not going to honor his request. Assuming Bob resolves before the timeout, he provides the Arbiter with the escrow e and signature s that he received in step 4, and also the verifiable escrow v he received in step 1 from Alice. The escrow e should be labeled with four hash values $h_{f_A}, h_{f_B}, h_{c_A}, h_{c_B}$, and a *time* value. The verifiable escrow v should be labeled with a public key pk_A for a signature scheme. If the labels of the escrows are ill-formed, the Arbiter will not honor the request. The Arbiter checks the signature s using the public key in the verifiable escrow v, and if it verifies he asks Bob to present his correct key K_B that verifies using the VerifyKey protocol in Appendix A (*i.e.* it decrypts a ciphertext with hash h_{c_B} to a plaintext with hash h_{f_B}). If Bob succeeds in giving the correct key, the Arbiter stores the key K_B, decrypts the escrow e and hands in the key K_A from the escrow to Bob. Bob checks if K_A decrypts Alice's file f_A correctly. If not, he proves this to the Arbiter using the technique in Appendix A and gets the endorsement *end* in the verifiable escrow v from the Arbiter.[18] Notice that only Bob may succeed in the BobResolve protocol with the Arbiter because any other party will fail to provide the correct key matching hashes of Bob's files (see Appendix B).

3.3 AliceResolve

When Alice contacts the Arbiter for resolution, she asks for Bob's key K_B. If such a key exists, then the Arbiter sends K_B to her.[19] K_B has already been verified, so Alice does not need to perform any further action. If such a key does not exist yet, Alice should come back after the timeout. If, even after the timeout K_B does not exist, then Alice is assured that it will never exist, and can consider that particular trade as aborted.

3.4 Efficiency Analysis

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

We consider a concrete instantiation using endorsed e-cash [17], Camenisch-Shoup verifiable escrow [18], AES encryption [24], DSS signatures [37], and RSA-OAEP public key encryption for (non-verifiable) escrow [10]. Our protocol has only negligible overhead over just doing an unfair exchange. Sending the ciphertexts in step 2 and 3 just corresponds to sending the files in any

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

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

(even unfair) exchange.[20] Then, the keys sent in step 5 and 6 are extremely short messages (16-32 bytes each for a 128-256 bit AES key). For a fair exchange, step 4 is still very cheap since the only primitives used are an ordinary (non-verifiable) escrow (just a public key encryption), and a signature. (A DSS signature created using a 1024 bit key is about 40 bytes, while an RSA-OAEP encryption with a 1024 bit key is about 128 bytes.) The only problem is the first step, where the endorsed e-cash and the verifiable escrow are costly (see below).

Our protocol, in addition to guaranteeing fair barter efficiently, is optimized for multi-barter situations. One such situation is a file sharing scenario as in BitTorrent [21, 8]. The peers Alice and Bob are expected to have a long-term barter relationship. Hence, **step 1 needs to be carried out only once per peer, and remaining cheap steps 2-6 would be repeated for each file**. This greatly amortizes the costly step 1, when multiple or large files (or blocks) are exchanged, **even when the files to be exchanged are not pre-defined**. To give some numbers, our current Java implementation using endorsed e-cash [17] and Camenisch-Shoup verifiable escrow [18] takes about 2 seconds of computation for step 1 on an average computer ($1.6GHz$), but since the rest is almost the same as exchanging plain files, even for relatively small files this cost will be dominated by the file transfer times (*e.g.,* for $8.4MB$ average-sized *youtube.com* videos [20] upload will take a few minutes over a DSL connection [22]). Hence, for multiple file exchanges or large files composed of many blocks (*e.g.,* movies), our protocol's overhead is uniquely negligible (constant instead of linear in the number of exchanges). Our network overhead is similarly negligible (around $13KB$, almost all of which is the one-time cost of step 1).

As for the Arbiter, he checks a signature, sometimes decrypts a (verifiable) escrow, and performs the VerifyKey protocol of Belenkiy *et al.* [8]. The signature check and ordinary escrow decryption takes only milliseconds, the verifiable escrow decryption can take a few hundred milliseconds. The bottleneck is the data that the Arbiter needs to download for the VerifyKey protocol, which is about $22chunks \times 64KB = 1.4MB$ [8]. An important point to note is that the Arbiter's work is independent of the size of the file that is being exchanged. Some limitations and possible solutions, along with more evaluation numbers are discussed in Appendix D.

3.5 Security Analysis

In this section, we assume that we are given a one-way function, a universal one-way hash function, a chosen plaintext secure encryption scheme, a chosen plaintext secure verifiable escrow scheme, a chosen ciphertext secure escrow scheme, an unforgeable signature scheme, and an e-cash scheme which is unforgeable, anonymous and unlinkable. For precise definitions of security of these primitives, please see the references [27, 29, 37, 36, 33, 31, 24, 18, 17, 8]. In particular, we can use the instantiation in Section 3.4.

Theorem 1. *Our efficient barter protocol with timeouts as given in Section 3 is a secure optimistic fair exchange protocol according to definition 4 in Section 2.2.*

Proof. It is obvious that our protocol satisfies the *optimistic completeness* (and therefore the *completeness*) property. We prove the fairness of our protocol over the fairness game defined in Section 2.2. Remember that the win condition for the adversary would be that *the adversary obtains the honest party's file or e-coin (resp. both) whereas the honest party obtains nothing (resp. only the adversary's file).*

An honest party will always use **independent keys** for each ciphertext (s)he sends. Furthermore, endorsed e-cash [17] forces the users to use **independent** (***coin', end***) **pairs** in different

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

exchanges by using randomness contributed by both parties involved in the exchange. Our goal is that even if the adversary corrupts all other parties in the system (except the TTPs), he cannot obtain more than the union of what each of these individual corrupted parties was supposed to obtain from an honest trade with the honest user.

Security of the Resolution Protocols:

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

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

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

If a dishonest Bob contacts the Arbiter, he cannot provide an incorrect key to the Arbiter and make him accept. This can easily be shown by reduction to the security of universal one-way hash functions [36] (see Appendix B) or the VerifyKey protocol of Belenkiy *et al.* [8]. If dishonest Bob provided the Arbiter his correct key K_B and obtained honest Alice's correct key K_A, the only way he can be unfair against an honest Alice is to obtain her coin *end* in addition. But, Bob cannot obtain *end* because he either has to forge Alice's signature on another escrow e' of some junk key K'_A which does not decrypt correctly, or he could break our assumption on the hash functions by providing some ciphertext with description h_{c_A} which does not give a plaintext with description h_{f_A} when decrypted using Alice's key K_A in the escrow e. So, a dishonest Bob cannot obtain the endorsement of an honest Alice. Furthermore, he can obtain Alice's correct key K_A only if he deposits his correct key K_B.

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

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

Case 1: Honest Alice vs dishonest Bob:

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

Proof. A_C is given a challenge $coin'$ and her goal is to output an endorsement end. [21] She guesses an index i that Bob will succeed in being unfair, and replaces the $coin'^{(i)}$ by the given $coin'$. Since A_C does not know the $end^{(i)}$, she puts garbage into the verifiable escrow $v^{(i)}$, and sends it to Bob. She fakes the verifiability by using the simulator for the verifiable escrow [4, 18].[22] For all the other interactions, A_C acts exactly as an honest Alice would. Since A_C is honest, the verifiable escrow $v^{(i)}$ will never be decrypted by the Arbiter (shown in claim 1), and by the security of verifiable escrow, the adversary cannot obtain the endorsement by decrypting it, nor can the adversary distinguish it from a verifiable escrow of a valid endorsement (can be shown by a straightforward reduction to CPA-security of the verifiable escrow scheme, since the verifiable escrow v will never be decrypted if Alice is honest). At some point, Bob outputs an endorsement $end^{(j)}$ with non-negligible probability. The probability that $i = j$ is non-negligible by definition (the total number of barters n is a polynomial in SP as defined in Section 2.2). If the indices match ($i = j$), A_C outputs the $end^{(i)}$. Therefore, A_C breaks the endorsed e-cash [17] with non-negligible probability, by endorsing an unendorsed coin $coin'$ without the endorsement end. □

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

Proof. A_E generates her files using the setup phase. Then she guesses an index i that Bob will succeed in being unfair, and sends two file to the challenger of the encryption scheme. A_E is given back a challenge ciphertext c_A and her goal is to decide which file she sent was encrypted. She replaces the $c_A^{(i)}$ by c_A. For the rest of the interaction, A_E behaves as an honest Alice. A_E does not know the key $K_A^{(i)}$, but she can fake the escrow $e^{(i)}$ by encrypting junk in it. Due to the security of the escrow scheme, Bob cannot distinguish it from an honest escrow (can be shown by a straightforward reduction to CCA-security of the escrow scheme). At the end, Bob returns a plaintext $f_A^{(j)}$. If the guessed i was correct ($i = j$), then A_E returns $f_A^{(i)}$ and wins with the same probability as Bob does. Since A_E interacts with Bob only polynomially many times, the event $i = j$ has non-negligible probability, and since Bob has non-negligible probability of obtaining Alice's file, then A_E has non-negligible probability of breaking the encryption scheme used. □

Case 2: Honest Bob vs dishonest Alice:
The argument is symmetric to claim 3. The symmetric version of A_E can easily be reconstructed as B_E in this scenario, indistinguishable from an honest Bob. Hence, if Alice obtains Bob's file before step 5, B_E breaks Bob's encryption scheme. After step 5, Alice already has Bob's file, and can choose not to send her key in step 6. But, the security of BobResolve guarantees that Bob can obtain Alice's key or e-coin in exchange to his file from the Arbiter (shown in claim 1).

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

[21] A detailed proof will give A_C two oracles, one for $coin'$ creation, and one for end creation. Then, A_C will play a CCA-security like game with the e-cash scheme. The challenge $coin'$ will be the one used in the ith exchange, on which A_C cannot query the endorsement oracle.

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

3.6 Generalized Version

We have shown an instance of our protocol which uses hashes for verification, and endorsed e-cash for payment. In general, our protocols can employ any verification algorithm provided by the Tracker, instead of the hashes. Similarly, our protocols can easily make use of other payment methods (see [1] for a compilation) or signatures instead of e-cash, but then privacy of the participants will not be preserved (see Appendix C for a privacy analysis). The modification is straightforward, and involves just replacing the verifiable escrow of the e-coin with a verifiable escrow of any other form of payment.

4 Efficient Barter without Timeouts

We provide another protocol which makes no use of timeouts. In this case, both parties give e-coins to each other as a warranty. A similar setup applies here, where Bob is also required to have withdrawn an e-coin. Furthermore, Bob also generates a public-private key pair for his signature scheme. Details that were explained in our previous protocol will be omitted here.

1. ***a.*** Alice sends her unendorsed coin $coin'_A$, along with the verifiable escrow $v_A = VE_{Arb}(end_A; pk_A)$ of the endorsement to Bob.
   b. Bob sends his unendorsed coin $coin'_B$, along with his verifiable escrow $v_B = VE_{Arb}(end_B; pk_B)$ of his endorsement to Alice.

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

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

4. ***a.*** Alice sends her key K_A to Bob.
   b. Bob sends his key K_B to Alice.

The escrows in step 3 are a bit different than the previous protocol. First, there is no *time* value attached, since no timeouts are used. Furthermore, both escrows need to contain a value $g(r)$ where g is a one-way function, and only Alice knows r. This is achieved by requiring Alice to pick a random r in step 3.a, and then put $g(r)$ in the label of the escrow. After receiving Alice's escrow e_A, Bob also incorporates $g(r)$ into the label of his escrow e_B.[23]

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

The logic behind getting rid of the timeouts is similar to the idea in ASW [4]. If Alice wants to abort the protocol (because something was wrong with the message she received in step 3.b, or she

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

did not receive any response, she can do so by contacting the Arbiter using the *AliceAbort* protocol below. She no longer needs to wait until after the *timeout*. After receiving (or not receiving) Alice's message at step 3.a, Bob can simply abort locally if anything is wrong.

4.1 AliceAbort

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

4.2 Analysis

Theorem 2. *Our efficient barter protocol without timeouts in Section 4 is a secure optimistic fair exchange protocol due to the definition 4 in Section 2.2.*

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

Regarding efficiency, again, **step 1 has to be completed only once per peer, and then multiple files can be exchanged by carrying out steps 2-4** as long as both parties are honest, amortizing the cost of the coin and verifiable escrow exchange in step 1. The Arbiter's cost will be doubled though, due to the need to perform two costly resolutions (AliceResolve is as costly as BobResolve now). It is easy to see the generalization of the protocol in Section 3.6 also applies here.

5 Conclusion

There already are many scenarios where peers trade content [21, 30]. These systems unfortunately rely on the honesty of the peers for providing fairness, partly because of the high cost incurred by the previous fair exchange protocols [2, 3, 4, 5, 8, 17, 35]. By limiting the use of the costly primitives (verifiable escrow and e-cash) once (or twice) per peer, our protocols achieve negligible efficiency loss, and hence enable the use of fair exchange in those systems. Besides, most of the existing systems already rely on similar trusted parties [2, 3, 4, 5, 8, 16, 17, 19, 21, 30, 35, 38]. Therefore, by using our protocols, such bartering systems will experience almost no performance loss, while the benefit of providing fairness guarantees will be very noticeable indeed (*e.g.,* see [8] for how the use of fair exchange can solve the free-riding problem of BitTorrent). As a guideline, we suggest that systems which expect long-term barter relationships and are not willing to use timeouts use our protocol *without timeouts*, but systems that will conduct mainly short-term barters and can tolerate timeouts use our protocol *with timeouts*. Already, the Brownie project [14] is adopting our protocols in their BitTorrent deployment.

References

[1] N. Asokan, PA Janson, M. Steiner, and M. Waidner. The state of the art in electronic payment systems *IEEE Computer*, 30:28–35, 1997.

[2] N. Asokan, M. Schunter, and M. Waidner. Optimistic Protocols for Fair Exchange. *CCS*, 1997.

[3] N. Asokan, V. Shoup, and M. Waidner. Asynchronous protocols for optimistic fair exchange. *IEEE Security and Privacy*, 1998.

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

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

[6] G. Avoine, and S. Vaudenay. Optimistic Fair Exchange Based on Publicly Verifiable Secret Sharing. *ACISP*, 2004.

[7] M. Backes, A. Datta, A. Derek, JC. Mitchell, and M. Turuani. Compositional analysis of contract-signing protocols. *Theoretical Computer Science*, 367(1-2):33-56, 2006.

[8] M. Belenkiy, M. Chase, C.C. Erway, J. Jannotti, A. Küpçü, A. Lysyanskaya, and E. Rachlin. Making P2P Accountable without Losing Privacy. *WPES*, 2007.

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

[10] M. Bellare, and P. Rogaway. Optimal Asymmetric Encryption. *EUROCRYPT*, 1994.

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

[12] G.R. Blakley. Safeguarding cryptographic keys. *National Computer Conference*, 1979.

[13] D. Boneh, and M. Naor. Timed commitments. *CRYPTO*, 2000.

[14] Brownie Project. *http://cs.brown.edu/research/brownie.*

[15] J. Camenisch, and I. Damgård. Verifiable Encryption, Group Encryption, and Their Applications to Group Signatures and Signature Sharing Schemes. *Asiacrypt*, 2000.

[16] J. Camenisch, S. Hohenberger, and A. Lysyanskaya. Compact e-cash. *Eurocrypt*, 2005.

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

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

[19] D. Chaum. Bling signatures for untraceable payments. *CRYPTO*, 1982.

[20] X. Cheng, C. Dale, and J. Liu. Understanding the Characteristics of Internet Short Video Sharing: YouTube as a Case Study. *eprint arXiv*, 0707.3670, 2007.

[21] B. Cohen. Incentives build robustness in bittorrent. *IPTPS*, 2003.

[22] L. Cohen. Testimony of Larry Cohen, President of Communications Workers of America. May, 2007.

[23] R. Cramer, and V. Shoup. A Practical Public Key Cryptosystem Provably Secure Against Adaptive Chosen Ciphertext Attack. *CRYPTO*, 1998.

[24] J. Daemen, and V. Rijmen. The Design of Rijndael: AES–the Advanced Encryption Standard. *Springer books*, 2002.

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

[26] D. Dolev, C. Dwork, and M. Naor. Nonmalleable cryptography. *SIAM Journal on Computing*, 2000.

[27] E. Fujisaki, T. Okamoto, D. Pointcheval, and J. Stern. RSA-OAEP Is Secure under the RSA Assumption. *Journal of Cryptology*, 17(2):81–104, 2004.

[28] JA. Garay, M. Jakobsson, and P. MacKenzie. Abuse-free optimistic contract signing. *CRYPTO*, 1999.

[29] S. Goldwasser, S. Micali, and R. Rivest. A Digital Signature Scheme Secure Against Adaptive Chosen Message Attack. *SIAM Journal on Computing*, 1988.

[30] A. Iosup, P. Garbacki, J. Pouwelse, and D.H.J. Epema. Correlating Topology and Path Characteristics of Overlay Networks and the Internet. *GP2PC*, 2006.

[31] J. Katz, and Y. Lindell. Introduction to Modern Cryptography. *Chapman and Hall/CRC Press*, 2007.

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

[33] R. Merkle. A digital signature based on a conventional encryption function. *CRYPTO*, 1987.

[34] S. Micali. Simultaneous Electronic Transactions. *U.S. Patent*, No. 5,666,420, 1997.

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

[36] M. Naor, and M. Yung. Universal one-way hash functions and their cryptographic applications. *STOC*, 1989.

[37] NIST. Digital Signature Standard (DSS). *FIPS*, PUB 186-2, 2000.

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

[39] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. *EUROCRYPT*, 1999.

[40] A. Shamir. How to Share a Secret. *ACM Communications*, 1979.

[41] V. Shmatikov, and JC. Mitchell. Finite-state analysis of two contract signing protocols. *Theoretical Computer Science*, 283(2):419–450, 2002.

[42] V. Shoup, and R. Gennaro. Securing threshold cryptosystems against chosen ciphertext attack. *EUROCRYPT*, 1998.

A Subprotocols

We use two subprotocols from Belenkiy *et al.* [8] that make the interaction with the Arbiter efficient. One protocol is used to prove that a key is not correct, while the other is used to prove that the key is in fact correct. For efficiency, Merkle hashes [33] are used in these subprotocols (see Belenkiy *et al.* [8] for more information on the protocols and the use of Merkle hashes).

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

Proving a key is correct: Using a challenge-response protocol (Belenkiy *et al.* **VerifyKey** protocol), one can prove that a key is correct. As when some party wants to prove that a key is not correct, the Arbiter asks for proofs of the key decrypting correctly on random chunks. If Bob can reply correctly to all chunks providing valid proofs for Merkle hashes, the the Arbiter accepts Bob's key. If Bob corrupts $1/m$ fraction of the file, and the Arbiter verifies k parts, then the Arbiter will catch Bob with a probability of at least $1 - (1 - \frac{1}{m})^k$ [8].

B Universal One-Way Hash Functions (UOWHF)

Let H_k be a family of hash functions, where k is the security parameter. We assume that the following experiment has negligible probability of success for any polynomial-time adversary A, for sufficiently large k: We have a file f and a hash function $hash \leftarrow H_k$ uniformly chosen from the family. Given that file f and the hash function's description $hash$ (which effectively also means giving $hash(f)$) as input, A returns a c, K pair, where $hash(Dec_K(c)) = hash(f)$ but $Dec_K(c) \neq f$. Remember that A cannot control the file's hash, due to the trusted Tracker, hence he needs to find a targeted collision. The assumption that he cannot is at the heart of our resolution protocols.

This requirement is equivalent to the security of Universal One Way Hash Functions (UOWHF) [36]. We first reduce our assumption to the UOWHF assumption. Specifically, let A be a polynomial-time adversary succeeding in the above attack with non-negligible probability. We can construct an adversary B which finds a collision in our UOWHF as follows: When B is given $(f, hash)$, he runs A on $(f, hash)$ to obtain (c, K). B then checks if $Dec_K(c) = f$, in which case it fails. Otherwise, if $Dec_K(c) \neq f$ but $hash(Dec_K(c)) = hash(f)$, then B outputs $Dec_K(c)$ as the collision. As easily seen, B has the same success probability as A, and has polynomial runtime complexity.

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

C Privacy Analysis

None of the exchanged material contains information to identify Alice or Bob (not even Alice's signature, since her public key is generated just for the exchange; it is a temporary key, not permanent). Moreover, even an adversary performing multiple exchanges with the same honest party cannot link those exchanges together using the protocol messages since the honest party uses fresh keys every time and endorsed e-cash is unlinkable (IP address linking or similar means might be possible, but our protocol does not create any additional means of identification and linking). Furthermore, the Arbiter does not necessarily know who he is talking to, apart from the fact that the resolution is on a particular exchange (possibly identified by a random exchange ID). The Arbiter may be able to find out whether he is talking to Alice or Bob, but not who Alice or Bob is. Anonymous communication techniques such as onion routing [25] can be used when necessary. Lastly, e-cash [17] is anonymous, and thus even when Bob deposits the e-coin, no one can know it was Alice's e-coin (unless she double-spends).

D Limitations and Future Work

One limitation of our work is the need for exchanging parties to trust the Arbiter. Alice trusts the Arbiter not to give away both her e-coin and the key to her file. Even though giving away the key only makes the exchange unfair, giving away the coin may result in even an honest Alice becoming a *double-spender*.[24] One possible way to reduce this need for the trust would be using several arbiters, who do not necessarily know each other. Alice and Bob can mutually agree on a specific arbiter, *the Arbiter*, before the protocol begins. Since, there is no registration with the Arbiter in our protocol, any arbiter can accomplish the job.

Fortunately, if a proof of dishonesty is requested, neither the Arbiter, nor Bob, nor anyone else can frame an honest Alice.[25] The Arbiter may be asked to prove Alice's guilt by presenting a verifiable escrow, a non-verifiable escrow and a signature on it, along with the proofs that Bob's key decrypts correctly yet Alice's key in the (non-verifiable) escrow does not. Due to the security of these primitives, no one can frame an honest Alice. Of course, this requires the Arbiter to store all past resolutions, and Alice's privacy has already been invaded by the double-spending detection. In order to prevent a malicious Alice from framing the Arbiter by intentionally double-spending, we can require either Alice's or the Arbiter's signature when a coin is being deposited. We leave the issue of efficiently reducing the need to trust the Arbiter or verifying the Arbiter's behavior without violating Alice's privacy as a future work.

As for the bottleneck that can be caused by a central Arbiter, Avoine *et al.* [6] show how to employ secret sharing techniques [40, 12] to distribute the shares of the coin among arbiters. This will decrease the amount of job each arbiter needs to perform, yet it will reduce the efficiency of our resolution protocols. In another work, Belenkiy *et al.* [9] show how to outsource computation, which can be used as a means to distribute the work of our trusted parties. The Brownie project [14] is analyzing this distributed arbiter strategy in their BitTorrent deployment.

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

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

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

Without considering such DDoS attacks, let us provide some numbers for evaluation. To have an idea, consider a p2p system of $1,700,000$ users, exchanging $2.8GB$ files on the average [30]. Exchanging two such files means exchanging $5.6GB$ of data. If 1% of all users are malicious, this can correspond to $17,000$ exchanges requiring an arbiter at a given time (where one user is honest and the other is malicious. If both of them are malicious, this number reduces to half). We said, in case of a dispute, a peer should upload $1.4MB$ of data to the Arbiter. Assume that the same upload speed is used when trading files and contacting the Arbiter. If we assume the worst case scenario where the Arbiter can handle only one user at a time and every user is active at all times, this requires having 5 arbiters; with 10% malicious user ratio, we need 43 arbiters. Under the very realistic assumption that an arbiter can handle 25 users at a time (*e.g.,* assuming 25 times as fast download speed of the Arbiter as upload speed of the users [22]), we will need at most 2 arbiters in this system. When we use our protocol without timeouts, these numbers will double.

Lastly, our fairness definition states that a file and a payment can be fairly traded, as in previous works [4, 8, 17, 32]. The economics of this system, deciding on how much a file is worth fairly, is outside the scope of this paper. The participants can somehow agree on the price before our protocol begins (variable pricing), or alternatively a system can set the price that will apply to all participants (fixed pricing). In Belenkiy *et al.* [8], the authors assume each block in the BitTorrent system are worth one e-coin. We leave this pricing issue as an interesting application-dependent open problem.