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 fair exchange cannot be done without a trusted third party (called the *Arbiter*). 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 mechanism for bartering digital files, where receiving a payment in return to a file (buying) is also considered fair. The exchange is optimistic, removing the need for the Arbiter's involvement unless a dispute occurs. While the previous solutions employ costly cryptographic primitives for every file or block exchanged, our protocol employs them only once per peer, therefore achieving $O(n)$ efficiency improvement when n blocks are exchanged between two peers. The rest of our protocol uses very efficient cryptography, making it perfectly suitable for a p2p file sharing system where *tens* of peers exchange *thousands* of blocks and they do not know beforehand which ones they will end up exchanging. Therefore, our system yields to one-two orders of magnitude improvement in terms of both computation and communication (40 seconds vs. 42 minutes, 1.6MB vs. 200MB). Thus, for the first time, a provably secure (and privacy respecting when payments are made using e-cash) fair exchange protocol is being used in real bartering applications (e.g., BitTorrent) [14] without sacrificing performance.

Keywords: optimistic fair exchange, barter, peer-to-peer file sharing, BitTorrent.

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 [39] for more details and examples).

In this paper, we consider a general file exchange (bartering) scenario, inspired by the BitTorrent [22] peer-to-peer file sharing protocol. Alice has several files (BitTorrent blocks) of interest to Bob, and Bob has several files (blocks) of interest to Alice. They do not know ahead of time how many or which blocks they will end up exchanging. They want to perform a fair exchange: Alice should get Bob's file (block) if and only if Bob gets Alice's file (block). In a signature fair exchange [4, 3, 2], there is a verification mechanism (i.e., the public key) that enables the sender to verifiably encrypt the signature so that the receiver can check that the encrypted signature verifies. No such efficient verifiable encryption method is currently known for exchanging files. Therefore, a compensation is required after the fact if one of the parties cheat. In our scenario, we are assuming

that Alice/Bob will be equally happy to get a payment in return to her/his file. Thus, exchanging a file with a payment (buying) is also considered fair, as in some previous works [4, 8, 18, 36, 35].

One of the hardest points in creating a usable optimistic fair exchange protocol suitable for p2p file sharing applications is that the peers to contact and the content to exchange are not pre-defined. BitTorrent clients keep connecting to different peers to obtain different blocks. Fault-tolerance issues, connectivity problems, and availability of data blocks are all factors affecting from whom which block should be obtained. Our protocol uniquely addresses these issues by removing the need to know what content to exchange with whom beforehand.

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

We provide two versions of the protocol: In the first one (the one described briefly above) only one party provides a verifiable escrow. This version requires the use of timeouts for dispute resolution purposes. We provide another version that needs both parties to provide verifiable escrows but requires no timeouts. Both versions are very efficient since they use only *one* (resp. *two*) expensive primitives (verifiable escrow and payment) regardless of the number of files exchanged. We stress the fact that our timeouts can be very large (e.g., one day or week) to allow for unexpected situations in which the participants act honestly (e.g., network failure), and thus require very loose synchronization (e.g., one hour difference), and users can freely participate in other exchanges without waiting for the timeout.

Previous Work: It is well-known that a fair exchange protocol is impossible without a trusted third party (TTP) [42] (called the *Arbiter*) that ensures that Alice cannot take advantage of Bob, and vice versa. Without loss of generality, Alice will have to send the last message of the protocol, and we want to protect Bob in case she chooses not to do so. Without an arbiter, gradual release type of protocols where parties send pieces to each other in rounds can provide only weaker forms of fairness, and are much less efficient [11, 13].

Luckily, the impossibility result [42] 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 [38], Asokan, Schunter and 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, 30] were analyzed in [45] (see also [7]).

Asokan, Shoup and Waidner (ASW) [4] gave the first provably secure and completely fair optimistic exchange protocol for exchanging digital signatures. 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. They also assume there is an additional TTP (which we call the *Tracker*) that provides a means

of verification that the file actually contains the right content (e.g., using hashes). Such entities certifying hashes already exist in current BitTorrent systems [22].

Belenkiy et al. [8] used e-cash (introduced by Chaum [20]), in particular, endorsed e-cash [18] in their constructions. The reason is that other forms of payments (signatures or electronic checks used in [4, 36]) do not provide any privacy. In our protocols, any form of payment can be employed, but we will also use endorsed e-cash in our sample instantiation since it is efficient and anonymous. See Section 3.5 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 the same pair of users, even when peers do not know beforehand which blocks they will end up exchanging.* Using the best previous work (Belenkiy et al. barter protocol [8]), n pairs of blocks can be exchanged using n transactions, each of which requires a costly step involving expensive cryptographic primitives (a verifiable escrow and an e-coin). Our contribution is a very efficient fair exchange protocol using which this can be done with only *one* (or *two* if we do not want to employ timeouts) step in total that involves the same expensive primitives (verifiable escrow and payment). This is a property that is unique to our protocol: Instead of employing the costly primitives for every file or block that is exchanged, we employ them once per peer, even when peers do not know beforehand which blocks they will end up exchanging. Then, exchanging multiple files/blocks between peers involves only very efficient cryptography (i.e., symmetric- and public-key encryption, and digital signatures). In a real setting where BitTorrent peers exchange thousands of blocks with only tens of peers, there is *one or two orders of magnitude improvement in terms of both computation and communication* (40 seconds vs. 42 minutes computational overhead and 1.6MB vs. 200MB communication overhead for a 2.8GB file —for detailed numbers, see Section 3.2). This means that, with no (i.e., neglectable) efficiency loss, our fair exchange protocol can be used to exchange files instead of the unfair protocol currently used by BitTorrent or similar file sharing protocols.

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

We take the idea of using verifiable escrow from ASW [4], and the subprotocols of Belenkiy et al. [8] that increase the efficiency of the Arbiter (see Appendix B). 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 [18] as the payment mechanism. Note that other (non-anonymous) forms of payments (e.g., electronic checks [21]) will be more efficient.

Our additional contribution is definitional. We give a general definition of fair exchange of digital content (not just digital signatures) provided that it can be verified using some verification algorithm (defined in Section 2.2). Furthermore, our fairness definition covers polynomially many exchanges between an honest party and an adversary controlling polynomially-many other participants (see [27] for an example fair exchange protocol that is fair for a single exchange but stops being fair in a multi-user setting). We then prove our protocol's security based on this definition. We sum up the most important properties of our protocols below.

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

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

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

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

Already, the Brownie Project [14] is using our protocols in their BitTorrent deployment. We discuss the efficiency of our protocols and our initial implementation results in Section 3.2. 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, 19, 15] means that the recipient can verify that the contents of the ciphertext satisfy some relation (therefore stating that the ciphertext contains the expected content). A contract (a.k.a. label, condition, or tag) attached to such a ciphertext defines the conditions under which the TTP should decrypt and give away the encrypted secret [46]. The label is public and it is integrated with the ciphertext in a such way that it cannot be modified. We will use $E_{Arb}(a; b)$ to denote an escrow of the secret a under the Arbiter's public key, with the contract b. Similarly, $VE_{Arb}(a; b)$ will denote a verifiable escrow.

Any payment protocol that can efficiently be verifiably escrowed and is secure can be used in our protocols. Furthermore, if privacy is desired, the payments should be anonymous as in e-cash [20]. We provide an instantiation using endorsed e-cash [18] (which is an extension of compact e-cash [17]), 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 [19] matches the given unendorsed coin $coin'$ (without learning the endorsement end). Furthermore, given only the unendorsed part $coin'$, no other party (except the owner) can come up with a valid endorsement end. Endorsed e-cash moreover has the ability to catch double-spenders. Hence, if one uses two different $coin', end$ pairs trying to spend the same coin twice, (s)he will be caught (and, since her identity is revealed, can be punished). Note that if a party tries to deposit the same coin twice (using the same $coin', end$ pair), the operation can easily be denied by checking against

a list of past transactions. Lastly, only matching $coin', end$ pairs can be linked, unendorsed coins and endorsements prepared for different exchanges remain unlinkable.

Wherever used, K_P will denote a symmetric key of a party P, generated through an encryption scheme's key generation algorithm. We let $c = Enc_K(f)$ denote that the ciphertext c is an encryption of the plaintext f under the symmetric key K. Similarly, $f = Dec_K(c)$ will denote that the plaintext f is the decryption of the ciphertext c under the symmetric key K. Our protocol can make use of any secure symmetric encryption scheme (see the book by Katz and Lindell [33] for definitions and constructions).

Let pk_P and sk_P denote public and secret keys for a party P. Then $sign_{sk}(x)$ will denote a signature on x under the secret key sk which can be verified using the corresponding public key pk. Our protocol can make use of any secure public-key encryption scheme [24, 28] and any secure signature scheme [31].

Furthermore, let H_k be a family of (universal one-way) hash functions [40], where k is the security parameter, and let *hash* be a hash function uniformly choosen from the family H_k of hash functions. Then, $h_x = hash(x)$ will denote that h_x is the hash of x under the hash function *hash*. We now introduce a definition we frequently use in 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 $hash(f') = h_f$.*

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 polynomially-many exchanges between an honest user and an adversary controlling polynomially-many other users. Furthermore, we separate and clearly define the roles of all trusted parties. While providing models and definitions for a general framework of (optimistic) fair exchange applicable to a broad range of protocols, we will also show its extensions to our case.

MODEL: The model is adapted from the ASW definition [4], with clarifications and generalizations. There are three players; *Alice* and *Bob* exchanging two digital items, and the *Arbiter*[1] for conflict resolution. All players are assumed to be polynomial time interactive Turing machines. We make no assumption about the underlying network capability.[2] Any message that does not confirm with the protocol specification will be discarded by the honest parties. Any input which does not verify according to the protocol will be resolved as stated by the protocol or the protocol will be aborted if no resolution is applicable. It is important that the Arbiter resolves conflicts on the same exchange *atomically*.[3] Thus, it will only interact with either Alice or Bob at any given time

[1] One of the TTPs in ASW.

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

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

instance, until that interaction ends as specified by the protocol.[4] Sensitive communication (e.g., exchange of decryption keys for files or endorsement of an e-coin) will be carried out over a secure (and possibly authenticated) channel (e.g., SSL can be used to connect to the Arbiter, a secure key exchange with no public key infrastructure can be used for the communication between Alice and Bob).

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

In our model, we have two additional players, namely the *Tracker* (also in [4, 8, 22])[6] providing verification algorithms, and the *Bank* dealing with monetary parts of the system.

Setup Phase: Before the fair exchange protocol is run, we assume there is a setup phase. In this one-time pre-exchange phase, the Arbiter generates his public-private key pair (for the (verifiable) escrow schemes) and publishes his public key(s) so that both Alice and Bob obtain it. Optionally, the Arbiter may learn public keys of Alice and Bob in the setup phase, but our focus is on the case where the Arbiter does not need to know anything (and learns almost nothing, see Appendix 3.4) about Alice or Bob. *The adversary cannot interfere with the setup phase.*[7] In the setup phase, the Bank and the Tracker also generate their public-private key pairs and publish their public keys.

Definition 2. *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).[8] In general, we can consider these files as some strings in $\{0,1\}^*$, therefore consider fair exchange of anything that is verifiable. Without loss of generality, the Tracker gives Alice a *verification algorithm* $V_{f_B^{(i)}}$ for each file $f_B^{(i)}$, and Bob a verification algorithm $V_{f_A^{(i)}}$ for each file $f_A^{(i)}$ before the exchange takes place.[9]

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

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

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

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

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

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

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

the Tracker in that any input accepted by that verification algorithm will be the content they want. In other words, they are going to be happy with any content that verifies under that verification algorithm. In particular, the content generation process is trusted. The adversary cannot generate "junk" files and ask the Tracker to create verification algorithms for them. BitTorrent forum sites and ratings provide a level of defense against this in practice.

Definition 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)*

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

The definition above models the case in which the files to be exchanged cannot be found by the adversary by some other means[10] (and hence exchanging files makes sense for the adversary), even with the help of associated verification algorithms and public information[11].

To provide evidence on the generality and applicability of our definition, we present several example verification algorithms for various tasks. For example, a file verification can be performed using hashes. So, each verification algorithm $V_{f_A^{(i)}}$ for Alice's file $f_A^{(i)}$ contains the definition of hash function used –*hash*–[12], and the hash value $h_{f_A^{(i)}} = hash(f_A^{(i)})$. The i^{th} verification algorithm computes the hash of the given input according to the description of the hash function, and accepts it if and only if the computed hash matches $h_{f_A^{(i)}}$ (see Appendix C for a security analysis). As another example, consider the ASW signature exchange protocol, in which each verification algorithm contains the signature scheme's description[12], the signature public key of Alice pk_A[12], and the message m_i to be signed. When it receives a signature as input, the i^{th} verification accepts the signature if and only if it is a valid signature on message m_i under the public key pk_A using the signature scheme. As yet another example, an e-coin verification algorithm can take a coin to verify, and use the Bank's public key while verifying the non-interactive proofs given. Such an algorithm is a part of the specification of every e-cash scheme (e.g., see [18, 17]). Verifiable encryption schemes (e.g., [19]) 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 $\mathsf{PPGen}(\mathsf{SP})$. The files and the verification algorithms are generated jointly by the Tracker and some trusted content generator (e.g., movie distributor) using the $\mathsf{Gen}(\mathsf{SP})$ procedure. In the context of BitTorrent, this means that we trust the content generator about the content, and the Tracker about the verification algorithms. In practice, BitTorrent forum sites and ratings on files provide this trust.

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

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

[12] possibly different for each verification algorithm

A "highly rated" BitTorrent user will be trusted about the content, or alternatively, comments on the forum sites will warn against bogus content. Besides, even the public information leaked from the generation procedure does not help the adversary. From now on, we assume the content and the verification algorithms used are secure and trusted.

Definition 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 (when Alice and Bob are both honest) and fairness (when either Alice or Bob is malicious).*

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

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

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

Optimistic Completeness for an optimistic fair exchange states that the interactive run of A and B by *honest parties* results in A getting $f_B^{(i)}$ and B getting $f_A^{(i)}$ for all $i : 1..n$ (the Arbiter's algorithm T is not involved, assuming an ideal network). The optimistic fair exchange protocol algorithms still get the public values although they may not be used. In our protocols, the Arbiter never gets involved in a transaction (does not even know such a transaction took place) unless there is a dispute. 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, even when one of them is malicious.[13] This definition is easy to satisfy using a (non-optimistic) fair exchange protocol since Alice and Bob can both hand their files to the Arbiter, and then the Arbiter can send Bob's files to Alice and Alice's files to Bob, if they pass respective verifications. Thus, below, we will define the more interesting case; fairness for an *optimistic* fair exchange. Even though we define fairness in a symmetric way, during the security analysis one may need to consider two cases independently since the protocol can be asymmetric: the case where Alice is honest but Bob is malicious, and the case where Bob is honest but Alice is malicious. It is important to note that the ASW definition of fairness applies only to a single exchange, whereas our definition covers polynomially-many exchanges between an honest party and other players all controlled by the adversary.

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

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 against all PPT adversaries $\mathcal{A}$, and all polynomially-long auxiliary inputs.

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

FAIRNESS GAME: There are three types of interaction in our fairness game. Type 1 interactions are between H and $\mathcal{A}$. Type 2 interactions are between H and T. Type 3 interactions are between $\mathcal{A}$ and T.[14] The adversary can arbitrarily interleave type $1, 2, 3$ interactions, but cannot prevent type 2 interactions from happening until the timeout if timeouts are used, or eventually otherwise. The game ends when the honest party H produces its final output (including aborts and resolutions) in all the started protocols. Without loss of generality, in the fairness game we assume both parties want to exchange different content in different exchanges ($\forall i \neq j \quad f_H^{(i)} \neq f_H^{(j)}$ and $f_{\mathcal{A}}^{(i)} \neq f_{\mathcal{A}}^{(j)}$ and $\forall i, j \quad f_H^{(i)} \neq f_{\mathcal{A}}^{(j)}$).[15]

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 $\forall i : 1..n \quad [\exists x \in X : V_{f_{\mathcal{A}}^{(i)}}(x) = accept \Rightarrow (\exists y \in Y : V_{f_H^{(i)}}(y) = accept \quad XOR \quad \exists y \in Y : V_{c_H^{(i)}}(y) = accept)] \wedge [\exists y \in Y : V_{f_H^{(i)}}(y) = accept \Rightarrow (\exists x \in X : V_{f_{\mathcal{A}}^{(i)}}(x) = accept \quad XOR \quad \exists x \in X : V_{c_{\mathcal{A}}^{(i)}}(x) = accept)]$ Our fairness condition states that either they both parties get each other's file, or one of them gets

[14] 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.

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

the other's file whereas the other gets his payment, or they both get nothing at each exchange. We believe that a broad range of optimistic fair exchange protocols can adapt the definition above using straightforward extensions whenever necessary.

Timely Resolution: Lastly, as pointed out by ASW [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, 39, 8, 18], when n files or blocks are exchanged, and almost as efficient as an unfair exchange, while still being provably fair.

3 Efficient Optimistic Barter Protocol

3.1 Barter with Timeouts

We will show a particular instantiation of our protocol, using endorsed e-cash [18] as the payment and hashes as the file verification algorithms, and then point out how to generalize it easily, in Section 3.5. Before the protocol begins, we assume Alice has withdrawn an e-coin from the Bank. Every time Alice and Bob wants to exchange two files (every time before step 2 of the protocol below), Alice generates her fresh key K_A and Bob generates his fresh key K_B for a symmetric encryption scheme. Alice and Bob both have their files (f_A, f_B), have the encrypted versions of their files $(c_A = Enc_{K_A}(f_A), c_B = Enc_{K_B}(f_B))$, have the hashes of their files and encryptions (Alice has $h_{f_A} = hash(f_A), h_{c_A} = hash(c_A)$, and Bob has $h_{f_B} = hash(f_B), h_{c_B} = hash(c_B)$). Besides, the Tracker provides them with the respective verification algorithms: Alice gets h_{f_B}, Bob gets h_{f_A}.[16] Everyone uses the same time zone (e.g., GMT), and the *timeout* is a globally known parameter[17]. 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.[18] Bob calculates $h_{c_A} = hash(c_A)$.[19]

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

[16] 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.

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

[18] 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.

[19] These will be Merkle hashes [37] for efficiency reasons, as discussed in Appendix B.

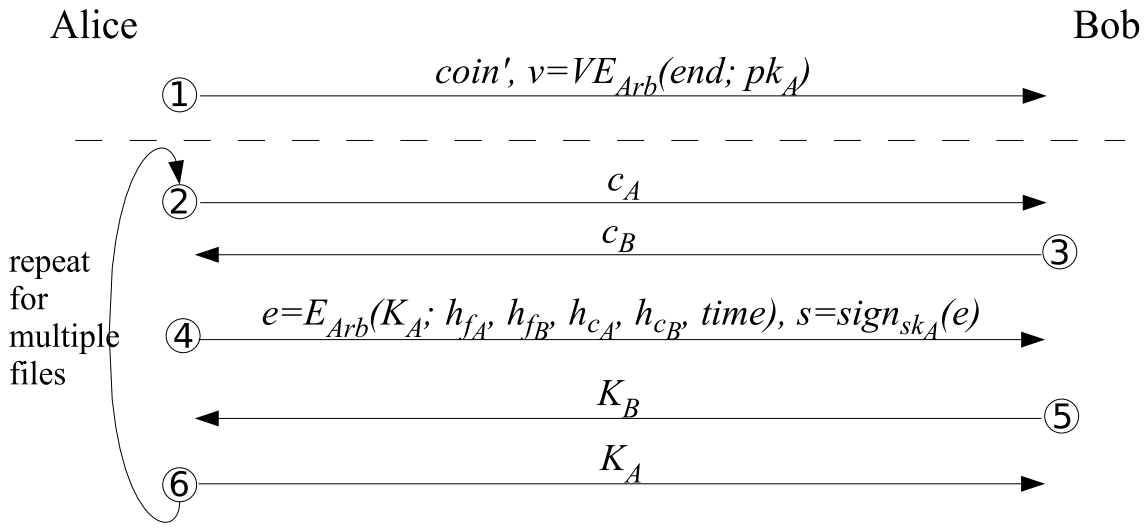

Figure 1: Our Barter Protocol with Timeouts

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.[20] 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.[21]

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

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

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

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

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 B (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 B and gets the endorsement *end* in the verifiable escrow v from the Arbiter.[22] 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 C).

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.[23] 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.

3.2 Efficiency Analysis

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

We consider a concrete instantiation of our protocol using endorsed e-cash [18], Camenisch-Shoup verifiable escrow [19], AES encryption [25], DSS signatures [41], and RSA-OAEP public key encryption for (non-verifiable) escrow [10]. Our protocol has only neglectable overhead over just doing an unfair exchange. Sending the ciphertexts in steps 2 and 3 just corresponds to sending the

[22] 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.

[23] 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 B 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.

files in any (even unfair) exchange.[24] The keys sent in steps 5 and 6 are extremely short messages (16 bytes each for 128-bit AES keys). For a fair exchange, step 4 is still very cheap since the only primitives used are an ordinary (non-verifiable) escrow (just a public key encryption), and a signature (A DSS signature created using a 1024-bit key is about 40 bytes, while an RSA-OAEP encryption with a 1024-bit key is about 128 bytes).

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

Our protocol, in addition to guaranteeing fair barter efficiently, is optimized for multi-barter situations. One such situation is a file sharing scenario as in BitTorrent [22, 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 block, whereas previous protocols required a costly step like step 1 to be performed for each block**. This greatly amortizes the costly step 1 in our protocol, when multiple blocks (or files) are exchanged, **even when the files/blocks to be exchanged are not pre-defined** (they need to be defined only before each execution of step 2).

To give some numbers, consider an average BitTorrent file of size $2.8GB$ made up of about $2,500$ blocks [32]. Using previous optimistic fair exchange protocols, this requires $2,500$ costly steps (one per block). Our C++ implementation using endorsed e-cash [18] and Camenisch-Shoup verifiable escrow [19] takes about 1 seconds of computation for step 1 (most of which is the verifiable escrow) on an average computer ($2GHz$). This corresponds to $2500 \times 1seconds =$ **42 minutes** of computation overhead. Considering a BitTorrent client that connects to about 40 peers, using our protocol, this overhead becomes just **40 seconds**, which is neglectable when exchanging such a big amount of data (this cost will be dominated by the file transfer times). Our network overhead is similarly neglectable (around $40KB$ per peer, almost all of which is the one-time cost of step 1, about half of it being endorsed e-cash). This corresponds to about $2500 \times 2 \times 40KB =$ **200 MB** total overhead using previous schemes, and only $40 \times 40KB$**1.6 MB** total overhead using our scheme (for a $2.8GB$ file).

As for the Arbiter, he checks a signature, sometimes decrypts a (verifiable) escrow, and performs the VerifyKey protocol of Belenkiy et al. [8] (see Appendix B). The signature check and ordinary escrow decryption takes only milliseconds, the verifiable escrow decryption, when necessary, can take a few hundred milliseconds. The bottleneck is the data that the Arbiter needs to download for the VerifyKey protocol, which is about $22chunks \times 16KB = 352KB$ [8]. An important point to note is that *the amount of data the Arbiter's needs to download is independent of the size of the file that is being exchanged.*[25]

Without considering distributed denial of service (DDoS) attacks, let us provide some numbers for evaluation. To have an idea, consider a p2p system of $1,700,000$ users, exchanging $2.8GB$ files on the average [32]. 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

[24] 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.

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

one user is honest and the other is malicious. If both of them are malicious, this number reduces to half of it). We said, in case of a dispute, a peer should upload $352KB$ of data to the Arbiter. Assume that the same upload speed is used when trading files and contacting the Arbiter. If we assume the worst case scenario where the Arbiter can handle only one user at a time and every user is active at all times, this requires having 2 arbiters; with 10% malicious user ratio, we need 11 arbiters. Under the very realistic assumption that an arbiter can handle 25 users at a time (e.g., assuming 25 times as fast download speed of the Arbiter as the upload speed of the users [23]), we will need 1 arbiter in this system (even with 10% malicious user ratio). Some more efficiency evaluation, limitations and possible solutions are discussed in Appendix D.

3.3 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 [29, 31, 41, 40, 37, 33, 25, 19, 18, 8]. In particular, we can use the instantiation in Section 3.2.

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 our fairness condition states that either both parties obtain the other party's file, or one party obtains the other party's file while the other party obtains the e-coin (effectively turning into a buy protocol), or no party obtains anything.

An honest party will always use independent keys for each ciphertext (s)he sends. Furthermore, endorsed e-cash [18] forces the users to use independent $(coin', end)$ pairs in different exchanges by using randomness contributed by both parties involved in the exchange. Our goal is that even if the adversary corrupts all other parties in the system (except the TTPs), he cannot obtain more than the union of what each of these individual corrupted parties was supposed to obtain from an honest trade with the honest user.

Security of the Resolution Protocols:

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

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

Proof. ***BobResolve***: When an honest Bob contacts the Arbiter, he provides the correct key K_B and obtain the decryption of the escrow e from the Arbiter. If this escrow contained the correct key K_A, then we are done. Otherwise, Bob can prove so (as in Appendix B) 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 [40] (see Appendix C) or the VerifyKey protocol of Belenkiy et al. [8] (Appendix B). If dishonest Bob provided the Arbiter his correct key K_B and obtained honest Alice's correct key K_A, the only way he can be unfair against an honest Alice is to obtain her coin *end* in addition. But, Bob cannot obtain *end* because he either has to forge Alice's signature on another escrow e' of some junk key K'_A which does not decrypt correctly, or he could break our assumption on the hash functions by providing some ciphertext with description h_{c_A} which does not give a plaintext with description h_{f_A} when decrypted using Alice's key K_A in the escrow e. So, a dishonest Bob cannot obtain the endorsement of an honest Alice. Furthermore, he can obtain Alice's correct key K_A only if he deposits his correct key K_B.

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

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

Case 1: Honest Alice vs dishonest Bob:

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

Proof. A_C is given a challenge *coin'* and her goal is to output an endorsement *end*. [26] 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, 19].[27] For all the other interactions, A_C acts exactly as an honest Alice would. Since A_C is honest, the verifiable escrow $v^{(i)}$ will never be decrypted by the Arbiter (shown in Claim 1), and by the security of verifiable escrow, the adversary cannot obtain the endorsement by decrypting it, nor can the adversary distinguish it from a verifiable escrow of a valid endorsement (can be shown by a straightforward reduction to CPA-security of the verifiable escrow scheme, since the verifiable escrow v will never be decrypted because Alice is honest). At some point, Bob outputs an endorsement $end^{(j)}$ with non-negligible probability. The probability that $i = j$ is non-negligible by definition (the total number of barters n is a polynomial in SP as defined in Section 2.2). If the indices match ($i = j$), A_C outputs the

[26] 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 i^{th} exchange, on which A_C cannot query the endorsement oracle.

[27] 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.

$end^{(i)}$. Therefore, A_C breaks the endorsed e-cash [18] with non-negligible probability, by endorsing an unendorsed coin $coin'$ without the endorsement end. □

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

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

3.4 Privacy Analysis

None of the exchanged material contains information to identify Alice or Bob (not even Alice's signature, since it is a temporary -just for the exchange-, not permanent). Moreover, even an adversary performing multiple exchanges with the same honest party cannot link those exchanges together using the protocol messages since the honest party uses fresh keys every time and endorsed e-cash is unlinkable (IP address linking or similar means might be possible, but our protocol does not create any additional means of identification and linking). Furthermore, the Arbiter does not necessarily know who he is talking to, apart from the fact that the resolution is on a particular exchange (possibly identified by a random exchange ID). The Arbiter may be able to find out whether he is talking to Alice or Bob, but not who Alice or Bob is. Anonymous communication techniques such as onion routing [26] can be used when necessary. Lastly, e-cash [18] 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).

3.5 Generalized Version

We have shown an instance of our protocol which uses hashes for verification, and endorsed e-cash for payment. In general, **our protocols can employ any secure verification algorithm** (see Definition 3) 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. 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 does not make use of timeouts. In this case, both parties give e-coins to each other as a warranty. A similar setup applies here, where Bob is also required to have withdrawn an e-coin. Furthermore, Bob also generates a public-private key pair for his signature scheme. Details that were explained in our previous protocol will be omitted here.

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

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

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

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

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.[28]

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.

[28] 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.

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 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.

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.[29] (Remember, Alice needed to come back after the timeout in our previous protocol.)

4.1 Analysis of Barter without Timeouts

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

Theorem 2. *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. □

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

Theorem 3. *Our efficient barter protocol without timeouts preserves the privacy of the honest participants even when Arbiter resolution is required.*

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

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

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. We believe, in many situations, our more efficient protocol with timeouts will be sufficiently useful. Yet, to provide options, we chose to present another efficient barter protocol that does not require the use of timeouts. Our protocol without timeouts requires *two* costly operations (step 1) instead of *one* in our protocol with timeouts. As in our protocol with timeouts, this cost is independent of the number of files exchanged, and becomes negligible when multiple or large files are exchanged. The cost of step 1 will be doubled for both parties, yet for the rest of the protocol the cost will stay the same. The Arbiter's cost will be doubled though, due to the need to perform two costly resolutions (AliceResolve is as costly as BobResolve now). Nevertheless, using similar numbers as in Section 3.2, if our arbiter can handle 25 users at a time, we still need only 1 arbiter.

5 Conclusion

There already are many scenarios where peers trade content [22, 32]. 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, 18, 39]. Our protocols uniquely limit the use of the costly primitives (verifiable escrow and e-cash) to once (or twice) per peer, as opposed to per file/block. We have shown in Section 3.2 that there are one or two orders of magnitude efficiency gains over previous protocols. Besides, most of the existing systems already rely on similar trusted parties [2, 3, 4, 5, 8, 17, 18, 20, 22, 32, 39, 42]. Therefore, for the first time, by using our protocols, such bartering systems will experience almost no performance loss, while the benefit of providing fairness guarantees will be very noticeable indeed (e.g., see [8] for how the use of fair exchange can solve the free-riding problem of BitTorrent). Already, the Brownie Project [14] is adopting our protocols in their BitTorrent deployment.

As a guideline, we suggest that systems which expect long-term barter relationships and are not willing to use timeouts use our protocol without timeouts, but systems that will conduct mainly short-term barters and can tolerate timeouts use our protocol with timeouts.

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, M. Kohlweiss, A. Lysyanskaya and M. Meyerovich. How to Win the Clonewars: Efficient Periodic N-times Anonymous Authentication. *CCS*, 2006.

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

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

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

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

[21] D. Chaum, B. den Boer, E. van Heyst, S. Mjolsnes, and A. Steenbeek. Efficient offline electronic checks. *EUROCRYPT*, 1990.

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

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

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

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

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

[27] Y. Dodis, P.J. Lee, and D.H. Yum. Optimistic Fair Exchange in a Multi-user Setting. *PKC*, 2007.

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

[29] 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.

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

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

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

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

[34] A. Küpçü, and A. Lysyanskaya. Framework for Analyzing Optimistic Fair Exchange Protocols with Distributed Arbiters. *Cryptology ePrint Archive, Report 2009/069*, 2009.

[35] Y. Lindell. Legally Enforceable Fairness in Secure Two-Party Computation. *CT-RSA*, 2008.

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

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

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

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

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

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

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

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

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

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

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

A ASW Fair Exchange in More Detail

We present the ASW protocol for fair signature exchange without timeouts, as a reference. The protocol in its basic sense (without conflict resolution details) is:

1. Alice sends Bob a non-verifiable escrow of her signature, with a label defining how Bob's signature should look like. Bob checks if the definition is the correct definition.
2. Bob sends Alice a *verifiable* escrow of his signature, with the label defining how Alice's signature should look like and also attaching the escrow he obtained in step 1. Alice verifies the verifiable escrow. She furthermore checks if the label is formed correctly. If anything goes wrong at this step or a *message timeout* occurs, she aborts the protocols and runs AliceAbort with the Arbiter.
3. Alice sends Bob her signature. Bob verifies this signature, and stops and runs BobResolve if it does not verify or a *message timeout* occurs.
4. Bob sends Alice his signature. If the signature does not verify, Alice runs AliceResolve.

The AliceAbort, BobResolve and AliceResolve protocols have a similar logic as ours. AliceAbort tells the Arbiter to consider that trade as aborted and not to honor any further resolution request on that particular trade. BobResolve gets Alice's signature by providing Bob's signature, and similarly, AliceResolve gets Bob's signature by providing Alice's signature. ASW provide a more complicated protocol for exchanging an electronic check for a digital file, building on top of their signature exchange protocol. The details of both protocols can be found in [4].

B 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 [37] 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, to prove the key is not correct, gives the Arbiter a part c_i of data which does not decrypt correctly. The Arbiter can check if the given part c_i matches the Merkle tree hash of the ciphertext, and $Dec_K(c_i)$ does not match the hash of the plaintext, using the proof provided by Carol.

Proving a key is correct: Using a challenge-response protocol (Belenkiy et al. VerifyKey protocol), one can prove that a key is correct. The Arbiter asks for proofs of the key decrypting correctly on random chunks. If Bob can reply correctly to all chunks providing valid proofs for Merkle hashes, the the Arbiter accepts Bob's key. If Bob corrupts $1/m$ fraction of the file, and the Arbiter verifies k random parts, then the Arbiter will catch Bob with a probability of at least $1 - (1 - \frac{1}{m})^k$ [8].

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

Algorithm B.1: VerifyKey from Belenkiy et al. [8]

Arbiter's Input: Two Merkle hashes h_f and h_c, key K

Bob's Input: Ciphertext $c = c_0..c_n$, key K

`Step 1: Arbiter's challenge`

The arbiter sends Bob a set of random indices I.

`Step 2: Bob's response`

Bob replies with $c_i, cproof_i, fproof_i$ for every $i \in I$, where $cproof_i$ proves that c_i is in the Merkle tree corresponding to h_c, and $fproof_i$ proves that $f_i = Dec_K(c_i)$ is in the Merkle tree corresponding to h_f.

`Step 3: Verification`

The arbiter *accepts* the key if Bob responds with valid $c_i, cproof_i, fproof_i$ for every $i \in I$, and *rejects* otherwise.

C 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 content and verification algorithm generation process, hence he needs to find a targeted collision.

This requirement is equivalent to the security of Universal One Way Hash Functions (UOWHF) [40]. We first reduce our assumption to the UOWHF assumption. Specifically, let A be a polynomial-time adversary succeeding in the above attack with non-negligible probability. We can construct an adversary B which finds a collision in our UOWHF as follows: When B is given $(f, hash)$, he runs A on $(f, hash)$ to obtain (c, K). B then checks if $Dec_K(c) = f$, in which case it fails. Otherwise, if $Dec_K(c) \neq f$ but $hash(Dec_K(c)) = hash(f)$, then B outputs $Dec_K(c)$ as the collision. As easily seen, B has the same success probability as A, and has polynomial runtime complexity.

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

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

D Limitations and Future Work

One limitation of our work is the need for the exchanging parties to trust the Arbiter. Alice trusts the Arbiter not to give away both her e-coin and the key to her file. Even though giving away the key only makes the exchange unfair, giving away the coin may result in even an honest Alice becoming a *double-spender*.[30] 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.[31] The Arbiter may be asked to prove Alice's guilt by presenting a verifiable escrow, a non-verifiable escrow and a signature on it, along with the proofs that Bob's key decrypts correctly yet Alice's key in the (non-verifiable) escrow does not. Due to the security of these primitives, no one can frame an honest Alice. Of course, this requires the Arbiter to store all past resolutions, and Alice's privacy has already been invaded by the double-spending detection. In order to prevent a malicious Alice from framing the Arbiter by intentionally double-spending, we can require either Alice's or the Arbiter's signature when a coin is being deposited. We leave the issue of efficiently reducing the need to trust the Arbiter or verifying the Arbiter's behavior without violating Alice's privacy as a future work.

As for the bottleneck that can be caused by the central Arbiter, Avoine et al. [6] show how to employ secret sharing techniques [44, 12] to distribute the shares of the secrets among arbiters. This will decrease the amount of job each arbiter needs to perform, yet it will reduce the efficiency of our resolution protocols. As argued in [34], the same techniques can be applied to our protocol with timeouts. 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 strategy to distribute the arbiter and the bank in their BitTorrent deployment.

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

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

In terms of the storage load associated with the trusted parties, the techniques from [16] can be applied. Using those techniques, the Bank can have a limited storage, as opposed to a groving storage. For example, if every e-coin is valid for a limited but long time (e.g., one month), then the bank needs to keep track of only the transactions that happened in the past period, instead of

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

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

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

Lastly, our fairness definition states that a file and a payment can be fairly traded, as in previous works [4, 8, 18, 36, 35]. 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.