A SMART CONTRACT REFEREED DATA RETRIEVAL PROTOCOL WITH A PROVABLY LOW COLLATERAL REQUIREMENT

A PREPRINT

James M. Shook
National Institute of Standards and Technology
james.shook@nist.gov

Scott Simon
National Security Agency
sbsimon@tycho.ncsc.mil

Peter Mell
National Institute of Standards and Technology
peter.mell@nist.gov

May 21, 2019

ABSTRACT

We present a protocol for a cryptoeconomic fair exchange of data previously owned by the purchaser for tokens that functions even when both parties are anonymous. This enables peer-to-peer data storage without identity verification. We use a smart contract on a decentralized ledger as a trusted third party. Actual data transfer can take place with any standard anonymous exchange channel. Due to the anonymity of the parties, the smart contract cannot punish either party's off-ledger reputation. Furthermore, the contract has limited power to arbitrate fault in off-ledger disputes. Thus, an important feature of our protocol is a collateral mechanism that collectively punishes both Alice and Bob if either of them abandons the protocol or cheats. However, we prove that parameters can be chosen such that the collateral can be made, subject to data size, arbitrarily low and still result in an expected financial loss if either Alice or Bob cheats. We are able to achieve this due to our non-standard use of error-correcting codes. In addition, the protocol allows those storing the data to exchange it without the client's participation.

***K*eywords** blockchain, collateral, decentralized ledger, decentralized storage, error-correcting codes, fair exchange, smart contract

1 Introduction

Sharing economies, in which participants can rent goods or services to each other, have been growing ever larger (e.g., ride sharing, the renting out unused rooms in homes, and peer-to-peer lending). With respect to data storage, decentralized data storing services are increasingly competing with centralized cloud data storage companies [6, 22, 40]. Aiding this trend is that distributed data storage reduces the risk inherent in placing all of one's data in a single place. However, such peer-to-peer networks present many new challenges. One unaddressed issue in such sharing economy based data storage networks is that it is difficult to build trust that the customer storing data will receive back the stored data upon payment; this is the focus of our work.

To address this problem, we present a protocol that allows a person (Bob) to rent out his extra storage to another person (Alice) even when both parties are anonymous. We make no claims concerning the anonymity of current blockchain technology, instead assuming that such a blockchain exists, and showing that our protocol works whether or not the identities of the parties are known. We assume Alice and Bob are rational people, do not trust any one person or entity, and want to minimize their loss of assets. Our protocol focuses on the delivery of the data to Alice and the payment of Bob, that is, data retrieval. Therefore, we assume there is some kind of storage market that matches a client to those offering data storage that charges per bit of data. The storage market should be able to verify that those storing the

data fulfill their contractual obligations. This includes verifying that the data is actually being stored in the correct number of distinct locations and is accessible for the required amount of time. There are many valid concerns about the storage market, such as whether Alice may have a holdup problem, see [18], where a storage host tries to change the agreed-upon price after the agreement and before delivery. We will assume that there are a sufficient number of participants in the market so that Alice will have other alternative hosts, and therefore be able to avoid such problems. There is also the issue of whether Alice might disappear without ever asking for her data again. Solutions to this problem already exist, such as requiring Alice to make regular payments, as in [40]. In general, we will not address the storage market or its problems.

Data retrieval involves a *fair exchange*: Alice exchanges tokens for data, where ideally neither would receive the desired item unless both do. As shown in [37], it is impossible to have a fair exchange without a trusted third party. Since Alice and Bob want to remain anonymous and do not trust any entity, we must rely on the use of a smart contract (see Section 2.3) hosted on a distributed ledger (see Section 2.1) as the trusted third party. By its nature, the smart contract has limited power to arbitrate activities that happen off the distributed ledger. Thus, our smart contract will collectively lock up collateral from both Alice and Bob until the contract is satisfied. However, we keep the financial exposure for Alice and Bob low since the design of our protocol leads to an expected financial loss for a cheating party even with a collateral that is a small fraction of the cost of the data.

Next, we present two naive storage retrieval protocols and discuss some of their deficiencies. These examples will lay a foundation for our more advanced protocol. We will not give details on how Alice and Bob are matched in these examples. We also assume that Alice has encrypted her data so that Bob cannot read it.

Example 1. Once Bob agrees to store Alice's data, she will generate a smart contract for the deal. After the smart contract is initialized, Bob must post his public key. Alice will encrypt her data with Bob's public key and hash the encrypted data. She will then store this hash on the contract with any padding information needed for the encryption protocol. At this point, she will send Bob her data that is not encrypted with Bob's public key. Upon receiving the data, Bob will encrypt it with his public key and then hash it. If the hash matches what is on the contract, then Bob may agree to store the data.

At some point, Alice will request her data from Bob. Bob will encrypt the data with his public key and then send it to Alice. Once Alice receives her data, she will check that the hash of it matches the one she posted on the contract. If the hash matches, she will send Bob's payment to the contract. The contract will wait for Bob to post his private key. Once the key is posted, the contract will verify that Bob's posted private key matches his posted public key. If the keys are verified, the contract will pay Bob.

In this scenario, Alice risks Bob disappearing with her data or extorting her if she does not store her data with other hosts. To mitigate this, Alice would have to repeat the protocol with multiple hosts, creating a lot of overhead for her. Furthermore, Bob would not be able to sell the data to a third party who could store it for Alice, since that party may suspect that Bob has already sold his private key to Alice. Therefore, we expand on this simple protocol to allow Alice to easily store her data with multiple people and allow Bob to sell the file storage to other hosts.

Example 2. Our revised example protocol also begins with Alice generating a smart contract. However, Alice will not encrypt her data with Bob's key. Instead, she will break her data into t pieces in an agreed upon way. She will then hash the list of the hashes of each piece. We will call this hash the root hash. Along with the value of t, Alice will store the root hash on the contract. This allows Alice to verify pieces individually at a later time without being given all of the data, and without storing the entire hash list, since it can be re-generated later by the host and verified by Alice using the root hash. Alice will then send the data to Bob. At this point Bob will also break up the file into t pieces so that he can verify the root hash on the contract. Again, if everything checks out, Bob may agree to store the data.

Later, Alice can request her data from Bob by posting the full price of the data onto the contract as collateral. Bob will respond by posting the same collateral. Then Bob will break up the data exactly as they did before and store the hashes of the pieces in a list. Bob will send the hash list and the first piece to Alice. Alice will verify that the hash of the piece is in the list and then verify that the hash of the list will match the root hash on the contract. If everything checks out, she can pay Bob through the contract for another piece. This signals Bob to send another piece that Alice can then verify is in the hash list. The protocol will continue in this fashion until Alice has received all of the pieces. Once Alice pays for the last piece, the contract will release the collateral back to Bob and Alice.

From these examples we can see that it is in Bob's interest to have some kind of mechanism to deter Alice from running away with the data without paying. The first Example protocol did this with encryption and the second did this with a collateral mechanism. Note that Filecoin [22] uses a similar protocol to Example 2 between their retrievers and clients

that does not use hashes or collateral. However, they assume there exists at least one honest retrieval miner, where we only assume the parties are rational. In both Examples, with the use of hashes on the contract Alice is able to verify that the data sent by Bob was hers. Since Example 2 does not have an encryption component, Alice can use the same hash list to store her data with many people. Moreover, those storing it can freely trade the data. Unlike the first protocol, the major risk with the second protocol and the retrieval method given in [21] is the large collateral. If something goes wrong, then both parties risk losing more than the cost of the data.

1.1 Our Contribution

We designed a new data retrieval protocol with a collateral mechanism calibrated to disincentivize cheating and Distributed Denial of Service (DDoS) attacks at every stage, but that is at a fraction of the cost of storing the data. Unlike previous work where the collateral was near or even twice the cost of the data being stored, we prove the collateral can be made significantly lower than the cost of the stored data and still result in a expected financial loss for cheaters. This should encourage the use of the protocol since it significantly lowers the financial loss if the exchange fails.

Parts of our protocol can be found in various forms in [6, 7, 17, 19, 20, 21, 22, 40]. However, we believe that the combination of those parts and the collateral mechanism we use is unique. The FairSwap protocol presented in [8] and discussed in Section 7, by contrast, gives only a cursory treatment of DDoS attacks and collateral. Additionally, our protocol allows the hosts to easily exchange the data without Alice participating in the transaction, unlike Example 1. Furthermore, the computational cost and space required on our contract are both constant. For FairSwap, on the other hand, the computational cost on the contract is logarithmic in the data size in the case where someone cheats (and a small constant when everyone behaves). The work from Alice and Bob grows linearly in the amount of data, as in FairSwap, but not in the Zero Knowledge Contingent Payment method in [15] and discussed in Section 7, which requires much more work.

1.2 Organization

We have organized the rest of this paper as follows. In Section 2 we explain the various mechanics we need for our protocol. We present the protocol in Section 3, followed by its security analysis in Section 4. We present a proof that an arbitrarily low collateral (as a fraction of the total cost) can be achieved in Section 5. We have a discussion on parameter choices in Section 6. In Section 7, we discuss related work. We have a brief conclusion in Section 8. In Appendix A we give details of our smart contract implementation.

2 Foundational Technologies

To provide data integrity, privacy, and anonymity for both parties, our proposed data storage protocol combines five independent, well-studied mechanisms that encourage Alice and Bob to follow the protocol faithfully. The protocol makes use of an error-correcting code scheme, encryption, a hash list, a collateral mechanism, and a smart contract, as presented below. We also discuss distributed ledgers, decentralized storage, decentralized applications, and secure links.

2.1 Distributed Ledger

A ledger is a list of all transactions relating to an organization. Traditionally, ledgers are controlled by a central entity such as a bank or a government. These "centralized" ledgers are susceptible to modifications by the entity or nefarious individuals. A decentralized ledger is a ledger that is not controlled by one entity, but by many participants in a decentralized network. Thus, it would take a majority of the participants to modify a ledger. In a distributed ledger, participants vote on an official ledger that comes from the ledgers submitted by the participants. Participants build their own ledger independently by processing every transaction they receive. A blockchain, first used with the cryptocurrency Bitcoin [28], is an example of a distributed ledger. The cryptocurrency Ethereum [11] is the first blockchain that supports smart contracts. See Section 2.3 for more information on smart contracts.

2.2 Distributed Storage Network

A distributed storage network (DSN) is one that stores data in many locations. The locations can be in the same building or on servers across the world. A distributed storage system is said to be decentralized if the storage space is not controlled by one entity. Filecoin, Sia [6], and Storj [40] are examples of decentralized storage networks that

use distributed ledgers to manage the system [1]. For example, Filecoin uses the InterPlanetary File System[2] (IPFS), a hypermedia distribution protocol, for content addressing. They then built an incentive layer on top of IPFS using a blockchain to encourage people to offer their unused storage. In this paper, we do not discuss all of the details of how to set up a distributed storage network. Instead, we imagine that this is already set up, and the client stores her data on this network. We provide a mechanism for anonymous data retrieval (and the setup to make that retrieval possible). We also assume the networks charge per bit of data stored.

2.3 Smart Contract

Our protocol will leverage smart contracts running on a cryptocurrency-based distributed ledger. The cryptocurrency will enable payments to be made between parties. However, a cryptocurrency payment scheme is not sufficient because, with direct atomic payments, one party may defraud another.

To prevent fraud, we use a smart contract. Smart contracts are programs that reside on a blockchain. They can be made immutable, and their code is publicly accessible, so all parties know exactly how they will behave. They are executed in parallel by all maintainers of the cryptocurrency, and these block producers check that the producer of the next blockchain block updated the contract state correctly. Smart contracts can receive, store, and send cryptocurrencies as well as perform Turing complete computations. They have built-in functionality to leverage the cryptocurrency accounts used by the blockchain to authenticate users (verifying ownership of the private key associated with an account). This collection of features enables our smart contract to act as a trusted arbiter between parties involved in our data transfer protocol.

As discussed in [35], the underlying cryptocurrency aims to provide pseudonymity. The activity of a user account can be tracked on the blockchain, but not linked to individuals. Since accounts do not need to be registered to be used, users can create many accounts; they can even create a unique account for each transaction to make it even more difficult to link an account to a person. Creating an account is as simple as generating a public-private key pair. We will assume that the cryptocurrency used in this protocol correctly ensures pseudonymity. The protocol will still work in the context of this constraint. It does this by identifying users based on their account number (which maps to the generated public key) and authenticating them based on their ability to prove knowledge of the associated private key.

2.4 Collateral

A breach of contract can be punished by damaging the party's reputation or through a monetary penalty. If the parties are anonymous, then it may be difficult to damage a party's reputation. Thus, we look to storing the collateral in escrow that is managed by a smart contract on a decentralized distributed ledger.

It is important to point out that a smart contract can only punish protocol violations based on information found on the distributed ledger. The smart contract cannot check the veracity of any claim of a protocol violation that occurs off the ledger unless adequate information is supplied to the contract. Which is difficult to do efficiently. One solution to this is for the smart contract to hold the collateral in escrow and to punish all parties in the event that one party reports a breach in protocol. This notion has roots stemming from the Ultimatum game [16] and is sometimes referred to as mutually assured destruction (MAD). The term MAD is a throw back to nuclear deterrence during the cold war. The use of MAD in escrow accounts was first suggested in [21] and used in [17, 20, 7]. However, in those papers the amount of assets put in escrow is the cost of the data plus insurance. Although, [20] handles the payment and the insurance separately so that the parties would only lose their insurance. Our method allows us to use the MAD mechanism with a cost significantly less than the cost of the data.

2.5 Decentralized Application (DApp)

Since the focus of our paper is on the retrieval mechanism, we will assume for the rest of this paper that Alice relies on some storage market, in the form of a decentralized application (DApp), to match her with Bob when she requests her data. A DApp is an application that is made up of a collection of smart contracts that run on a distributed ledger and can consist of off ledger software that interacts with it. We are assuming here that the management of Alice's data is decentralized. Alice will provide the DApp with all necessary information to run our protocol. She may also provide any price or storage length options. The DApp will pass that information on to all potential hosts. Every host that agrees to store Alice's data should pay a small deposit[3] to keep them invested. The deposit will be returned when the data is no longer needed. At Alice's request the DApp will choose a host (Bob), this can be done with a random selection, bidding

[1] Any mention of commercial products is for information only; it does not imply recommendation or endorsement.

[2] https://ipfs.io/

[3] Filecoin [22] requires each host to pay a collateral equal to the size of the data storage they are offering.

system, or a variety of other ways, to return Alice's data. Once Bob is selected, the DApp will create a smart contract initialized with all necessary information so that Alice and Bob can complete the transfer with our protocol. We assume that Alice and Bob communicate with an anonymous messaging system and use an anonymous file sharing channel to transfer data.

2.6 Error-Correcting Code and Interleaving

Error-correcting codes are a standard method for efficiently building redundancy into data so that errors and erasures can be corrected. This paper will use a type of error-correcting code called a block code. Given a block of data with k bits, we can use a block code to encode the data into a block of n bits (where $n > k$). To encode a larger amount of data, one divides the data into blocks of size k and applies the above encoding to each block to produce a set of blocks of size n. The ratio k/n is known as the rate of the code. These blocks should not be confused with the larger pieces that are used for the remainder of the paper, including later in this section, as well as Sections 3, 4, 5, and 6, for interleaving, storage, and retrieval.

We use Reed-Solomon codes, although similar results will hold for other codes. With Reed-Solomon codes, if the encoded message is corrupted and we can recover at least k of those n bits in a block, then we can decode those k bits to produce the original message. For a brief primer on how error-correcting codes are used in storage systems, see [38]. For a more in-depth introduction to error-correcting codes, see [26].

A common problem with using block codes alone is their inability to correct "burst errors", a string of several errors in a row. Since these are exactly the sort of errors expected with our protocol, another trick is required: interleaving, see [45]. Interleaving involves permuting the data so that errors which were formerly next to each other are now spread out. A random permutation will work, but then one must keep track of the permutation. This is not prohibitive, but neither is it desired. Instead, we use a block interleaver, which performs a matrix transposition on the data. To understand how this works, imagine that the data is split into b pieces of m bytes each. The data can be rearranged as m pieces of b bytes each in the following way: Number the pieces 1 to b, and number the bytes of each piece 1 to m. For the first byte of the first new piece, take the first byte of the first old piece. But for the second byte of the first new piece, take the first byte of the second piece, and for the ith byte, take the first byte of the ith piece. Continue in the same manner to produce the rest of the pieces. This has the benefit that any two bytes in one of the old pieces are now at a distance at least b from each other, which is maximal. This is optimal in the case where entire pieces are expected to be deleted, but we wish to keep bytes together. Note that now there are m pieces, each of size b, so that the roles of m and b are interchanged.

2.7 Cryptography

Outside of the cryptographic security involved with the use of a smart contract on a decentralized ledger, we use a hybrid cryptographic scheme in our protocol. A standardized version is the Elliptic Curve Integrated Encryption Scheme (ECIES), which involves elliptic curve cryptography and symmetric key cryptography. A standard for this can be found at [1] or [41], and all of the same ingredients are in [33]. A single elliptic curve public key will be used to generate several symmetric keys. This will allow Bob to encrypt data in a way that Alice will be able to verify. This involves elliptic curve Elgamal encryption on an elliptic curve with a standard base point (a 256 bit version is suggested for 128-bit security, see [9] and [32]), symmetric key cryptography (the Advanced Encryption Standard 256, or AES-256 is suggested, see [30]), and a key derivation function, or KDF, such as KDF2 or KDF3, see [31]. The KDF may use a cryptographic hash such as the Standard Hash Algorithm, SHA-256, see [34]. The hash list that we describe in Section 2.8 will play the role of message authentication, so that part of the ECIES protocol can be ignored. Identity verification is not necessary, since parties are anonymous, and there will be underlying encryption which only the owner can decrypt for privacy. Since ECIES is Elliptic Curve Diffie Hellman (ECDH) plus a hash (SHA-256) plus symmetric key encryption (AES), we can estimate timing by adding timings of the component parts. On a Macbook Pro 2.8 GHz Intel Core i7, the timings are in Table 1. That means that the ECDH plus SHA, which is independent of the amount of

Table 1: Cryptographic Algorithm Timings

Algorithm	speed	units
ECDH-256	12 437.6	ops/s
SHA-256	70 053.77	kb/s
AES-256	95 582.73	kb/s

data to be encrypted in this case (we only hash the 64 byte output from ECDH), takes 8.08×10^{-5} seconds, or 12 375.1

operations per second. Timing may vary, depending on background processes, etc. If we include AES along with these algorithms, a kilobyte would take about 9.13×10^{-5} seconds.

Let G be the base point for the elliptic curve (this is just a standard point that everyone knows about and agrees to). Bob will use a single static secret (a positive integer x modulo some prime) and public key elliptic curve point P with $P = xG$ for his Elgamal encryption. He will have to generate several secret and public ephemeral key pairs (y_i, Q_i) with $Q_i = y_iG$. Then, he calculates the secret xy_iG from these keys. The KDF is a function whose input is such an elliptic curve point (in bit string form) and whose output is a symmetric key. It is usually just a specification for how to use a hash for this purpose. Bob applies the KDF to the secret xy_iG to produce the symmetric key which he uses to encrypt the piece of data. Bob can send the encrypted pieces together with the public ephemeral key to Alice, who will not be able to decrypt the data. The reason for this will become clear in Section 3.3, but the summary is that it will allow Alice to verify what key was used to encrypt a piece without getting the secret static key or all of the decrypted data. This is backwards from the usual protocol, where Alice would do all of this using Bob's public key to send to Bob, who would be in a position to decrypt.

2.8 Hash List

A cryptographic hash is a function that maps arbitrary strings to numbers of a fixed size. The function should have two properties. First, it should be computationally difficult to reverse. This means that given an output of a hash function (also called a hash or a digest), the fastest way to find the input that produced that output is to guess and test all possible inputs until one finds an input that produces the correct output. Second, it should be computationally difficult to find any collision, where any two different inputs produce the same output. One example of a cryptographic hash is SHA-256, see [34].

If data is broken up into pieces, then a hash of each piece can be kept in a list. This is called a hash list. With such a list, it is possible to separately verify that each piece is authentic and not corrupted. The hash of the hash list itself is often used to show that the hash list is not corrupted. This is called the root hash. Normally the root hash is obtained from a trusted third party. However, in our trust-less system, we will store the root hash on a smart contract.

2.9 Secure Link

In our protocol, it does not matter how Alice and Bob securely share data; some mechanism must be used. As an example, the open source project called OnionShare [23] promises the ability to share data of any size securely and anonymously. Another possibility is that Alice and Bob could first start with an anonymous and secure chat service to arrange the data sharing method and to mitigate any problems that may arrive during the exchange.

3 Protocol

In this section, we present the details of our data storage protocol. We first describe the procedure that the client (Alice) needs to complete to initialize the storage of her data with the DApp (see Section 2.5). We end with the delivery protocol for the host (Bob) to send the data to Alice and to receive payment.

3.1 Storage - Client

This subsection describes how Alice prepares her data that will be stored by multiple hosts. This will involve encryption, encoding, interleaving, and hash list generation.

Alice will need to set up an anonymous account with the DApp. She will use this account to share the parameters needed to run the protocol with the hosts. She may have to indicate how long she wants to store the data an pay appropriate fees. Depending on the business model of the DApp, it could collect a returnable deposit or charge user fees. The point here is to prevent Alice from disappearing without penalty and to deter Alice from spamming the hosts with junk data.

The first thing Alice should do is encrypt her data with a symmetric encryption algorithm to preserve privacy. She will need to keep track of this secret key for the duration of the storage (i.e., this is not transferred to the hosts). The next steps will manipulate this encrypted data, but a copy of the encrypted data must be set aside at this stage to be sent to the hosts at a later time.

Alice will then need to run the following three-step generation process on the encrypted data. First, Alice encodes the encrypted data using an error-correcting code such as a Reed-Solomon code, and then interleaves the encoded data, which breaks the data into pieces. This is described in Section 2.6. Secondly, each of the pieces are hashed separately

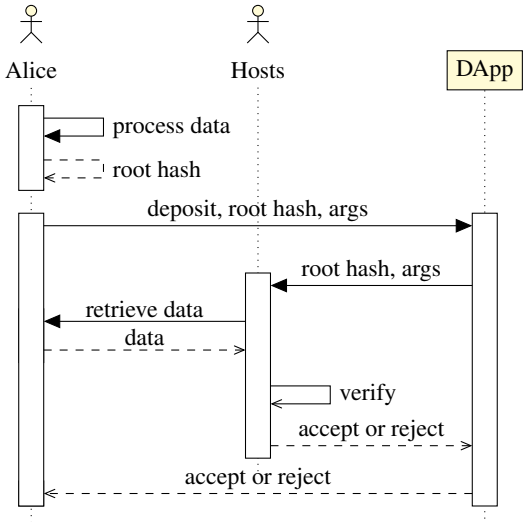

Figure 1: Storage of Alice's data

and each hash added to a hash list. Finally, Alice hashes the hash list. This 'root' hash is sent to the Dapp. Alice can now delete the pieces and hash list.

Next, Alice pays a small deposit and any fees to the DApp, and posts a secure link on the DApp where potential hosts can retrieve the data (see Section 2.9). Alice also posts the following parameters to the DApp: the values k and n for error correction (see Section 2.6), the piece size b, the number of pieces m (again see Section 2.6), and the number t of sample pieces to be verified in the retrieval portion of the protocol in Section 3.3. A discussion of how to choose t can be found in Section 5.

3.2 Storage - Host

All hosts that interact with the DApp who are interested in storing Alice's data can access the necessary parameters on the DApp and use the secure link posted by Alice to retrieve her data. After downloading the encrypted data, each host will use the parameters posted by Alice and the procedure described in Section 3.1 to reproduce the root hash posted by Alice on the DApp. If the resulting hash matches the root hash, then the hosts can safely agree to store Alice's data. At this point the storing host pays a deposit to the DApp. This will deter them from disappearing with Alice's data. The hosts, if they wish, may now delete the pieces and hash list, and simply store the data until Alice retrieves it.

3.3 Retrieval

Recall for our protocol that we are assuming that there is some outside DApp that selects, at Alice's request, one of the hosts that is storing Alice's data (whom we refer to as Bob). We assume the fee was agreed upon fairly between the DApp and Bob at market rates. We will denote Bob's fee by F. The DApp will generate a smart contract that contains all the necessary information so that Alice and Bob can complete the protocol. Only Alice and Bob will be authorized to interact with the contract. The DApp will collect Bob's collateral C_B and public Elgamal key and place them on the contract. Alice will post her collateral C_A and a secure link encrypted with Bob's public key so that they can transfer data. The link is encrypted so that nobody but Bob can use it. Besides preventing cheating below, the collateral will also prevent DDoS attacks from both parties.

Bob will have to regenerate the pieces and the hash list as described in Section 3.1. At this point Bob should randomize the ordering of the pieces and encrypt each piece using ECIES with his *own* public key P, as described in Section 2.7. Each public ephemeral key Q_i will be attached to its corresponding piece, together with a number indicating the ordering as a header. Bob's use of hybrid encryption is to speed up encryption and to save bandwidth later when sending data back and forth with Alice. Once this is done, Bob will send the encrypted pieces and hash list to Alice through the secure link, and keep a copy for himself.

Upon receiving the pieces and hash list, Alice will calculate the root hash of the hash list and compare it to the one stored on the contract. If the hash matches, Alice will select t pieces at random for testing. In Section 6 we discuss

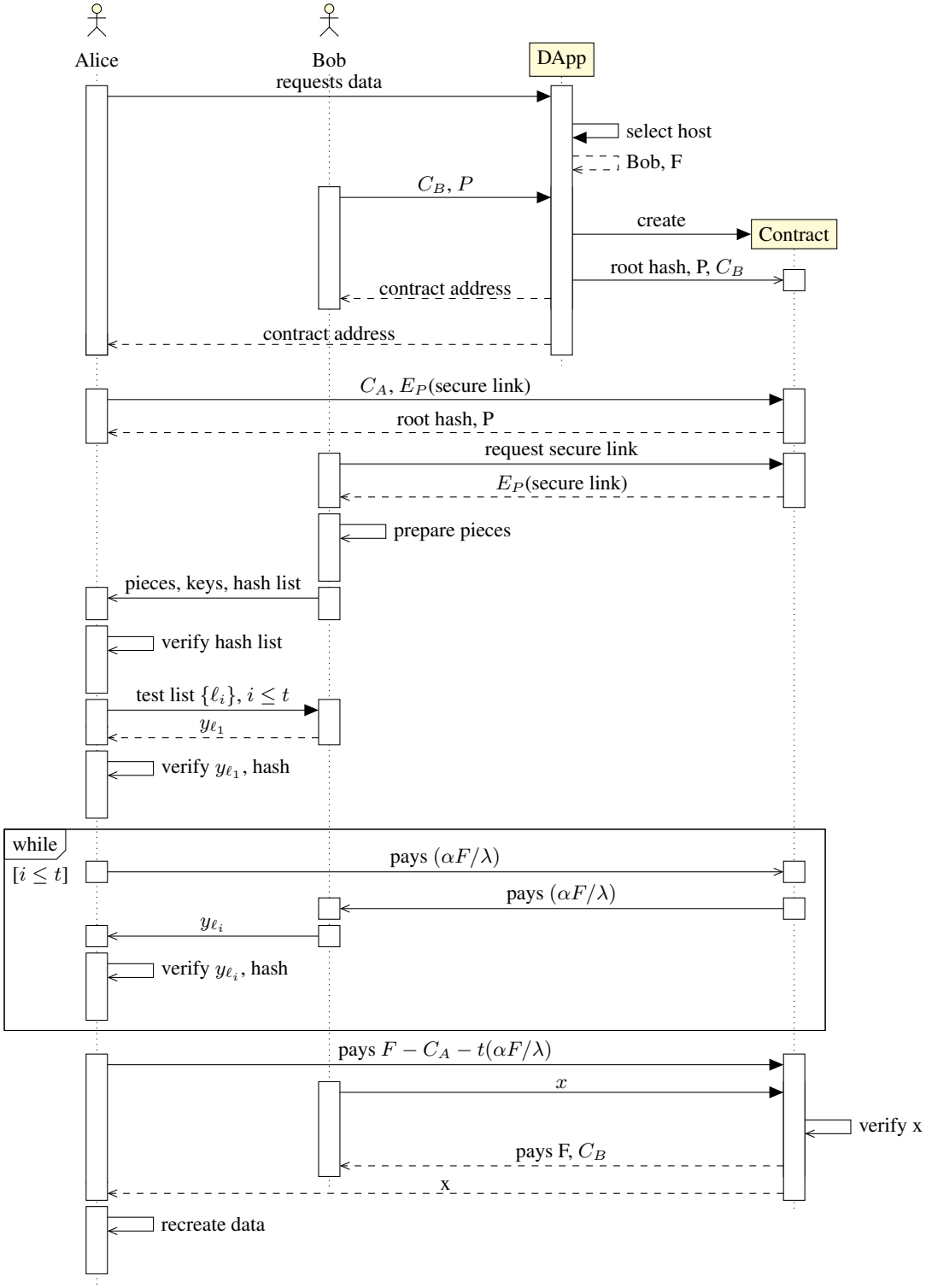

Figure 2: Retrieval of Alice's Data

possible values for t. Alice will then send this list of t random selections (in random order) to Bob. Note that Alice does not have to send the actual pieces back, only the positions $\{\ell_i\}$ of the t pieces specified by the numbering in the header.

Once Bob has the list of t piece numbers, he sends the secret ephemeral key y_{ℓ_1} corresponding to the first number on Alice's list to Alice. Alice uses the secret ephemeral key y_{ℓ_1} to compute the public ephemeral key $y_{\ell_1}G$ and compare with the one that Bob provided, Q_{ℓ_1}, together with the corresponding piece he sent earlier. If these match, Alice uses Bob's public key P on the contract to compute the secret $y_{\ell_1}P = xy_{\ell_1}G$. Then, she uses the KDF (see Section 2.7) to derive the symmetric key. Next, she uses the symmetric key decryption algorithm with the symmetric key to decrypt the corresponding piece. Alice searches for a match with one of the hashes on the sorted hash list. If the hash matches one in the list, then for $0 < \alpha \leq 1$, Alice pays the smart contract a fee of $\alpha F/\lambda$ monetary units, where $\lambda = \lceil \frac{m}{b} \rceil$ is the number of pieces required to reconstruct Alice's data using the error-correcting code, and α is the fraction of the "full price" F/λ of a piece Alice pays at this stage. The nature of the units depends on the platform used. We will revisit the importance of α in Section 4 and Section 5. Note that if Bob had used a different key to encrypt from the key $P = xG$ that is posted on the contract, Alice would have calculated the wrong shared secret $xy_{\ell_1}G$, and not been able to decrypt this piece. That is why Bob could not simply reveal the symmetric key: Alice needs to check that P was actually used to encrypt this piece before she is willing to pay for x. The contract will pay Bob immediately, which in turn signals to Bob to send another private ephemeral key. The above steps will be repeated until all t private ephemeral keys have been sent to Alice.

In Section 4 we show that with the right choices of t and m Bob will lose money on average if he cheats. If Bob cheats during the procedure and is caught, Alice will notify the contract. At that point the contract will permanently lock up the funds and Alice is free to request her data from another host. If Alice is satisfied with the t test pieces, then she can request Bob's private key by paying the remaining

$$F - C_A - \frac{F}{\lambda}(t-1)$$

to the contract. Note that Bob cannot post his private key x until after Alice requests it. Once Bob posts his private key, the contract will verify that it matches the posted public key. If the private key posted by Bob does not match the public key, then the contract will release all of the funds to Alice. Otherwise, the contract releases all funds to Bob. With Bob's private key x, together with all of the public ephemeral keys Q_i, Alice can compute the secrets xQ_i and use the KDF to derive the symmetric keys. With those, she can decrypt all of the pieces. Then she can de-interleave, decode, and perform the last layer of symmetric decryption using the key that she kept at the beginning of this storage process.

4 Security Analysis

The goal of this protocol is that Alice receives her data and Bob is paid. Smart contracts are well studied and can easily handle the distribution of funds and key verification in a secure manner. Thus we will focus our analysis on the off-contract activities of Alice and Bob.

Since the contract cannot arbitrate activities that happen outside of the decentralized ledger, any collateral penalty must punish both Alice and Bob. This will deter both parties from openly cheating, as they will both lose. Since collateral also deters both parties from walking away at every stage before the end, it also prevents DDoS attacks. However, we need to determine the lowest collateral that yields a negative expected payout for both Alice and Bob. This can be done by designing a protocol such that cheating is detectable with high probability.

Our solution is to use error-correcting-codes with interleaving applied to Alice's data. We assume there are m pieces, and that Alice needs any λ of them to rebuild her data. Let t be the number of pieces that Alice has decided to test.

Recall that $1 \geq \alpha > 0$, and Alice will pay Bob $\alpha F/\lambda$ for each test piece. It is necessary that

$$\frac{\alpha F}{\lambda}(t-1) < C_B,$$

since Bob may faithfully follow the protocol by storing the data and delivering the test pieces, but then may simply abandon the protocol once he is paid for the last test piece. His hope would be to mask his identity and try to sell the data a second time to Alice. Similarly, since Alice is only paying $\alpha F/\lambda$ per test piece, her collateral would have to be

$$C_A > \frac{Ft}{\lambda} - \frac{\alpha F}{\lambda}(t-1) = \frac{F}{\lambda}\left((1-\alpha)t + \alpha\right) \tag{1}$$

to prevent her from repeatedly walking away from the protocol in an attempt to rebuild her data by paying less than F. It is actually worse for Alice because she does not know the identity of the test pieces, so she would most likely buy several pieces multiple times.

Suppose Alice wants to store data D, but engineers a larger data set W and stores it such that when W is returned she can rebuild D from the test pieces for W. Her hope is that she will pay less than the cost of storing D directly. Suppose Alice needs s of the t test pieces to rebuild D. In the best case Alice can quit the protocol after rebuilding D with the first s test pieces. However, she will have to pay

$$\frac{\alpha F}{\lambda}(s-1)$$

for the test pieces and lose her collateral. This amounts to

$$C_A + \frac{\alpha F}{\lambda}(s-1) > \frac{F}{\lambda}((1-\alpha)t + \alpha + \alpha(s-1)) \geq \frac{sF}{\lambda}.$$

We let $|W|$ and $|D|$ be the number of bits in W and D respectively. Assuming the market charges per bit of data, Alice will pay for $|D|$ bits of data at a price of $\frac{F}{|W|}$ if she stored D directly. With respect to storing W Alice will pay for $\frac{s|W|}{\lambda} \geq |D|$ bits of data. From this we can see the cost of storing her data D directly is at most

$$\frac{s|W|}{\lambda}\frac{F}{|W|} = \frac{sF}{\lambda}.$$

Therefore Alice will pay more than the cost of storing D with this strategy.

One of the main security concerns is that all of the pieces that Alice checks will be correct, Bob will be paid, and Alice will not be able to rebuild her data from the pieces Bob sent. Bob may hope that if he successfully cheats Alice, then he can mask his identity and try to sell the data to Alice again. Bob may do this by using a different public key than the one on the contract. As explained below, Bob may also engineer the pieces so that Alice cannot use them to rebuild her data. Due to our use of error-correcting codes, we can show that Alice can discover either of those situations with a sufficiently high probability so that Bob will have a negative expected payout.

Bob can try to cheat by using different keys or by not delivering enough unique pieces to Alice. The latter method can be done by either encrypting false data or duplicating other pieces. Either way, Bob would need to do this with at least $m - \lambda + 1$ pieces. Otherwise, there would be λ distinct pieces that Alice could rebuild from once she obtains the private key from Bob. To succeed with this strategy, Bob will want to maximize the number of ways Alice can select t pieces without detecting the cheat. Clearly, Bob is less likely to be caught if he encrypts a duplicate piece than if he encrypts a false piece. Similarly, using different keys would have the same effect, since Alice would catch Bob either way as soon as she found either a false piece or one that used an incorrect key. Thus, Bob may try the duplicate strategy.

We need to show that if Bob attempts the duplicate piece cheating strategy, he should expect a loss for t sufficiently large. Alice cannot catch Bob with a single test. Thus $t \geq 2$. Let X be the discrete random variable that may take on values $\{2, \ldots, \lambda\}$, where x_i represents the event that Bob is caught cheating on the ith test. Let $P(X = x_i) = q(i)$. We let $Q(t) = \sum_{i=2}^{t} q(i)$ and note that $Q(\lambda) = 1$. We can calculate the expected value of X:

$$E(X) = \sum_{i=2}^{\lambda} E(x_i) = \sum_{i=2}^{\lambda} q(i)i. \tag{2}$$

Observe that $P(X \leq t) = Q(t)$. Thus, the probability that Bob was not caught after t tests is

$$1 - P(X \leq t) = 1 - Q(t). \tag{3}$$

We now calculate the expected payout after t tests. Each test piece is worth $\frac{\alpha F}{\lambda}$. Since Alice does not pay for the first piece, Bob is only paid

$$\frac{\alpha F}{\lambda}(i-1)$$

if Alice requests the first i pieces. Let Y_t be the random variable where Y_t takes on the values

$$\left\{\frac{\alpha F}{\lambda}(x_2 - 1) - C_B, \ldots, \frac{\alpha F}{\lambda}(x_t - 1) - C_B\right\}$$

if Bob is caught within the t tests and $Y_t = F$ if he is not caught. We now have the expectation

$$\begin{aligned} E(Y_t) &= (1 - Q(t))F + \sum_{i=2}^{t} q(i)\left(\frac{\alpha F}{\lambda}(i-1) - C_B\right) \\ &= F(1 - Q(t)) - \left(C_B + \frac{\alpha F}{\lambda}\right)Q(t) + \frac{\alpha F}{\lambda}\sum_{i=2}^{t} q(i)i. \end{aligned} \tag{4}$$

Since we are interested in a negative expected payout we set $E(Y_t) < 0$. Solving for C_B we have

$$\frac{(1-Q(t))F}{Q(t)} + \frac{\alpha F}{\lambda}\left(\frac{\sum_{i=2}^{t} q(i)i}{Q(t)} - 1\right) < C_B. \tag{5}$$

Combining our derivations from above, both Alice and Bob will have an expected loss if C_A and C_B are chosen so they satisfy the following lower bounds.

$$\begin{aligned} C_A &> \frac{F}{\lambda}\left((1-\alpha)t + \alpha\right) \\ C_B &> \max\left\{\frac{\alpha F}{\lambda}(t-1), \frac{(1-Q(t))F}{Q(t)} + \frac{\alpha F}{\lambda}\left(\frac{\sum_{i=2}^{t} q(i)i}{Q(t)} - 1\right)\right\} \end{aligned} \tag{6}$$

5 Proof of Low Collateral Requirements.

In this section we show that for any fixed F, $1 \geq \alpha > 0$ and any given $\varepsilon > 0$, there exists a λ, t, C_B, and C_A such that $E(Y_t) < 0$, $max\{\frac{t}{\lambda}, \frac{C_B}{F}, \frac{C_A}{F}\} < \varepsilon$. In particular, we show that for every $\varepsilon > 0$ there is a λ and t such that $C_B = \frac{\alpha F}{\lambda}(t+1)$ and $C_A = \frac{F}{\lambda}(t+\alpha)$ satisfy the desired ratios. However, C_B and C_A can often be made significantly lower for smaller λ and t, see Figure 3. It is important to point out that λ is limited by the minimum piece size, so these inequalities will only hold for sufficiently large amounts of data.

This section proceeds by first finding an upper bound on the probability that Bob is not caught cheating. We then show that with the right parameters, this upper bound can be arbitrarily small.

Let B be the set of unique pieces Bob sends to Alice. We let a_i be the number of pieces identical to piece i, including piece i itself.

The number of ways Alice can select t pieces one at a time without discovering a duplicate is

$$\sum_{\substack{T\subset B \\ |T|=t}} t! \prod_{i\in T} a_i. \tag{7}$$

We now calculate $P(X = t)$. This means that the first $t-1$ pieces do not match, but piece t matches one of the first $t-1$. Now T will therefore be the set of the first $t-1$ non-matching pieces. As above, there are

$$(t-1)! \sum_{\substack{T\subset B \\ |T|=t-1}} \prod_{i\in T} a_i$$

ways to choose these $t-1$ pieces. For each j in T there are $a_j - 1$ ways to choose piece t so that it matches piece j. Therefore, there are

$$(t-1)!\left(\sum_{\substack{T\subset B \\ |T|=t-1}} \sum_{j\in T} (a_j - 1) \prod_{i\in T} a_i\right)$$

ways that Bob can be caught at piece t. Since the total number of ways to choose t pieces remains unchanged, Thus

$$P(X = t) = \left(\sum_{\substack{T\subset B \\ |T|=t-1}} \sum_{j\in T} (a_j - 1) \prod_{i\in T} a_i\right)\left(t\binom{m}{t}\right)^{-1}. \tag{8}$$

The number of ways to select t pieces one at a time from the m total pieces is $t!\binom{m}{t}$. Thus, from Equation 7, we have

$$1 - P(X \leq t) = \left(\sum_{\substack{T\subset B \\ |T|=t}} \prod_{i\in T} a_i\right)\binom{m}{t}^{-1} \tag{9}$$

as the probability of Bob successfully cheating with the duplicate strategy. We would like to bound this probability.

Since the tth elementary symmetric polynomial is

$$S_t = \binom{|B|}{t}^{-1} \sum_{\substack{T \subset B \\ |T|=t}} \prod_{i \in T} a_i,$$

we can see that $1 - P(X \leq t) = \binom{|B|}{t}\binom{m}{t}^{-1} S_t$. By Maclaurin's inequality, $S_t^{1/t} \leq S_1 = |B|^{-1} \sum a_i$, which in our case is $m/|B|$. In the case where m is divisible by $|B|$, this upper bound is sharp, since if we take every $a_i = m/|B|$, then $S_t = (m/|B|)^t$. Since Bob can send at most $\lambda - 1$ unique pieces without allowing Alice to recover her data, we set $|B| = \lambda - 1$. This leads to the upper bound

$$1 - P(X \leq t) \leq \frac{\binom{\lambda-1}{t}\left(\frac{m}{\lambda-1}\right)^t}{\binom{m}{t}} \tag{10}$$

for Equation 9.

We let $w = \frac{m}{\lambda-1}$, where w^{-1} is approximately the rate of the error-correcting code (see Section 2.6) and simplify the right hand side of Equation 10 to

$$\prod_{i=0}^{t-1} \frac{w(\lambda-1) - wi}{w(\lambda-1) - i}. \tag{11}$$

To finish this section, it will be more convenient to generalize Equation 11. Fix $w > 1$ real. For positive real x and positive integers ℓ such that $\ell - 1 \leq x$, we let

$$f(x, \ell) = \prod_{i=0}^{\ell-1} \frac{wx - wi}{wx - i}.$$

Note that $f(\lambda - 1, t) \geq 1 - P(X \leq t)$. Observe that the function f is increasing in x and decreasing in ℓ and w. One way to see this is to take partial derivatives of each factor

$$\frac{wx - wi}{wx - i}$$

with respect to x, w, and i. Since each term is less than one, the more terms there are, the smaller the product, proving that f is decreasing in ℓ. Since this term is also decreasing in i, each term is smaller than the last. We can also see that

Lemma 5.1. *For all real $x \geq 1$, and all integers $s, t \geq 1$,*

$$f(xt, s + t) \leq \left(\frac{w(x-1)}{wx-1}\right)^s f(xt, t). \tag{12}$$

Proof. Observe that

$$f(xt, t+1) = \frac{wxt - wt}{wxt - t} f(xt, t) = \frac{w(x-1)}{wx-1} f(xt, t).$$

Next, observe that since the terms in the product $f(xt, t + s)$ are decreasing, the subsequent terms after the $(t + 1)$st term are smaller than $\frac{w(x-1)}{wx-1}$. Since there are $s - 1$ subsequent terms, this proves the lemma. □

Lemma 5.2. *For every $\alpha, \varepsilon > 0$ there is a positive x_0 such that for every $x \geq x_0$ there is an integer s such that $f(x, 2s) < \frac{\alpha}{x+2}$ and $3s/x < \varepsilon$.*

Proof. Fix $\alpha, \varepsilon > 0$. Let $y > 3\varepsilon^{-1} + 1$. Then there is some s_0 such that

$$\left(\frac{w(y-1)}{wy-1}\right)^s < \frac{\alpha}{ys+2}$$

for all $s \geq s_0$ since the left hand side decreases exponentially in s while the right hand side decreases as $1/s$. We also require s_0 to be greater than y. Now let $x_0 = ys_0$. Fix $x \geq x_0$, and set $s = \lceil \frac{x}{y} \rceil \geq s_0$. Observe that $x \leq ys$. Then

$$f(x, 2s) \leq f(ys, 2s) \leq \left(\frac{w(y-1)}{wy-1}\right)^s f(ys, s) < \frac{\alpha}{ys+2} < \frac{\alpha}{x+2}.$$

Since $x \geq ys - y$ and $s > y$, $\frac{3s}{x} \leq \frac{3s}{ys-y} < \frac{3}{y-1} < \varepsilon$. □

Using ε and $x = \lambda - 1$ in Lemma 5.2 we have that

$$1 - P(X \leq x_t) = f(\lambda - 1, 2s) < \frac{\alpha}{x+2} < \frac{\alpha}{\lambda + \alpha}$$

when $t = 2s$.

We wish to use Equation 5 to choose a small C_B using the above observations. Choose t and λ as above, and use $Q(t) = 1 - f(\lambda - 1, t)$ and $1/Q(t) < (\lambda + \alpha)/\lambda$ to compute

$$\begin{aligned} \frac{(1-Q(t))F}{Q(t)} + \alpha \frac{F}{\lambda} \left(\frac{\sum_{i=2}^{t} q(i)i}{Q(t)} - 1 \right) \\ < F \left(\frac{\lambda + \alpha}{\lambda} - 1 + \frac{\alpha}{\lambda} \left(\frac{\lambda + \alpha}{\lambda} \sum_{i=2}^{t} q(i)i - 1 \right) \right) = \frac{\alpha F}{\lambda} \left(1 + \frac{\alpha}{\lambda} \right) \sum_{i=2}^{t} q(i)i. \end{aligned} \tag{13}$$

Since $\sum_{i=2}^{t} q(i)i \leq \sum_{i=2}^{t} q(i)t \leq t$, we have that

$$\frac{\alpha F}{\lambda} \left(1 + \frac{\alpha}{\lambda} \right) \sum_{i=2}^{t} q(i)i \leq \frac{\alpha F t}{\lambda} \left(1 + \frac{\alpha}{\lambda} \right) < \frac{\alpha F}{\lambda} (t+1).$$

Therefore, we can set $C_B = \alpha F(t+1)/\lambda$. Since $t + \alpha > t(1-\alpha) + \alpha$ we can let $C_A = F(t+\alpha)/\lambda$. Note that C_B is at most C_A, and

$$\frac{t}{\lambda} < \frac{C_A}{F} = \frac{t+\alpha}{\lambda} < \frac{3s}{x+1} < \varepsilon,$$

so the same holds for C_B/F.

6 Discussion

Although the data size expands at different points in time, this protocol does not increase the size of the data Bob is storing. It is only at the beginning and the end of the protocol when Bob generates the pieces and the hash list that the amount of storage briefly increases. If each piece is b bytes, then the expanded data size is roughly $m(b + 65) + 32$ bytes, in comparison with the original data size of λb bytes. This holds because there are about mb bytes of expanded data, plus a 1-byte numbering label, a 32-byte public ephemeral key, and a 32-byte hash for each piece plus a root hash. However, Bob will need to transfer this larger amount of data to Alice. The possible choices for λ are limited by the data size. Therefore, in this section, when λ varies, we are assuming the data size is sufficiently large.

Recall that in Section 5 we proved that for a fixed α, any $\varepsilon > 0$, and for sufficiently large data size there exists a λ and t such that the ratios of needed tests t/λ and collateral C_A/F and C_B/F are less than ε. However, our calculations can be drastically optimized if one wants more accurate predictions, since we were only interested in the existence of such parameters in Section 5. We demonstrate empirically below that as λ increases, those ratios decrease.

As before, we assume that the price of storing Alice's data is F. Recall that for every α, λ, t and F we have the following collateral bounds

$$C_A > \frac{F}{\lambda} ((1-\alpha)t + \alpha) \tag{14}$$

$$C_B > \frac{\alpha F}{\lambda} (t-1) \tag{15}$$

$$C_B > \frac{(1-Q(t))F}{Q(t)} + \frac{\alpha F}{\lambda} \left(\frac{\sum_{i=2}^{t} q(i)i}{Q(t)} - 1 \right) \tag{16}$$

Notice that F can be factored out of Equations 14, 15, and 16. This is convenient for analyzing C_A and C_B as a fraction of F.

Since the right hand side of Equation 15 is increasing with t while the right hand side of Equation 16 is decreasing, we can see that the lower bound of C_B is minimized when its two lower bounds are close to each other. As α decreases, the needed number of tests will have to increase so that C_B can be minimized. If $\alpha = 1$, then any $C_A > 1$ and

$C_B > t$ for some sufficiently large t will result in an expected loss. Instead, we will add the additional constraint that $C_A = C_B$ for the rest of the discussion, although the minimal number of tests may not occur when this holds. Setting $\alpha(t-1) = t(1-\alpha) + \alpha$ and then solving for α, we have that $\alpha = \frac{t}{2(t-1)}$. With this α, we have to find the t that minimizes C_B.

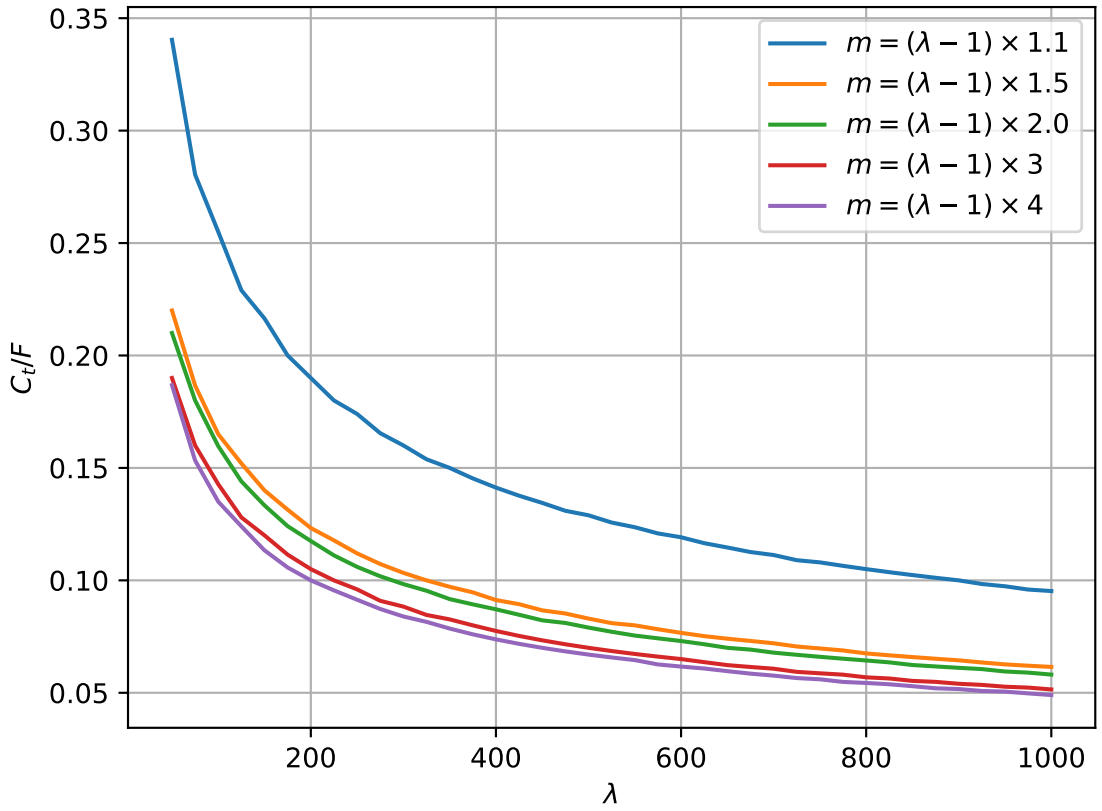

Figure 3: For various m and λ we ran 500 000 trials. For each λ, m and $\alpha = \frac{t}{2(t-1)}$ we found the t that satisfies and minimizes the right hand side of Equation 15 and Equation 16. This strict lower bound is denoted by C_t, and the ratio $\frac{C_t}{F}$ is plotted above.

We performed experiments for various values of λ and m that modeled Alice selecting test pieces until she caught Bob. For $w \in \{1.1, 1.5, 2, 3, 4, 5\}$ we let $m = w(\lambda - 1)$. We fixed λ and w for each experiment. Since Bob has to engineer at least $m - (\lambda - 1)$ pieces, we can use the residue classes modulo $\lambda - 1$ to represent the rest of the pieces. For each trial, we randomly selected a number from 1 to m until we chose one that was congruent modulo $\lambda - 1$ to a number already selected. We repeated the trials 500 000 times while keeping track of how many selections it took for each trial. Next, we created a probability mass function (pmf) to estimate the probability of Alice catching Bob on test i. For each i, we let $\alpha = \frac{i}{2(i-1)}$ and recorded the maximum of the right hand sides of Equation 15 and Equation 16 with F factored out and called it C_i/F. We then found the smallest t such that $C_t/F \leq C_i/F$ for all i. We performed the experiment for various λ larger than 50 and $m = w(\lambda - 1)$ and plotted C_t/F in Figure 3.

From Figure 3 we can see that as λ increases, C_t/F decreases. We also see that C_t/F decreases as w increases. Note that at $\lambda = 50$, our method shows $C_t/F < .25$ and only requires the generation of 100 pieces for $w \geq 1.5$. It is easy to decrease C_t/F, but it does not appear that the extra processing needed for larger w is worth the modest gains past $w = 3$. Therefore, we recommend $w = 2$.

If Alice chooses m such that Bob may evenly distribute the duplicates, then we can reduce Equation 8 to

$$P(X = t) = \frac{(t-1)(w-1)}{m-(t-1)}(1 - P(X \leq t-1)), \tag{17}$$

where $w = \frac{m}{\lambda - 1}$. Choosing such a w allows us to quickly calculate the expected loss for various parameter choices without having to run experiments. For each t we let $\alpha = \frac{t}{2(t-1)}$ and $C_A = C_B$. Letting $w = 2$, for each λ we found the t that minimizes the lower bounds of C_B. We called this minimum C. We then plotted in Figure 4 the ratios t/λ and C/F. The ratios are not strictly decreasing, but overall are decreasing.

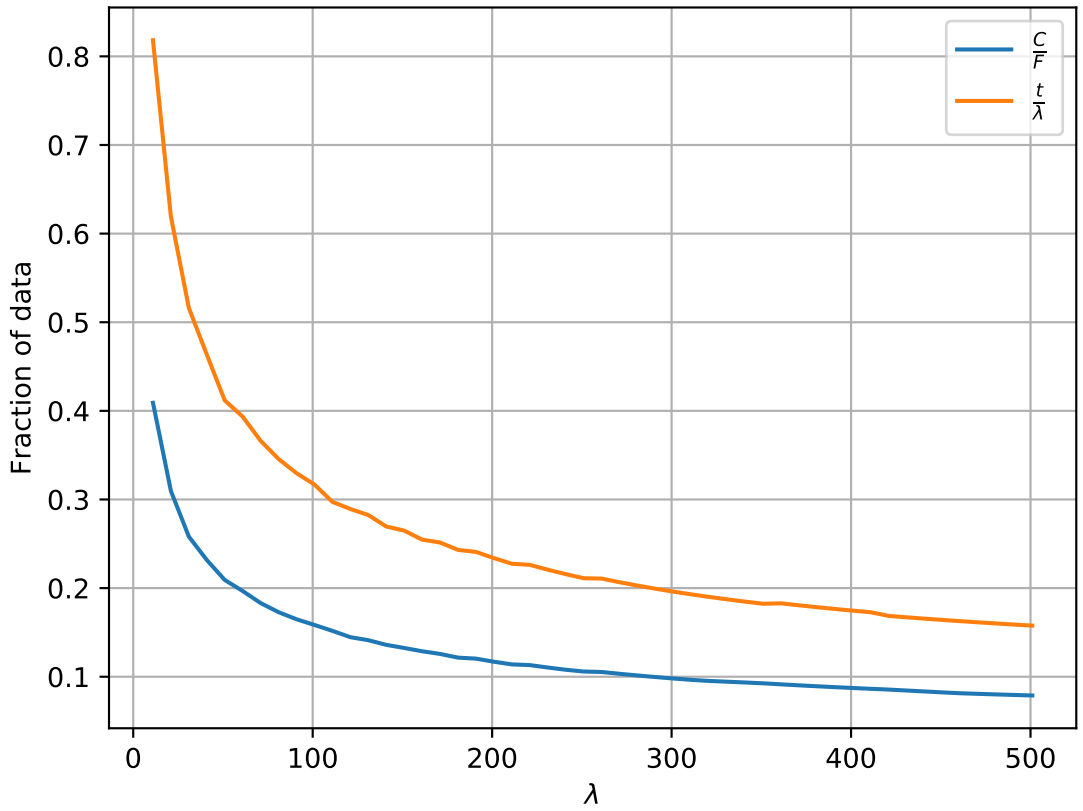

Figure 4: For each λ between 10 and 1000, we let $m = 2 \times (\lambda - 1)$ and found the t that minimizes C_B in Equation 15 and Equation 16 where $\alpha = \frac{t}{2(t-1)}$. We then plotted $\frac{t}{\lambda}$ and $\frac{C}{F}$.

As shown in Table 2 we chose various upper bounds for C/F, and then, using Equation 17, searched for the parameters w, λ, and t with $\alpha = \frac{t}{2(t-1)}$ that minimized m and satisfied the C/F bounds. We can see from the table that if we let $C/F \geq .4$, then m is relatively small, $t/\lambda > .5$ and the encoding only increases the data size at the time of processing by about half. On the other hand for $C/F \leq .30$, m is larger, and the encoding only increases the data size at the time of processing by at most two, but the fraction of tests needed decreases.

Recall that existing solutions have collateral cost near or greater than the cost of storing the data. We see from Table 2 that we can reduce C to less than half the cost of storing the data with very little data processing by both Alice and Bob. However, If the absolute number of tests is not a limiting factor for either Alice or Bob, then there are benefits for both Alice and Bob to increase λ. In general as λ increases the cost per piece $\frac{F}{\lambda}$ decreases. Furthermore, as seen in Figure 4 and Table 2 as λ increases the fraction of tests needed decreases. This correlates to Bob only giving Alice a

$C/F \leq$	.5	.45	.4	.35	.3	.25	.2	.15	.1	.08	.05
w	1.2	1.6	1.5	1.8	1.5	1.8	1.9	2.0	1.8	2.0	2.0
λ	11	11	15	16	29	36	61	110	321	482	1410
t	11	9	12	11	17	18	24	33	64	77	141
$m = w \times (\lambda - 1)$	13	16	21	27	42	63	114	218	576	962	2818

Table 2: Using Equation 17 for the probabilities and fixing various upper bounds of C/F, the table gives parameters that minimizes m

small fraction of her data before she pays full price. Overall, increasing λ lowers Alice and Bob's exposure to financial loss in the unlikely event that the exchange fails.

It is important to remember that although increasing w and λ will increase the size of the hash list and processing by both Alice and Bob, it does not increase the size of the data being stored. However, the choice of λ is limited by the data size, since the size of the pieces cannot be too small. Alice and Bob will have to carefully consider the overhead of creating the pieces, their keys, and the amount of data transferred when choosing parameters.

7 Related Work

Distributed storage networks already exist. Even in the decentralized cryptocurrency world, examples include [22], [40], and [6], see Section 2.2 for a description of this topic. In addition, decentralized distributed storage solutions were proposed in [8, 21]. However, this paper focuses specifically on data retrieval.

A proof-of-retrievability (POR) is a protocol in which a server shows with high probability that a client's data is accessible. The first notions of a POR appeared in [24, 29]. However, the first formal definition of a POR was given in [19]. A simple POR would be for the server to return the data to the client. However, this is expensive and can be done more cheaply. For a survey of POR methods see [43]. A common ingredient in various POR methods is the use of error-correcting codes. We also use error-correcting codes in our work. However, the difference between a POR and our work is that the POR proves the server still has access to the data and the data can be returned, while we focus on returning the data to Alice in a fair exchange.

One solution to the fair exchange problem uses a zero knowledge proof that the correct data has been returned in exchange for cryptocurrency, as in [15]. This method proves with near certainty that neither party will be cheated, and would allow hosts to exchange Alice's data without her participation. It is also possible for almost all of the computation to occur off-chain, making it financially inexpensive. However, the amount of computational work that goes into proving and verifying correctness of the document being returned is high.

Another solution to the fair exchange problem is FairSwap [8]. The authors also use a blockchain together with a smart contract as the trusted third party, and while maintaining many of the advantages of [15], this method requires considerably less work from Alice and Bob. FairSwap addresses a more general problem involving evaluation of a general circuit, but gives as an example the data retrieval application we discuss here, which allows some simplification. We include a brief description of that special case, since it simplifies the presentation and allows for a better comparison.

Briefly, before storing data, Alice constructs a Merkle tree (see [27] for details on Merkle trees) of the data, cut into n pieces in a predetermined way, which we call tree A. She places the root hash r_A on the contract. When Alice wants to retrieve the data, Bob sends the data pieces that Alice wants to her, but symmetrically encrypted. He also sends her some auxiliary symmetrically encrypted data. The encrypted data pieces and auxiliary data form leaves of a Merkle tree B whose root hash r_B Bob places on the contract, along with a hash of the key. Alice builds the tree B and checks it against r_B on the contract. If satisfied, she pays the contract in an account with a time-lock before receiving the key to decrypt. Then, Bob puts the key on the contract. If Alice is satisfied with the decrypted data, she does nothing, and the tokens are released to Bob at the appointed time. However, if Alice finds fault with the data, she must present a proof of an error to the contract in order to reclaim her tokens before the time-lock releases them to Bob.

Assume that there is a fault in the data. The first portion of the decrypted data is supposed to be the data that Alice wants. Alice uses these decrypted pieces to construct a Merkle tree A'. If Bob had carried all of this out correctly, then A' would be the same as A, but we are assuming that this is not the case. Bob was supposed to construct exactly this same Merkle tree, keep track of all of the hash outputs at each node, and encrypt them. These are the auxiliary data that form the rest of the leaves of tree B with root r_B.

Alice can trace through A', starting at the leaves, and compare each output value with those in B until she finds the first mistake. The inputs and output of this mistaken hash calculation are leaves in tree B. Their branches in B form a proof of Bob's error. The contract traces through tree B to determine that the hash inputs and output are what Bob claims to have calculated, and then computes the same hash to compare. Assuming that Bob did indeed make a mistake, the tokens are released to Alice, and otherwise they are released to Bob.

FairSwap only uses Merkle tree proofs in the circumstance where Alice has caught Bob cheating and needs to prove it to the contract. Otherwise, no proofs are required. There is essentially no opportunity for Alice or Bob to cheat, assuming that all encryption, etc. is secure. While computationally less expensive to both Alice and Bob than the zero knowledge proof method, the contract's work to verify the proof grows logarithmically in the size of the data in the case where Bob cheats. For small data sizes, this computation is not significant, but eventually it would become prohibitive. In addition, FairSwap is vulnerable to a Distributed Denial of Service attack. There is a brief paragraph about using collateral to mitigate such an attack in [8], but the actual collateral is not calculated.

The idea of using smart contracts for a collateral mechanism has been around for a while. In [42], the authors recognized that smart contracts allow a collateral mechanism to make it expensive to those that violate the contract. The use of collateral and a smart contracts on a blockchain to exchange goods can be found in [20, 7, 21, 17] and many others.

8 Conclusion

In this paper, we presented a data storage protocol that Alice and Bob can use to facilitate a storage agreement between them. The protocol can be used on an arbitrary distributed ledger that supports smart contracts. Both parties' pseudonymity is not violated, unless simple use of a blockchain does so. A main feature of the protocol is a collateral mechanism that collectively punishes both Alice and Bob if either of them abandon the protocol or cheat. However, we prove that parameters can be chosen such that the collateral can be made arbitrarily low, up to data restrictions, and still result in an expected loss if either Alice or Bob cheat. We are able to achieve this due to our non-standard use of error-correcting codes.

Another important feature is Alice's ability to verify Bob's adherence to the protocol using a combination of Elgamal encryption, symmetric key encryption, and a hash list, while only gaining limited knowledge of her data before paying the full price for it. She verifies that the data is authentic, as is the public key, and therefore acquiring the matching private key is equivalent to recovering her data, all while protecting anonymity.

This retrieval process can also be applied to the sale of any digital media, as long as the root hash can be obtained from a trusted source. Either the parameters need to be included as well, or else there needs to be a standardized way to produce the pieces (for example with fixed parameters, or values given as a function of data size). Here, we require Alice posts the parameters on the DApp. She is also both the purchaser and the person who generates the root hash, so no trusted third party is needed for that role.

References

[1] IEEE standard specifications for public-key cryptography - Amendment 1: Additional techniques. *IEEE Std 1363a-2004 (Amendment to IEEE Std 1363-2000)*, pages 1–167, September 2004.

[2] Anonymous. Eth gas station, 2018.

[3] Jordi Baylina. ecsol. `https://github.com/jbaylina/ecsol`, 2018.

[4] block.one. Eos dawn 2.0, 2018.

[5] CoinMarketCap. Cryptocurrency market capitalizations, 2018.

[6] Luke Champine David Vorick. Sia: Simple decentralized storage [whitepaper]. Technical report, Nebulous Inc., November 2014.

[7] S. Dekorte.

[8] Stefan Dziembowski, Lisa Eckey, and Sebastian Faust. Fairswap: How to fairly exchange digital goods. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 967–984, New York, NY, USA, 2018. ACM.

[9] Taher Elgamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE transactions on information theory*, 31(4):469–472, 1985.

[10] Cardano Foundation. Iohk|cardano blockchain technology | cryptocurrency softwae ... and beyond, 2018.

[11] Ethereum Foundation. Ethereum, 2017.

[12] Ethereum Foundation. Ethereum classic, 2018.

[13] ICON Foundation. icon hyperconnect the world, 2017.

[14] Tron Foundation. Tron, 2018.

[15] Gmaxwell. Zero Knowledge Contingent Payment, November 2011.

[16] Werner Güth, Rolf Schmittberger, and Bernd Schwarze. An experimental analysis of ultimatum bargaining. *Journal of Economic Behavior & Organization*, 3(4):367 – 388, 1982.

[17] H. R. Hasan and K. Salah. Proof of delivery of digital assets using blockchain and smart contracts. *IEEE Access*, 6:65439–65448, 2018.

[18] Richard Holden and Anup Malani. Can blockchain solve the holdup problem in contracts? *University of Chicago Coase-Sandor Institute for Law & Economics Research Paper*, (846), December 2017.

[19] Ari Juels and Burton S. Kaliski, Jr. PORs: Proofs of retrievability for large files. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 584–597, New York, NY, USA, 2007. ACM.

[20] Ido Kaiser. A decentralized private marketplace: DRAFT 0.1.

[21] Henning Kopp, David Mödinger, Franz Hauck, Frank Kargl, and Christoph Bösch. Design of a privacy-preserving decentralized file storage with financial incentives. In *2017 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 14–22. IEEE, 2017.

[22] Protocol Labs. Filecoin: A decentralized storage network [whitepaper], August 2017.

[23] Micah Lee. OnionShare, March 2014.

[24] Mark Lillibridge, Sameh Elnikety, , and Michael and Burrows. A cooperative internet backup scheme. In *Proceedings of the 2003 Usenix Annual Technical Conference*, pages 29–41. USENIX, June 2003.

[25] Lisk. Access the power of blockchain, 2018.

[26] Florence Jessie MacWilliams and Neil James Alexander Sloane. *The theory of error-correcting codes*. Elsevier, 1977.

[27] Ralph C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *Advances in Cryptology — CRYPTO '87*, pages 369–378, Berlin, Heidelberg, 1988. Springer Berlin Heidelberg.

[28] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *URL: http://www. bitcoin. org/bitcoin. pdf*, 2008.

[29] Moni Naor and Guy N. Rothblum. The complexity of online memory checking. *J. ACM*, 56(1):2:1–2:46, February 2009.

[30] National Institute of Standards and Technology (NIST). Advanced Encryption Standard (AES). Technical Report Federal Information Processing Standards (FIPS) Publication 197, U.S. Department of Commerce, Washington, D.C., November 2001.

[31] National Institute of Standards and Technology (NIST). Recommendation for Key Derivation Using Pseudorandom Functions (Revised). Technical Report NIST Special Publication (SP) 800-108, U.S. Department of Commerce, Washington, D.C., October 2009.

[32] National Institute of Standards and Technology (NIST). Digital Signature Standard (DSS). Technical Report Federal Information Processing Standards (FIPS) Publication 186-4, U.S. Department of Commerce, Washington, D.C., July 2013.

[33] National Institute of Standards and Technology (NIST). Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography. Technical Report NIST Special Publication (SP) 800-56A REV. 3, U.S. Department of Commerce, Washington, D.C., May 2013.

[34] National Institute of Standards and Technology (NIST). Secure Hash Standard (SHS). Technical Report Federal Information Processing Standards (FIPS) Publication 180-4, U.S. Department of Commerce, Washington, D.C., August 2015.

[35] National Institute of Standards and Technology (NIST). Blockchain Technology Overview. Technical Report NIST Internal Report (IR) NISTIR 8202, U.S. Department of Commerce, Washington, D.C., January 2018.

[36] NEM.io Foundation Ltd. NEM the smart asset blockchain, 2018.

[37] Henning Pagnia and Felix C Gärtner. On the impossibility of fair exchange without a trusted third party. Technical report, Technical Report TUD-BS-1999-02, Darmstadt University of Technology, Department of Computer Science, Darmstadt, Germany, 1999.

[38] James S Plank. Erasure codes for storage systems: A brief primer. *The Usenix Magazine*, 38(6):44–50, 2013.

[39] Qtum Foundation. Qtum the blockchain made ready for business, 2018.

[40] Josh Brandoff James Prestwich Gordon Hall Patrick Gerbes Philip Hutchins Chris Pollard Shawn Wilkinson, Tome Boshevski. Storj a peer-to-peer cloud storage network [whitepaper]. Technical report, December 2016.

[41] Victor Shoup. A proposal for an iso standard for public key encryption (version 2.1). *IACR E-Print Archive*, 112, 2001.

[42] Nick Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2(9), 1997.

[43] Choon Beng Tan, Mohd Hanafi Ahmad Hijazi, Yuto Lim, and Abdullah Gani. A survey on proof of retrievability for cloud data integrity and availability: Cloud storage state-of-the-art, issues, solutions and future trends. *Journal of Network and Computer Applications*, 110:75 – 86, 2018.

[44] NEO Team. Neo white paper, a distributed network for the smart economy, 2018.

[45] Stephen B Wicker. *Error control systems for digital communication and storage*, volume 1, chapter 16, pages 245–246. Prentice hall Englewood Cliffs, 1995.

A Implementation

Table 3: IDMS Smart Contract Function Gas Requirements

Function	**Permitted Role**	**Gas**	**Ether**	**USD**
Contract Deployment	Marketplace DApp	1 530 398	4.6×10^{-3}	\$1.93
Submit Public Elgamal Key	Bob	65 760	2.0×10^{-4}	\$0.08
Provide Interaction Location	Alice	109 082	3.30×10^{-4}	\$0.14
Request Block	Alice	49 605	1.5×10^{-4}	\$0.06
Request Private Elgamal Key	Alice	49 723	1.5×10^{-4}	\$0.06
Submit Private Elgamal Key	Bob	4 001 950 (estimated)	1.2×10^{-2}	\$5.03
Withdraw Payments	Bob	29 462	8.8×10^{-5}	\$0.04

We implemented our smart contract approach using the Ethereum cryptocurrency platform and using its Solidity smart contract development language. To check that Bob's private Elgamal key matches his previously announced public key, we leverage the Solidity code within the ecsol github repository [3]. We used the Ethereum officially supported Remix integrated development environment to both code and execute the contract.

In Ethereum, all transactions are executed simultaneously in parallel by all nodes competing to mine the next Ethereum block. Thus, transactions are expensive to process on the Ethereum network; they require payment to the miner that publishes the block in which the transaction is processed. The submitter of a transaction must pay a quantity of 'gas' which is calculated by the number and types of operations that the transaction will perform. This in turn is defined by the contract functions that will be called and the code executed within those functions. There is a ratio maintained by the Ethereum network that defines how to convert from gas to Ether, since gas payments are made using the Ethereum cryptocurrency, Ether. In Table 3, we provide the payment required (in gas, Ether, and U.S. dollars) for each transaction type.

Most calls to the contract are not prohibitively costly (less than \$0.14 USD). However, the deployment of the contract cost \$1.93 USD. What was surprisingly expensive was the submission of Bob's private Elgamal key. This costs up to an estimated \$5.03 due to the need for the contract to check to make sure that Bob's private key matches his previously posted public key (this estimate is based on measurement for this published by the ecsol repository). We used small private keys in our testing on Remix because Remix has a memory leak that makes large computations infeasible. We could have deployed our contract on an actual Ethereum network but ecsol had already taken the needed gas measurement using the Ethereum Ropsten test network.

As can be seen in Table 3, transactions in Ethereum can be expensive. This is largely due to the enormous increase in the price of Ether due to speculative activities. From 2017-03-02 to 2018-04-11, the price of Ether has increase from \$18.68 to \$420.94 per Ether [5]. Fortunately, the gas price (Ether to gas ratio) has decreased significantly to account for the increase in Ether price. The gas price decreased from 2.74E-8 Ether/gas to 0.30E-8 Ether/gas from 2017-03-01 to

2018-04-11[4] [2]. However, the gas price ratio decreased only 89 % in the same time period that Ether experienced a 2 253 % rise in USD fiat currency value, resulting in a fiat currency transaction cost increase of 247 %.

Despite the high transaction prices, we used Ethereum for our proof-of-concept implementation because it is well known, has a relatively mature development environment, and was the first widely used smart contract cryptocurrency. It also may get cheaper to use as Ethereum migrates from a Proof-of-Work consensus algorithm to one based on Proof-of-Stake (where the right to publish a block is related to the amount of coin possessed). Also, many additional smart contract cryptocurrencies exist or are under development that may create enough competition to lower smart contract transaction fees (e.g., Cardano [10], NEO [44], NEM [36], EOS [4], Tron [14], Ethereum Classic [12], Qtum [39], ICON [13], and Lisk [25])[5].

[4]Users may actually choose the amount of gas they supply to a transaction (and thus set the gas price). If they supply little gas relative to the overall market, their transaction will be delayed. The market price is highly variable.

[5]The smart contract cryptocurrencies listed are those with the largest market capitalization in decending order according to CoinMarketCap [5], as of 2018-01-17