AppeCoin

Practical Anonymous Peer-to-Peer e-Cash System

Sergio Demian Lerner:

First Revision: 07-2011
Revision 0.28: 04-2014

PRELIMINARY

1 Abstract

We present an efficient divisible e-cash scheme based on strong cryptographic assumptions. This scheme can be implemented on a peer-to-peer network without the requirement of a Trusted Third Party. Also the scheme can be implemented on a network with a known set of central nodes that require minimal trust from the users. The scheme can be implemented on a variety of cryptographic ciphers, over Zp and Elliptic curves, and other fields where the Diffie-Hellman problem is known to be hard. Instead of using monetary units of fixed value ("coins"), our scheme relies on electronic bills of arbitrary value. Each bill can be subdivided into bills of smaller amounts at will without disclosing the bill amount, and also bills can be added creating new bills of greater amounts. The size of each bill is proportional to the size of an element of the field in use (Zp, or a point in an EC). The computational and communication complexity of the protocol is proportional to the size of the bill and the number of transactions per second the system processes.

Keywords: digital cash, divisibility, unlinkability, annonymity, peer-to-peer.

Index

1 Introduction

In this paper we present AppeCoin, a practical and efficient e-Cash scheme based on strong cryptographic assumptions. AppeCoin has a number of outstanding properties:

- Efficient
- Security based on strong cryptography
- Truly Anonymous: Untraceable and unlinkable bills, private account balances, private transaction amounts.

A peer-to-peer currency system is a distributed system where computers (called peers) share a portion of their resources in order to verify transactions and issue payments, without a central authority and any Trusted Third Party (TTP). Some peer-to-peer currencies, such as Bitcoin and derived coins are not inherently TTP-free, but they behave as TTP-free as long as the majority of

the network computing power is honest. In Bitcoin transactions are accepted when they are included in a proof-of-work block chain. AppeCoin requires that the network decides an unique ordering of transactions, so it can be implemented using a proof-of-work block chain as in Bitcoin, by consensus or by the trust in a central node.

In AppeCoin money is represented by encrypted bills of different amounts. The amount of money contained by an encrypted bill is only known to the bill owner. The bill owner has a secrets keys that allow him to open the encrypted bill, combine it, subdivide it, or transfer it to another user. AppeCoin bills behave as banknotes but have an embedded public key that belongs to the bill's owner. Even each bill has a "serial number", this number is hidden as the bill is transferred. In the proposed implementation, each bill consist of 6 fields, but there also an alternate implementation will only 4 fields. Nevertheless using 4 fields requires that a complex proof of knowledge to transfer the bill, while using 6 fields a bill is transferred with a single digital signature.
We define a payment as a transaction where a certain amount of money is transferred from a user to another user.

How AppeCoin achieve anonymization

AppeCoin money is contained in bills. These bills are accepted by the network as long as they are active or minted. Active bills are stored in a database which generally is either replicated among peers or peers have access to it. Each bill has a temporally unique id to allow referencing active bills. Bills are anonymized by shuffles. A shuffle is an operation where a set of bills is mixed together and re-encrypted such that, given the input and the final output, it's infeasible for any party to recover any information regarding the applied permutation nor any information regarding the encryption key used, with any meaningful advantage better than random guessing, with the exception of the mixer and the owners of the bills mixed. The shuffler knows both the permutation and the encryption key and each bill owner knows the destination position in the output of his own bill, but not the key, neither the positions of the unowned bills. There are three types of mixes, private, delegated and public. Public mixes periodically and automatically mix the bills of random owners. In private mixes a single owner mixes some of his own bills with a set of bills randomly taken from the database of active bills. In delegated mixes, a third party is in charge of collecting a set of bills specified by different owners and mix them together, possibly also adding a random set of bills in the database of active bill. In all cases the mixer knows the permutation, and each bill owner is only able to detect what one of his bills has been mixed and which is the identification of a new bill that replaces the mixed bill (this is equivalent to detect bill position in the list of permuted bills). Because a mixer can shuffle bills he doesn't own, an re-encryption system that preserves the the the possibility that the bill owner proves ownership of the bill and decrypts it. A cryptosystem with this property is said to allow Universal Re-encryption. Each bill is associated with a public key (and therefore also with a private key). To transfer a bill, the previous association with its current public key is broken and a new association is made. This bill public key can be interpreted as a destination address. So the universal re-encryption cryptosystem must have the additional property that the public key associated with the message is maintained intact after re-encryption.

Privacy issues

Suppose that each bill represents a fixed monetary amount. The first problem one can find in an cryptocurrency scheme is that if the public key is used multiple times associated to different bills, information regarding the number of payments made, or even the amounts transferred to that public key (a destination address) is leaked. To maximize anonymity, each bill should be sent to a different unique address specified by the receiver or computed by the sender in relation to a fixed destination

address. But still there is a second problem: if each bill holds a fixed amount of money, a transaction may require the combination of several bills to cover a desired amount. To pay in specific amounts, most payment schemes, like Bitcoin, implement transactions that specify a set of input "bills" and a set of output destinations. But this method also leaks information, since the grouping of "bills" may suggest a single receiver and the number of input bills contained in the transaction may correlate with the amount of money transferred. We conclude that each bill should go to a different address of the payee in a different transaction. But this modified method leaks information too. Since these related transactions are probably created in a certain short period of time and by the same node of the network, other nodes connected to this peer may be able to infer that these transactions are part of a single payment. Also sending each bill in a different transaction may result in increased fees. It's clear that to achieve truly anonymization, each payment should be contained in a single transaction whose size, or any other detectable transaction field, is not correlated with the payment amount.
Even if a system like the one described could provide some anonymity, it has two other problems: continuous fragmentation and difficulty to pay an amount lower than the value of a bill. The last problem arises for the impossibility to specify anonymously the amount of "change" in a transaction or to anonymously get "change" from another peer such that a transaction can specify and exact amount of bills.

A full solution: AppeCoin, private divisibility and combinability

Bills are periodically anonymized by shuffles. To maximize anonymity, a standard payment should consists of a single bill, and this bill amount should match the exact amount of the payment. The transaction that contains the payment also specifies the fees as a single bill of the exact amount to be paid. To solve the problem of continuous fragmentation and specifying change, in AppeCoin bills can be anonymously divided and combined. Divisibility is required to solve the problem of change, and combinability is required to reduce fragmentation. Divisibility and combinability can be implemented for an arbitrary number of bills. Nevertheless in this paper we'll explain how to combine two bills into one or divide a bill in two for clarity. It's also possible to define protocols similar to the shown in this paper to transform any input set of bill into an output set of bills, thus removing the distinction between combination and division. Nevertheless, we'll restrict to the 1:n and n:1 case for clarity.

To build a standard payment the payor must first create a bill for the transaction fees (if any) and then a bill for the payment itself. To create each of this bills, the the following steps must be followed:

- If the bill amount is higher than any owned bill amount, combine a set of bills into a single bill of an amount higher or equal the amount to be paid. This requires a combination transaction.
- If there is no bill with the exact amount to be paid, divide a bill into two bills, where one of them has exactly the amount to be paid. This requires a division transaction. The combination and division transactions can be grouped in a single transaction without loss of anonymity.

After both the payment bill and the fee bill are ready, the following steps should be carried:

- Wait some time until a periodic public shuffle mixes the newly created bills or build a private shuffle transaction or ask for a delegated shuffle.
- Construct the payment transaction using the shuffled bills.

Public Mixing Service (PMS)

The public mixing service is a shuffling service provided by the system “free of charge”. The idea behind providing such service is to establish a base anonymity layer so users cannot be target of discrimination by governments because of using or not anonymization features. If the system is implemented in a peer-to-peer network using a block-chain (such as Bitcoin) then the public mixing service should be provided by the miners. Obviously a miner may store the information regarding the applied permutation, and provide that information to whatever government department, competitor or underground party he wishes. Nevertheless as long as there are “fair” miner with enough mining power the system will provide free anonymization of all bills sooner or later.
There are several options to choose which bills should be shuffled together. One method will be referred as **Fast-start**. With Fast-start, all of the bills created in the current block are always shuffled. Also a percentage of the bills that were the output of the previous block public shuffle (or the output of any shuffle that was performed in the previous block) is also re-shuffled. It's best that the bills that are taken from the previous shuffles outputs be pseudo-randomly chosen in a way that is hard for the miner to force a certain outcome. For example, the pseudo-random order can be dictated by taking as seed the block header hash digests of the ten previous blocks.

A simple fast-start strategy can be defined by three values p;c;d. Where p is the maximum number of bills to be taken from the previous block, c is the maximum number of bills to be taken from the current block and d is the maximum number of bills to be taken from the active bills database.
For example, 100;1000;100 is a strategy to take at most 100 bills from the previous block, at most 1000 bills from the current block and at most 100 bills from the database.
Also the strategy specification can be given in percentages, where P means the total number of bills in the previous block, C is the total number of bills in the current block and D is the total number of bills in the database.
The strategy 50P;100C;50C means that 50% of the previous bills are remixed with 100% of the current bills and half the number of bills in the current block taken from the database.
Other more elaborated strategies can be analyzed such as x=50P;y=100C;max(10,1000-x-y) which means that the number of bills to be taken from the database is chosen so the total number of mixed bills is limited but still always some database bills are mixed.

Other important aspect to analyze is whether the bills to be mixed in a block should be easily predicted or not. One method allow owners to predict when their bills in the database will be chosen to be part of a public shuffle is by making the pseudo-random selection function depend solely on the block height. If the percentage of bills taken from the database in each public shuffle is low, this allows client applications to reduce the workload of checking every mixed bill against their public keys (which should not be more than one, as opposed to Bitcoin privacy standards). Nevertheless there is still the possibility that a delegated or private shuffle mixes one of the owned bills at random. It's possible to force private and delegated shuffles to only mix public bills that belong to the same set of bills that will be public shuffled. To implement this, for each private bill that undergoes a private/delegated shuffle a signature of the bill id and the block height, using the bill public key, must be provided in the shuffling transaction.

If which bills are shuffled cannot be predicted, then the client application should decide, based on the usage pattern of the owned bills, if it's better for it to look for each owed bill after each shuffle or delay this verification. To explain a good delayed checking strategy, we'll say that a bill id is “active” if it's on the current active bill database, “inactive” if it was in the database but was re-encrypted, so there is still a bill in the active database associated with the inactive bill, and “destroyed” if the bill was destroyed because of a division or a payment. The client application may

delay any search for changed ids until a payment is required to be performed. If there there is no enough money in the active bill ids, then a search is performed starting from the last unsearched block, replacing inactive bills by their active counterparts until the required amount in active bills has been reached.

Delegated Mixing Service (DMS)

A delegated mixing service is a service provided by a third party that collects bills from other interested parties, adds some active bills from the database and mix them together. Users may choose delegated mixing services for several reasons:

1. They may trust the service owner more than any anonymous miner
2. They may want anonymization to occur faster than the public shuffling service may provide.

It must be noted that the DMS owner does not have any power to either steal nor forbid use of the bills it shuffles.
Since the transaction that contains the mixed bills may need to pay fees, the DMS will probably charge users willing to mix their bills. The payment can be achieved by a micro-payment method or by transferring bills as payment.

It must be noted that if both a shuffle and a bill transfer of the same bill occur in the same block, the transfer should be executed first, to prevent a denial of service attack by re-mixing a not owned bill. Nevertheless this also means that the network must be able to prioritize transactions queued to be included in block. Bill transfers must have higher priority than bill mixes.

Private Mixing (PRM)

Private mixing is a transaction that involves mixing a set of bills owned by the same person. From the perspective of the system, there is no different between a delegated mixing or a private mixing, and both transaction looks alike. Nevertheless a user my desire to be his own mixer to has the highest possible anonymity.

Pairwise Mixing (PAM)

As with the Bitcoin CoinJoin protocol, one can implement pairwise mixing and leverage the cost of the mixing transaction. If two peers decide they are willing to mix bills together, they build a transaction consisting on the bills to be mixed, where each input bill is signed by the corresponding parties, and two fee bills are included (one send by each party).

The performance of Mixing

One of the most important aspects of mixing cryptocurrencies is mixing requires all nodes to verify the correctness of the mix. This is done with zero-knowledge proofs. Verification performance is therefore is out utmost importance. In this paper we present two mixing (shuffling) protocols. The verification time is generally measured in the number of slow operations performed, usually modular exponentiations over a field of high order. The first is a standard cut-and-choose protocol, provided for simplicity and performance comparison. The second is a protocol whose number of modular exponentiations does not depend on the number of elements mixed, which is what makes mixing in Appecoin practical

2 Requirements

We now briefly describe the conditions that we think an unlinkable divisible digital cash scheme should satisfy. The list (based on [16] and [x]) is as follows:

Regarding to Security

No double-spending. All double-spending attempts will be detected and only one of the transactions will take place. The remaining transactions will be discarded.

No over-spending. No user can spend more than his bills monetary value nor any user can create a valid bill from thin air.

Unforgeability. Neither a bill nor a transcript of a payment of an honest user can be forged.

Portability. The security and use of digital cash is not dependent on any physical location.

Authenticity. A bill can be proved to be authentic, without the need to spend it.

Non-repudiation. No user can repudiate having spent a bill.

Regarding to Privacy

Anonymity. The payer that conforms to the protocol cannot be identified from the exchanged information during payment.

Unlinkability. It's infeasible to link any two payments executed by the same user (even without learning the payer's identity).

Untraceability. It should be possible to hide the past transactions of a bill or the number of transactions that the bill underwent.

Hidden amounts. It should be possible to hide the amounts of a transaction to prevent loss of privacy because of correlated amounts.

Regarding to Utility

Transferability. Bills can be transferred between the users without intervention of a central authority.

Divisibility. A user can subdivide a bill into bills of smaller amounts that add up to the original bill without disclosing the amounts processed.

Composability. A user can create a new bill by adding other bills without disclosing the amounts processed

3. General Definitions

Def: An HCGC (Homomorphic Commutative Group Cipher) is a tuple (E,D,*K*,*M*,*) that satisfies:

- *M* the message space (both plaintext and ciphertexts space are equal)
- *K* is the key space.
- *K* is a field.
- E is the encryption function of a symmetric cipher
- D be the decryption function of a symmetric cipher
- For all m in M and k_1,k_2 in K, $E(k_1*k_2,m) = E(k_1,E(k_2,m))$
- For all m in M and k in K, $D(k,E(k,m))=m$
- For all m in M and k in K, $D(k,m) = E(k^{-1},m)$
- E and D share an homomorphic operator ¨*¨ such that for any a,b in M and k in K, $E(k,a*b) = E(k,a) * E(k,b)$.
- E and D share an homomorphic operator ¨*¨ such that for any a in M and k_1,k_2 in K, $E(k_1+k_2,a) = E(k_1,a) * E(k_2,a)$.
- The operations "-" and "+" are on K are the ones defined for field.
- E has no weak or invalid plaintexts except a small set with negligible probability of random occurrence that can be easily detected. Because of this, almost every element m in M is a generator of any other element of M by the encryption function E, thus we can safely assume all elements are generators.
- E has no weak or invalid keys except a small set with negligible probability of random occurrence that can be easily detected.

Notation

- "*" stands for multiplication on the field K or the homomorphic operator of E, depending on the context it is used.

The HGCC is chosen so that the probability that $E(k,a)=1$ for a random a in M and k in K, where 1 is the multiplicative identity is negligible. The same applies for the identity key. Nevertheless we'll check that no procedure computes unexpectedly the identity and if computed, the running procedure will be aborted.

Def: The **Discrete Log Problem** (DLP)

Let a,b be elements of M. The Discrete Log Problem is to find k in K such that $a = E(k,b)$.

Solving the DL problem for any pair of elements is equivalent to being able to perform a known-plaintext-attack (KPA) on the cipher.
If the DLP is hard, we can say that two specific elements a,b are **independent** if is computationally infeasible for any user to find k in K such that $a = E(k,b)$.

Def: The **Representation Problem** (RP)

Let $x_1,..,x_n$ and a be elements of M. The representation problem (RP) is to find $k_1,..,k_n$ such that $a = E(k_1,p_1) * \ldots * E(k_n,p_n)$.

Def: The **Diffie-Hellman Problem** (DHP)

Let a be an element of M. The Diffie-Hellman problem (DHP) is to find $E(k_1*k_2,a)$ given $E(k_1,a)$ and $E(k_2,a)$. Since $E(k_1*k_2,a) = E(k_1,E(k_2,a))$ solving the DH problem for any elements a, k_1 and k_2 is equivalent to perform a malleability attack on the cipher.

For Appecoin we need a cipher HCGC such that the DLP, DH and RP problems are hard. Also we'll require some standard assumptions about the cipher: resistance against Ciphertext-only attack (COA) and Chosen-plaintext attack (CPA). All our computations will be based on an HCGC, and there are various examples of ready to use HGCCs:

- Pohlig-Hellman symmetric cipher on a Schnorr group (the subgroup of kth residues modulo a prime p, where $(p - 1) / k$ is also a large prime)
- Massey-Omura cryptosystem
- Pohlig-Hellman symmetric cipher on elliptic curves
- Pohlig-Hellman symmetric cipher analog on any other field
- The LUC cryptosystem

So simplify the forthcoming descriptions, instead of using the abstract notation $E(k,x)$ we'll use the exponential notation commonly used for the Z_p field, so that $E(k,m)$ will be noted m^k (omitting mod p). This notation is helpful since is widely used in cryptography.

Abstract definition of Schnorr Signatures with Universal re-encryption

Def: A **Schnorr signature** scheme over a given HCGC is defined by these these algorithms. This definition of Schnorr signatures differs from the standard definition in the fact that there is no public generator, the generator is variable and is part of the public key to allow universal re-encryption f public keys. Also H is a cryptographic hash function.

Key Generation

- Choose a random x in K.
- Choose a generator g in M.
- Compute $y = g^x$
- (y,g) is the public key, x is the private key.

Signing: to sign the message u

- Choose a random k in K
- Let $r = g^k$
- Let $e = H(u \| r)$
- Let $s = (k - x*e)$
- (s,e) is the signature

Verifying:

- Let $r_v = g^s*y^e$
- Let $e_v = H(u \| r_v)$
- If $e = e_v$ then the signature is authentic.

Universal Re-Encryption of a Public Key

- Let (y,g) be a public key
- Choose a random k in K.
- (y^k,g^k) is the new public key for the same private key.

Proof:
$r_v = g^s * y^{e =} r_v = g^{(k - x*e)} * g^{x*e} =$ E($k - x*e, g$) * E($x*e, g$) = E ($k - x*e + x*e, g$) = $g^k = r$
and so e_v = H(u || r_v) = e_v = H(u || r) = e

The universal re-encryption property requires that the HCGC withstands **Ciphertext indistinguishability** under Choosen-plantext attack, which is provided by the Diffie-Hellman hardness assumption .
In this paper we've chosen Schnorr signatures over ElGamal or DSA (EC-Schnorr vs. ECDSA) signatures, since verifying Schnorr signatures is faster, and there are some results that prove that key-pairs used in Schnorr signatures can be reused for Diffie-Hellman public key encryption of a session key, such as in DLIES/ECIES, without degrading the overall security [128]. Nevertheless, it's possible to use DSA-like/ElGamal/ECDSA signatures over Schnorr and the properties of AppeCoin are not affected. If adverse interaction between signing and encryption keys is suspected, then AppeCoin public keys can be constructed by concatenation of a signing-specific public key and an encryption-specific public key.

3.1 Bills

An AppeCoin bill a is a tuple ($a_{s1}, a_{s2}, a_t, a_u, a_v, a_w$) that consists of six fields:

1.a_{s1} and a_{s2} represent a destination address, and hides the private value s
2.a_v represents the bill value, and hides the private value v
3.a_t represents the first DH masking of the bills value, and hides the private value t
4.a_u represents the unit of value, and hides the private value u
5.a_w represents the second DH masking of the bills value, and hides the private value w

The system is defined by the HCGC and three generators of the message space g, a and b, where g, a and b are pairwise independent.

The destination is represented is by the tuple (a_{s1}, a_{s2}) = (g', g'^s), where s is a private key known only to the last receiver of the bill that represents the destination address and g' is a random generator of the message space, chosen by the last sender of the bill.
The bill value v is encoded in a the "exponent" (the encryption key of the HCGC) of a generator, masked by two other exponent values: $a_v = a_u{}^{twv}$, where $a_t = a_u{}^t$ and $a_w = a_u{}^w$, and t,v,w are secret values. This masking which result in the encryption of the value v, such as finding v is equivalent to breaking Diffie-Hellman in the HCGC. The unit of value (u) is a field that is used to track the changes on the a_v field when the bill is encrypted: it's a witness of all encryptions that a bill undergo. The bill value is the monetary amount transferred and this value is blinded by a Diffie-Hellman exponent.
The owner of a bill must keep track of the secret tuple (s,t,u,v,w) for each bill. All secret values correspond exactly to the discrete logarithms of the corresponding terms to the base a_u. When the bill is created, then $a_u = d$. After the bill undergoes encryptions, this will not normally hold.

Now we'll define procedures that operate on bills:

•r :K ← **GenRandKey()** is a random/pseudorandom key generation function.

•r : *M* ← **GenRandMsg()** is a random/pseudorandom message generation function.

•P : Permutation ← **GenRandPerm()** is a random/pseudorandom permutation generation function from the non-negative integers to the non-negative integers.

•*b* : Bill ← **TransferBill**(*d* :K, *a* : Bill) is a Bill that has been transferred to the destination address *d*.

•*b* : Bill ← **EncryptBill**(*k* :K, a : Bill) is a Bill encryption function with the key k.

•U^*: List<Bill> ← **ShuffleBills**(C^* : List<Bill>, *k* : K, *P* :Permutation) is a shuffle function and re-encryption with key *k* that transforms the bills $c_1, \ldots, c_n$ into the bills $u_1, \ldots, u_n$, where $U^* = (u_1, \ldots, u_n)$ and $C^* = (c_1, \ldots, c_n)$, using the permutation P.

•*p* : SProof ← **GetBillShuffleProof**(A^*,B^* : List<Bill>, *k* : K) is a function that returns a practical non-interactive zero-knowledge proof that the Shuffle operation has been done correctly. There are two versions of this function, one uses the classcal cut-and-choose method and the other uses a new method in which the number of modular exponentiations does not depend on the number of elements shuffled.

•*b* :Boolean ← **VerifyBillShuffleProof**(A^*,B^* : List<Bill>, *p* : SProof) is a function that verifies a zero-knowledge proof *p* and returns true if it is correct, and false otherwise.

• (*s,e*) : M← **Sign**(*s* : K, s_1,s_2 : M, *u* :Message) is function that returns a signature of the message *u* using the public key (s_1,s_2) and the private key *s* of URe-Schnorr signatures.

•*b*: Boolean ← **VerifySig**(*s,e*: M, s_1,s_2 : M , *u* :Message) is the verification function for the Ure-Schnorr signature (*s*,*e*), for the public key (s_1,s_2) and the message *u*.

Notation: (*a*,*b*) : M means that both a and b are fields of type M.

•(*y*,*g*) : M , *x* : K ← **GenPubPrivKey()** is a function that generates a new public and private key-pair: *y* and *g* are public, and *x* is private.

• (*y'*,*g'*) : M ← **UrePubKey(** *y*,*g* : M) is a function that returns a re-encryption of the public address (*y*,*g*) with a new random key.

GenRandKey
Output: k : K
1.Choose a random *k* in K. 2.Output *k*.

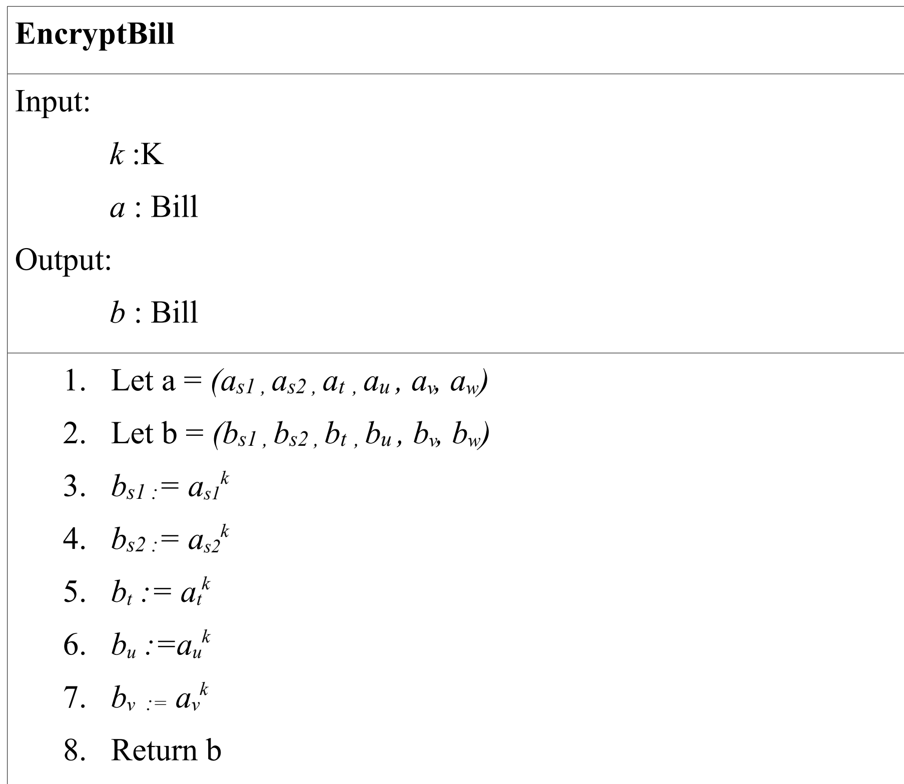
Figure 1

GenRandMsg
Output: m : M
1.Choose a random generator *m* in M. 2.Output *m*.

Figure 2

TransferBill
Input: d :K a : Bill Output: b : Bill
1. Let a = $(a_{s1}, a_{s2}, a_t, a_u, a_v, a_w)$ 2. Let b = $(b_{s1}, b_{s2}, b_t, b_u, b_v, b_w)$ 3. Let m := GetRandMsg() 4. b_{s2} :=m 5. b_{s1} := m^d 6. b_t := a_t 7. b_u :=a_u 8. b_v := a_v 9. b_{vw} := a_w 10. return b

Figure 3



EncryptBill

Input:

k :K

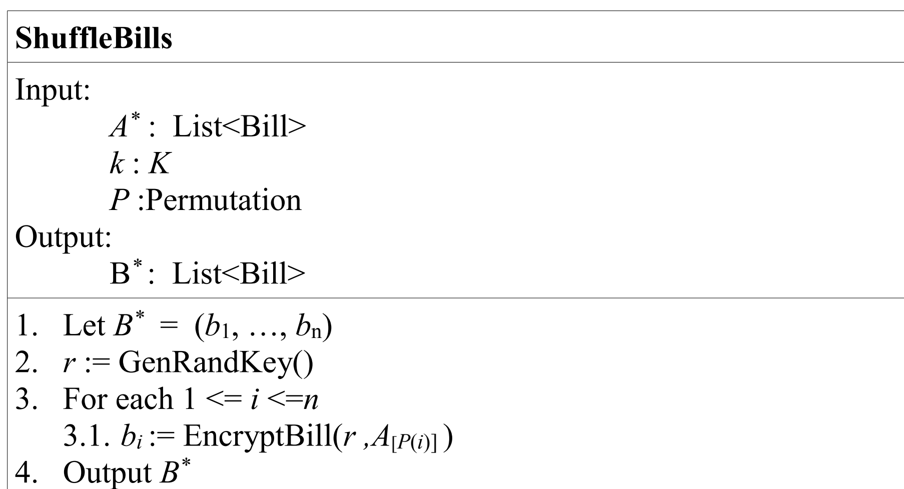
a : Bill

Output:

b : Bill

1. Let a = $(a_{s1}, a_{s2}, a_t, a_u, a_v, a_w)$
2. Let b = $(b_{s1}, b_{s2}, b_t, b_u, b_v, b_w)$
3. $b_{s1} := a_{s1}^k$
4. $b_{s2} := a_{s2}^k$
5. $b_t := a_t^k$
6. $b_u := a_u^k$
7. $b_v := a_v^k$
8. Return b

Figure 4



ShuffleBills

Input:

A^* : List<Bill>

k : K

P :Permutation

Output:

B^*: List<Bill>

1. Let $B^* = (b_1, \ldots, b_n)$
2. r := GenRandKey()
3. For each $1 <= i <= n$
   3.1. b_i := EncryptBill(r ,$A_{[P(i)]}$)
4. Output B^*

Figure 5

GetBillShuffleProof_CutAndChoose (cut and choose)

Input:

A^*,B^* : List<Bill>

k : K

P :Permutation

Output:

C :Blob

1. Let $A^* = (a_1, \ldots, a_n)$
2. Let $B^* = (b_1, \ldots, b_n)$
3. Append s to C.
4. Repeat s times
   4.1. Choose a random permutation Q.
   4.2. Let $Z_s^* = (z_1, \ldots, z_n)$
   4.3. r_s := GetRandKey()
   4.4. For each 1 <= i <=n
      4.4.1. $z_{s,i}$:= EncryptBill(r_s , $u_{[Q(i)]}$)
   4.5. Save Z_s into C.
5. Let h := HASH(C)
6. Let h_j *be* the j-bit of h.
7. For j from 1 to s
   7.1. If $h_j = 1$ then
      7.1.1.For i from 1 to n do
         7.1.1.1.Let $k'_{i,s} = k_{[Q(i)]} * r_s$
         7.1.1.2.Append k' to C.
         7.1.1.3.Append $Q(P(i))$ to C.
   7.2. If $h_j = 0$ then
      7.2.1.for i := 1 to n
         7.2.1.1.Append r_s to C.
         7.2.1.2.Append $Q(i)$ to C.

Figure 6

VerifyBillShuffleProof_CutAndChoose
Input: A^*, B^* , C :Blob Output: Boolean
1. Extract *s* from stream *C*. 2. Check that *s* is equal or higher than the expected security threshold. 3. For *i* :=1 to s do 3.1. Extract Z_s from C into D. 4. Let *h* := HASH(D) 5. Let h_j *be* the *j*-bit of *h*. 6. If $h_j = 1$ then 6.1. For *i* from 1 to *n* do 6.1.1. Extract $k'_{i,s}$ and j from B_2. 6.1.2. If index j has already been used, then return false. 6.1.3. Check that Encrypt(u_j, $k'_{i,s}$) $= z_i$. If not equal return false. 7. If $h_j = 0$ then 7.1. for *i* := 1 to *n* 7.1.1. Extract r_s and $q_{s,i}$ from *B*. 7.1.2. If index $q_{s,i}$ has already been used, then return false. 7.1.3. Check that Encrypt($c_{[q_{s,i}]}$, r_s) $= z_i$. If not equal return false. 8. Return true

Figure 7

GetBillShuffleProof_Linear (modexp count independent on the number of elements)

Input:

A^*,B^* : List<Bill>
k : K
P :Permutation

Output:

C :Blob

8. Let $A^* = (a_1, \ldots, a_n)$ where each bill is expanded in its components
9. Let $B^* = (b_1, \ldots , b_n)$ where each bill is expanded in its components
10. Append s to C.
11. For u from 1 to s
    11.1. Choose a random value t_u
    11.2. Choose a deterministic cryptographically pseudo-random subset S_u of indexes of the elements in $B*$ based on t_u as seed.
    11.3. Let z = Product (i in S_u: b_i)
    11.4. r_u := GetRandKey()
    11.5. z_u := EncryptBill(r_s , z)
    11.6. Save z_u into C.
12. Let h := HASH(C)
13. Let h_j *be* the j-bit of h.
14. For j from 1 to s
    14.1. If h_j = 1 then
        14.1.1. Append (t_s*k) to C.
        14.1.2. Append #S_u to C
            14.1.2.1. For i from 1 to #A*
                14.1.2.1.1. if ($P[i]$ in S_u) then
                    14.1.2.1.1.1. Append i
    14.2. If h_j = 0 then
        14.2.1. Append t_s to C.

Figure 8

GenPubPrivKey_Linear

Output

(y,g) : M
x : K

1. x := GetRandKey()
2. Let g be a fixed system-wide default generator.
3. $y := g^x$
4. Return (y, g), x

Figure 8

Sign

Input
    x : K
    s_1, s_2 : M
    u :Message
Output
    (s,e) : M

5. k := GetRandKey() 6. $r := g^k$ 7. $e :=$ H(u || r) 8. $s := (k - x*e)$ 9. Return (s,e)

Figure 9

VerifySig

Input:
    s,e: M
    s_1, s_2 : M
    u :Message
Output
    b :Boolean

1. $r_v := g^s * y^e$ 2. $e_v :=$ H(u || r_v) 3. If $e = e_v$ then return True, else return False.

Figure 10

URePubKey

Input:
    (y,g) : M
Output
    (y',g') : M

4. k := GetRandKey() 5. $y' := y^k$ 6. $g' := g^k$ 7. Return (y',g')

Figure 11

To prove that a owns a bill, the owner can show the secret value s or prove he knows s in zero knowledge. If the owner shows s, then anyone can impersonate the owner and transfer the bill, so

the owner must prove the knowledge of s without revealing it. Also during the proof of knowledge of s, the owner must be able to associate the identity of the receiver of the bill, so that the proof is also a signature of the new address.
Bills can be created by money minting, such as when the solver of a block in a block-chain system is rewarded, or when some bills are combined into a new bill. When a bill is created by money minting, the exact value of the bill is public, so the owner has to commit to such value, and everybody is able to verify the correctness of the construction of a new bill. When a bill is created by combination, it's necessary to prove that the new bills value is in the acceptable range without revealing the value. This is done by a special zero knowledge proof.

Before the sender can transfer a bill, the sender must receive the public key (or address) related to the receiver for the Ure-Schnorr signature scheme. The receiver doesn't need to generate an address each time he receives a bill, since the sender can randomize the destination address by the universal re-encryption procedure. To transfer the bill the sender broadcasts a special transaction where he proofs the knowledge of the current destination address (the secret value s) and simultaneously and atomically asks peers to replace the term a_s of the bill with a new destination address.

Encrypted bills are not self-describing and some information required to decode the contents of a bill (more precisely the monetary value) need to be transferred on a private channel between the parties involved in the transaction. This is the monetary amount unblinding secret. A split transmission (public payment, private unblinding secret) is preferred to keep the block-chain, or any other storage system for the transaction log, as small as possible. This private channel can be supported by the same peer-to-peer network or by direct connection between payee and payer. If a direct connection is used, then the Tor network can be used to maintain full anonymity. In this paper we will present another alternative, similar to the BitMessage protocol, in which the same peer to peer network used for sending transactions is used to send informational messages. These messages must be held in the network nodes memory for some time (e.g. 1 week). During that period the payment receiver will be able to receive the associated information if he connects to the network. When this period is over, the receiver would need to contact the sender to be sent the associated information again. Also the payer can send this information to an e-mail account of the receiver.

In the following description, fields in brackets represent the additional description information that must be sent by some other communication channel.

Destination Address

Our destination addresses are modeled by a tuple (y,x) where the the owner of the address knows s such that y=E(s,x) (or $y=x^s$ in exponential notation). To sign, we need a proof of knowledge of s and a certain message to authenticate. This could be done in zero knowledge and non-interactively using a cut-and-choose method and Fiat-Shamir heuristic, nevertheless we'll use URe-Schnorr Signatures that has the advantage of much shorter signatures. While Schnorr signatures are not zero knowledge, the security of this signature scheme has been well established in practice.
A destination address can be specified in a expanded or compact format. In the expanded format, a destination address is the tuple (y,x). In the compact format, the destination address is a value y such that $y=g^s$ where g is a fixed generator specified in the system. For example, if AppeCoin is implemented over Z_p, and p is 1024 bits long, then the compact address format is 128 bytes long. If AppeCoin is implemented over the elliptic curve secp256k1, then a compact destination address size is 32 bytes long. Compact addresses are published by individuals to identify themselves, while expanded addresses are found in bills.

Creation of Open Bills

An open bill is an encrypted bill whose monetary value is evident for all users. The creation of open bills is needed for minting money. Money can be minted in different procedures of the p2p protocol. For example, solving a block by generating a proof of work is generally rewarded by a fixed amount of minted currency.

Given the value v publicly known, let x be the private key for the owner of the bill. The public bill a is created by using the procedure shown in figure 15.

CreateOpenBill

Input
    (s_1,s_2) : public key
    v : Integer (monetary value)
Output
    $a = (a_{s1}, a_{s2}, a_t, a_u, a_v, a_w)$: Bill

1. $a_{s1} := s_1$ 2. $a_{s2} := s_2$ 3. $a_t := d$ 4. $a_u := d$ 5. $a_v := d^v$ 6. $a_w := d$ 7. Return a

3.2 Payments

Payments are done by transferring the ownership of encrypted bills. To transfer the ownership of a bill a, the sender must prove the knowledge of the value s in field $a_s = (a_{s1},a_{s2})$ and in the same operation give new field a_s' which represents the new destination address, such that the previous field a_s is replaced by the new field. This is done by a signature of a_s' for the public key (a_{s1},a_{s2}) (the signature also cover some other fields transmitted).
Normal payments are uni-directional. The sender generates messages that use hybrid encryption. The messages use the Schnorr public key of the payee as a Diffie-Hellman public key to generate a Diffie-Hellman ephemeral key used for symmetric encryption. Symmetric encryption is specified by the functions **SymEnc**(key,iv, msg) and **SymDec**(key,iv, msg), where the message is encrypted in CTR mode using the given initialization vector iv.

Def: A **payment** consists of one or two messages. The first is the **Public Transfer Message** or **PT_MSG**. The first message is spread within the p2p network. The second is the **Secret Transfer Message** or **ST_MSG**. The ST_MSG is transferred privately between the parties involved, using a direct connection, using Tor or the p2p network itself.

Def: **Public Transfer Message or PT_MSG =** < $H(a_s)$, (c_1, c_2), h, (s,e) >
where

- $H(a_s)$ is a cryptographic hash of the bill identification. If the receiver user is always online

and has a copy of the block chain, then $H(a_s)$ should be enough to retrieve a. This field could also be the minimum binary prefix that unequivocally identified the bill.

- $a_s = (a_{s1}, a_{s2})$ is the identification of bill to transfer that must be changed, let $a = (a_{s1}, a_{s2}, a_t, a_u, a_v, a_w)$.
- c_1 replaces a_{s1}
- c_2 replaces a_{s2}
- h is an optional sender provided cryptographic hash of the Secret Transfer Message that will be sent along this message: h=Hash(ST_MSG). It can be used to prioritize and facilitate the broadcast of the associated ST_MSG. This may be necessary in order to prevent spam STSs. This is the only value not strictly related to the payment verification that must be stored along the transaction and cannot be trimmed. Nevertheless, since the size is fixed, the overhead is small.
- (s,e) is a signature of the whole PT_MSG (except for the signature itself) for the Schnorr public key (a_{s1}, a_{s2}).

if AppeCoin is implemented over Z_p, and p is 1024 bits long, then the PT_MSG is 552 bytes. If AppeCoin is implemented over the elliptic curve secp256k1, then the PT_MSG is only 168 bytes.

The Secret Transfer Message or ST_MSG message is essentially the encryption of a the secret values (t,v,w) using a generic version of IES (Integrated Encryption Scheme). When the private-key encryptor E is implemented in in the discrete log setting, then the public key hybrid encryption DLIES is used. If E is implemented on elliptic curves, then ECIES is used.

Def: **Secret Transfer Message or ST_MSG (sent via the p2p network)** = < $H(a_s)$, e_k , e_m , t_m , w_{src}, w_{dst} >

where:

- $H(a_s)$ is a cryptographic hash of the bill identification. $a_s = (a_{s1}, a_{s2})$
- $e_k = c_1^k$ where k is a random key. e_k will be used as one of the public terms that build a Diffie-Hellman ephemeral encryption key.
- *Let* $K_{DHE} := c_2^k$. K_{DHE} is a Diffie-Hellman ephemeral key. Let s be the private key for the Schnorr public key (c_1, c_2). Because $K_{DHE} := c_2^k$ and $c_2 = c_1^s$, then the receiver can also compute K_{DHE} as $e_k^s = c_1^{ks} = c_2^k$.
- $K_E \,||\, IV \,||\, K_M$ = KDF(K_{DHE}). KDF is a key derivation function. If K_{DHE} is not a bit-string, but a tuple, then K_{DHE} is converted to a bit-string. In ECIES, the x-coordinate of the point is taken.
- K_E is a symmetric encryption. IV is the chaining mode initialization vector
- K_M is a MAC key
- e_m := SymEnc(S_k , IV, (t,v,w)) (this is the monetary amount blinding secret)
- t_m := MAC(K_M , e_m)
- w_{src} is a small sender provided value (e.g. no more than 64 bytes) that can be used by the receiver to identify the sender for reimbursements. In that case, this value is specified in the w_{dst} field.
- w_{dst} is a small receiver provided value (e.g. no more than 64 bytes) that can be used by the receiver to detect if the payment was sent to the receiver, without checking using expensive modular exponentiation or requiring the private key s.

The MAC computation and verification may not be used, since the ST_MSG is signed in the

PT_MSG, and the receiver won't open a ST_MSG message before the associated PRM has been received. Nevertheless, it's included to adhere to the standard.

SendBill
Input $a = (a_{s1}, a_{s2}, a_t, a_u, a_v, a_w)$: Bill c : M w_{src}, w_{dst} : Binary-strings x : K (the bill owner private key) Output PT_MSG = < z , $(c_1, c_2), h, (s,e)$ > ST_MSG = < z , e_k , e_m , w_{src}, w_{dst} >
1. Compute $z := H(a_s)$ 2. k := GenRandKey() 3. Compute $e_k := c_1^k$ 4. Compute $K_{DHE} := c_2^k$. 5. Compute $K_E \| IV \| KM$:=KDF(K_{DHE}) 6. Compute e_m := SymEnc(K_E , IV, (t,v,w)) 7. Compute t_m := MAC(K_M , e_m) 8. Compute h := H(ST_MSG) 9. Compute (s,e) :=Sign(x, a_{s1}, a_{s2} , z || c_1 || c_2 || h) 10. Return PT_MSG and ST_MSG

Figure 12

A user can check that a bill has been sent to him by inspecting the field a_s of each bill a. But this requires the user to access its private key. This could lead to attacks where the attacker tries to extract side-channel information (such as timing each bill checks) in order to discover users the private key. The field w_{dst} allows users to check faster than a bill is for them, by providing the sender with one-way authentication tag. For example, the receiver can provide a tuple (x,MAC(k',x)) where k' is a private key of the receiver. Standard MAC functions evaluate faster than public key constructions and are generally less sensitive to side-channel attacks.
Other reason why the field w_{dst} is important is that generally is not recommended to use the same key used to redeem the bills to check the destination of the bill. This is because the key to redeem bills generally is stored in cold storage while the detection of incoming payments must generally be done in realtime with far less protection measures.
The ST_MSG can be as low as 180 bytes for an implementation of Appecoin on curve secp256k1.

CheckPTM
Input PT_MSG = < z, $(c_1, c_2), h, (s,e)$ > Output b :Boolean
1. Locate an unspent bill a with field a_s such that H(a_s) = z. This can be efficiently done by indexing the unspent bills table with H(a_s). If not bill is found, then return false 2. If VerifySig((s,e) , (a_{s1}, a_{s2}) ,z || c_1 || c_2 || h) = false, then return false 3. Return true

Figure 13

ReceiveBill
Input PT_MSG = < z, (c_1, c_2), h, (s,e) > ST_MSG = < z , e_k , e_m , t_m, w_{src}, w_{dst} > s : Key (the receiver's private key) Output b :Boolean
1. Compute h := H(ST_MSG) 2. Verify that H(ST_MSG) = h. If not return false. 3. Check that the additional authenticated data w_{dst} corresponds to the PT_MSG/ST_MSG message data. Use table lookups, MACs or any other fast authentication method protected from side-channels. If the receiver does not accept empty w_{dst} values, and w_{dst} is empty, then return false. 4. Check that the additional data w_{src} is valid, if not then return false. 5. Compute $c_1' := c_2^s$. If not $(c_1' = c_1)$ then return false. 6. Compute $K_E \mid\mid IV \mid\mid KM$:=KDF(K_{DHE}) 7. Compute t_m' := MAC(K_M , e_m) 8. Check that $t_m' = t_m$. If not, return false. 9. Compute (t,v,w) := SymDec(K_E , IV, e_m) 10. Compute $a_v' := a_u^{(t*v*w)}$. If not $(a_t' = a_t)$ then return false. 11. Compute $a_t' := a_u^{\,t}$. If not $(a_t' = a_t)$ then return false. 12. Compute $a_w' := a_u^{\,w}$. If not $(a_w' = a_w)$ then return false. 13. Return true

Figure 14

Checking Public Transfer Messages

Every user must check PT_MSGs are valid by executing the function CheckBill, described in figure 13.

Even if the bill payment seems invalid to the receiver, it may look completely valid for the rest of the network.

Accepting Payments

To accept a bill as payment the function ReceiveBill is executed. If the bill is valid for the receiver, the function returns true. Note that the receiver must pair each PT_MSG messages with the corresponding ST_MSG message in order to decide if the payment will be accepted or not. Obviously the receiver may also check that the received amount is the expected amount. The function to receive a bill is shown in figure 14.

Def: A **Bill Creation Message or BC_MSG** is the tuple (a,v) where

- a is a bill
- (a_{s1}, a_{s2}) is a public key
- $a_t := d$
- $a_u := d$
- $a_v := d^v$

- $a_w := d$
- v is the bill value, which must be in a valid range.

Proof that bills are still valid after public encryption

We must prove that after encryption, even if every field of a bill is encrypted with an unknown key k, the same properties regarding the bill value hold. Suppose the original bill is a is $a = (a_{s1}, a_{s2}, a_t, a_u, a_v, a_w)$. Let s, t,v,w be the secret values stored by the bills owner such that:

- $a_{s1} = a_{s2}^s$
- $a_t = a_u^t$
- $a_v = a_u^{twv}$
- $a_w = a_u^w$

After encryption we have:

- $b_{s1} = a_{s1}^k$
- $b_{s2} = a_{s2}^k$
- $b_t = a_t^k$
- $b_v = a_v^k$
- $b_w = a_w^k$

We can easily prove the required conditions still hold:

- $a_{s1} = a_{s2}^s \Rightarrow a_{s1}^k = a_{s2}^{sk} \Rightarrow b_{s1} = a_{s2}^{ks} \Rightarrow b_{s1} = b_{s2}^s$
- $a_t = a_u^t \Rightarrow a_t^k = a_u^{tk} \Rightarrow b_t = a_u^{kt} \Rightarrow b_t = b_u^t$
- $a_v = a_u^{twv} \Rightarrow a_v^k = a_u^{twvk} \Rightarrow b_v = a_u^{ktwv} \Rightarrow b_v = a_v^{twv}$
- $a_w = a_u^w \Rightarrow a_w^k = a_u^{wk} \Rightarrow b_w = a_u^{kw} \Rightarrow b_w = b_u^w$

3.3 Bill subdivision

A Bill subdivision allows a user to subdivide a bill a into bills of smaller amounts without disclosing such amounts. The bill subdivision consist of a single message (Bill subdivision Message or BS_MSG) that must be published in the p2p network. The idea behind bill subdivision is that the sender decomposes the encrypted bill into two or more encrypted bills. The monetary values of each new bill must add up to the monetary value of the original bill. To simplify the description, we'll show how to divide a bill in only two bills. The process can be applied recursively to divide in more parts. Nevertheless the same process can be applied to divide a bill directly in multiple sub-bills with small modifications.

Def: A **Bill Subdivision Message** or BS_MSG is the tuple

$< H(a_s), b, c, \text{DECOMP_PROOF}(a, b, c, n, m), (s, e) >$

where:

- $H(a_s)$ is a cryptographic hash of the bill identification.
- $a_s = (a_{s1}, a_{s2})$ is the identification of bill to transfer that must be changed, let $a = (a_{s1}, a_{s2}, a_t, a_u, a_v, a_w)$. The bill a is disposed after the message is accepted, and the new bills b and c will be added to the database of active bills.

- b and c are new bills to create. The monetary values of b and c add up to the value of a.
- n, m is the range of the monetary value to subdivide. Using a reduced range allows to build shorter proofs (but it may leak an upper or lower limit on the amount decomposed)
- DECOMP_PROOF(a, b, c, n,m) is a non-interactive proof of decomposition of a into b and c.
- (s,e) is a signature of the whole PT_MSG (except for the signature itself) for the Schnorr public key (a_{s1}, a_{s2}).

We'll protocols for a proof of decomposition. The first protocol (DECOMP_PROOF1) is easier to understand an analyze. The second protocol (DECOMP_PROOF2) is an optimization of the first, where some operations are merged to reduce the proof size and verification time. For all these definitions let:

- $a = (a_{s1}, a_{s2}, a_t, a_u, a_v, a_w)$.
- $b = (b_{s1}, b_{s2}, b_t, b_u, b_v, b_w)$.
- $c = (c_{s1}, c_{s2}, c_t, c_u, c_v, c_w)$.

Proof of Decomposition

Def: **DECOMP_PROOF**(a, b, c,n,m) is a proof that:

1. If the values of the bills a, b and c are v_a, v_b, and v_c respectively, then $v_a = v_b + v_c$
2. v_b and v_c are positive and in the range $[2^n .. 2^{(m-n+1)}]$

We define two different splitting protocols **DECOMP_PROOF 1** and **DECOMP_PROOF2**

DECOMP_PROOF1

Input:

a, b, c : Bill

Output:

$b_v', c_v', a_t', a_w', a_v', b_v'', c_v'', z_b', z_c'$: Key

$P_k, P_{bq}, P_{bz}, P_{bd}, P_{cq}, P_{cz}, P_{cd}$: Blob

1. Compute $b_v', c_v', a_t', a_w', a_v', P_k, k$ = AddZerosSplitEncryptWithProof(a_t, a_w, a_v, v_b, v_c)
2. Compute Choose a random key q_b
3. Compute (b_u, b_t, b_w, b_v''), P_{bq} = EncryptWithProof((a_u, a_t', a_w', b_v') , q_b)
4. Compute $(b_v, z_b', P_{bz}, P_{bd})$ = RemoveZerosWithProofs($b_v'', b_t, - k_b{*}q_b, w{*}v_b$)
5. Choose a random key q_c
6. Compute (c_u, c_t, c_w, c_v'') , P_{cq} = EncryptWithProof($(a_u', a_t', a_w', a_v',)$, q_c)
7. Compute $(c_v, z_c', P_{cz}, P_{cd})$ = RemoveZerosWithProofs($c_v'', c_t, - k_c{*}q_c, w{*}v_c$)
8. Return ($b_v', c_v', a_t', a_w', a_v', b_v'', c_v'', z_b', z_c', P_k, P_{bq}, P_{bz}, P_{bd}, P_{cq}, P_{cz}, P_{cd}$)

Correctness Analysis of protocol SPLIT_PROOF1 for branch b (branch c is equivalent):

After step 1:

$a_z = a_v * z$

$a_v' = a_z^k = a_v^k * z^k = a_u^{t*w*v*k} * z^k$

$b_v' = a_t^{w*v(b)*k} * z^{k(b)}$

P_k proves the knowledge of k

After step 3:

$b_v'' = b_v'^{q(b)} = a_t^{w*v(b)*k*q(b)} * z^{k(b)*q(b)}$

P_{bq} proves the knowledge of q_b

After step 4:

$z_b' = z^{-k(b)q(b)}$; P_{bz} proves the knowledge of $-k(b)q(b)$

$b_v = b_v'' * z_b' = a_t^{w*v(b)*k*q(b)} * z^{k(b)*q(b)} * z^{-k(b)k(b)} = a_t^{w*v(b)*k*q(b)} = b_t^{w*v(b)}$

P_{bd} proves the knowledge of $w*v_b$ such that

$b_u^{t*x} = b_t^x = a_t^{w*v(b)*k*q(b)} = b_t^{w*v(b)} = b_u^{w*t*v(b)}$

Def: VerifyEncryption(L_2 : List<M>, L_1 :List<M>, k : K) returns true iff for every element at index i , $L_2[i] = L_1[i]^k$

DECOMP_PROOF1_Verify
Input: $a,b,c, b_v' , c_v' , a_t', a_w ', a_v' , b_v'' , c_v'' , z_b' , z_c'$:Message $P_k , P_{bq} , P_{bz} , P_{bd} , P_{cq}, P_{cz} , P_{cd}$: Blob Output: b :boolean
1. if not VerifyEncryption ($(a_t', a_w ', a_v')$, $(a_t, a_w , a_v * z)$, P_k) then return false 2. Verify that $b_v' * c_v' = a_v'$, if not then return false 3. If not VerifySlice ($a_u , a_t', a_w', b_v' , b_v , b_v'' , z_b'$) , P_{bq} , P_{bz} , P_{bd}) then return false 4. If not VerifySlice ($a_u , a_t', a_w', c_v' , c_v , c_v'' , z_c'$) , P_{cq} , P_{cz} , P_{cd}) then return false 5. return true

VerifySlice
Input: $a_u , a_t', a_w', b_v' , b_v , b_v'' , z_b'$: Message Output: b :boolean
1. if not VerifyEncryption ((b_u , b_t, b_w , b_v'') , (a_u , a_t', a_w', b_v') , P_{bq}) then return false 2. if not VerifyEncryption ((z_b' , z) , P_{cz}) then return false 3. Verify that $b_v = b_v'' * z_b'$ 4. if not VerifyEncryption ((b_v , b_t) , P_{cd}) then return false 5. return true

AddZerosSplitEncryptWithProof
Input: $a_t, a_w , a_v , w, v_b, v_c$: Message Output: $b_v' , c_v' , a_t', a_w ', a_v'$: Message P_k : Blob k : Key
1. Choose two random keys k_b and k_c. 2. Let $k = k_b + k_c$ 3. $a_z = a_v * z$ 4. Compute $(a_t', a_w ', a_v'),P_k$ = EncryptWithProof(a_t, a_w , a_z, k) 5. Compute $z_b = z^{k(b)}$ 6. Compute $z_c = z^{k(c)}$ 7. Compute $b_v' = a_t'^{w * v(b)} * z_b$ 8. Compute $c_v' = a_t'^{w * v(c)} * z_c$ 9. b_v' and c_v' should verify that $b_v' * c_v' = a_t^{w *(v(b) + v(c)) * k} * z^{(k(b) + v(c))} = a_v'$ 10. Return b_v' , c_v' , P_k , k

EncryptWithProof
Input:

X : List<Message>

Output:

Y : List<Message>

P_k : Blob

1. Compute Y = Encrypt(*X*, *k*)
2. Compute P_k = EQKEY_PROOF(*Y*, *X*, *k*)

RemoveZerosWithProofs

Input:

x, b, k_z, k_d

X,*Y* : List<M>

Output:

y, z', P_z, P_d

1. Compute $z' = z^{k(z)}$
2. Compute $y = x * z'$
3. Compute P_z = KEY_PROOF(z' , z ,k_z)
4. Compute P_d = EQKEY_PROOF($<y> \| Y$, $<b> \| X$, k_d)

Def: **EQUALGEN_PROOF**(*X* :List<*M*>, *Q* : List<*K*>, *g* :*M*)

Is a non-interactive proof that each element in *X* is an encryption of *g* under a known key.
It is equivalent to the concatenation of, for each possible *i* , KEY_PROOF($x[i]^{Q[i]}$,g, *Q*[*i*]).
We'll give two implementations, the simple one (EQUALGEN_PROOF1), is the application of the definition. The second (EQUALGEN_PROOF2) based on the random subset problem which requires less processing and less space if the number of elements is greater than half the security threshold.

EQUALGEN_PROOF1

Input:

X :List<*M*>, *Q* : List<*K*>, *g* :*M*)

Output:

B :Blob

1. For *i* in [0..Length(*X*)-1] do
   1.1. Compute $P_{e(i)}$ = KEY_PROOF($x[i]^{Q[i]}$, *g* , *Q*[*i*])
   1.2. Append $P_{e(i)}$ to B
2. Return B

EQUALGEN_PROOF2 (something is missing here)

Input:

X :List<M>, Q : List<K>, g :M) Output: B :Blob
3. For s in [1.. securityThreshold] 3.1. Let I be a random subset of elements of valid indexes i. 3.2. Compute R = Product(i in I: $x[i]^{Q[i]}$) 3.2.1. Compute $P_{e(i)}$ = KEY_PROOF($x[i]^{Q[i]}$, g , $Q[i]$) 3.3. Append $P_{e(i)}$ to B 4. Return B

Def: **KEY_PROOF**(m :M, g :M, k :K), which is a zero knowledge proof that g encrypts to m with a key k known to prover.

Def: **EQKEY_PROOF**(Y :List<M>, X :List<M>, k :K) which is a zero knowledge proof that the list Y is the result of encryption of the list X with the key k. This can be accomplished by multiple calls to KEY_PROOF.

First we'll define some auxiliary proofs (SPLIT_PROOF and INRANGE_PROOF):

Def: **SPLIT_PROOF**(a, b, c) is a proof that:

1. The bills b and c are valid bills
2. If the monetary values of the bills a, b and c are v_a, v_b, and v_c respectively, then $v_a = v_b + v_c$

Proof of Range of Amount

Def: **INRANGE_PROOF**(a,n,m) is a proof that a bill a (which is encrypted by key k) represents a valid amount of money greater or equal to 2^n and lower than $2^{(n+m)}$. This proof can ensure the amount is positive and restricted to a certain valid range (no overflow has occurred).

INRANGE_PROOF

Input:

a : Bill

n,m :Integer

Output:

$(P_z, P_s, , P_q, P_t, P_w)$: Blob

1. Build a list of unit value powers from n to m.
   $A = < a_u^{2^n}, a_u^{2^{(n+1)}}, a_u^{2^{(n+2)}}, ..., a_u^{2^{(m-1)}}, a_u^{2^{(m)}} >$
2. Build a list of private powers of the zero value generator. Each element is encrypted with a unique private randomization factor $r(i)$.
   Let $N = (m\text{-}n+1)$
   $Z_r = < z^{r(0)}, z^{r(1)}, \dots , z^{r(N-2)}, z^{r(N-1)} >$
   $R = < r(0), \dots , r(N-1) >$
3. Now each element in A is multiplied by an element of Z_r, building the list B:
   $B = < a_u^{2^n} * z^{r(0)}, a_u^{2^{(n+1)}} * z^{r(1)}, ..., a_u^{2^{(m-1)}} * z^{r(N-1)} >$.
   $\#B = N$
   The multiplication "*" represents the homomorphic operator on messages.
4. A second list of single zeros is built. Each element is encrypted with a unique randomization factor $l(i)$.
   $Z = < z^{l(0)}, z^{l(1)}, \dots , z^{l(N-2)}, z^{l(N-1)} >$.
   $L = < l(0), .. , l(N-1) >$
   $\#Z = N$.
5. Compute P_z = EQUALGEN_PROOF($Z_r \parallel Z$, $R \parallel L$, z). This is a proof that each of the values in $Z_r \parallel Z$ is an encryption of z under the corresponding key in the list $R \parallel L$.
6. Compute $C = B \parallel Z$. C is the concatenation of B and Z. #C = 2*N
7. Now C will be shuffled with the permutation P and each element will be re-encrypted into with a private key k into the resulting vector D. We add three first elements to the list (elements a_v , a_t and a_w) which are encrypted but not shuffled. These three elements are witnesses of the encryption.
   Compute (H , k , P_s) = ShuffleSkipWithProof($< a_v , a_t , a_w > \parallel C$, 3)
8. We therefore have:
   $H = < a_v', a_t', a_w' > \parallel D$
   $a_v', = a_v^k , a_t' = a_t^k , a_w' = a_w^k$
   $D = < C_{P(1)}^k, \dots , C_{P(2*N)}^k >$
9. Let v be the monetary value of a. Let $v[j]$ be the j-bit of the binary representation of v. We'll extract exactly N elements of D and Z and form the set D'. Let I be set of the indexes of the elements of D that are extracted into D'. An element cannot be extracted twice.
10. Let D' be an empty list.
11. For each 0<=j <N:
    If $v[j+n]$=1, then the element $(a_u^{2^{(j+n)}} * z^{r(j)})^k$, originally from D, is added to the list D'.
    If $v_b[j+n]$=0, then any element at index i, $(z^{m(i)})^k$, originally from Z, is added to the list D'.
12. Compute x = Prod (0 <= i < N: $D'[i]$).
    x will be equal to $a_u^{v*k} * z^q$, for some secret exponent q.
13. Now we remove the zeros (terms generated by z with a known key) of D' with a proof than only zeros are removed. Let $z' = z^q$
14. Compute P_q = KEY_PROOF(z', z, q). Here we prove that z' is a power of z, without

publishing q.

15. Let $x' = x * z'^{-1} = a_u^{v*k}$
16. Now is must be the case that $x'^{tw} = a_v'$, because
    $a_v' = a_v^k = a_v^k$
    $x'^{tw} = a_u^{v*k*t*w} = a_u^{v*k*t*w} = a_u^{t*w*v*k} = a_v^{k,}$
17. Now we'll prove that we can reach a_v' from x' by encrypting with t and w.
    Let $x_t = x'^{t} = a_u^{v*k*t}$
18. Compute P_t = EQKEY_PROOF((x_t', a_t), (x' , a_u) , t). This proves that x_t' is the encryption of x' and that a_t is the encryption of a_u using the key t)
19. Compute P_w = EQKEY_PROOF((a_v' , a_w), (x_t' , a_u) , w). This proves that a_v' *i*s the encryption of x_t' and a_w is the encryption of a_u using the key w.
    Since $a_v' = a_v^k$, the value is proven.
20. Return $(P_z, P_s, , P_q, P_t, P_w)$

<u>Def:</u> **DECOMP_PROOF2**(a, b, c,n,m) = < P_b, P_c > where

- $a_v = b_v * c_v$
- P_b =INRANGE_PROOF(b,n,m)
- P_c =INRANGE_PROOF(c,n,m)

<u>Def:</u> **DEC_MSG**(a, b, c, n,m) is the message =
< a, b, c, n, m , **DECOMP_PROOF**(a, b, c,n,m)>

Bill combination (this section is incomplete)

To combine bills a message similar to the bill subdivision is composed:

<u>Def:</u> Bill combination message = < *a , b, c ,n, m, P* >

Where:

- a is the bill to create by combination of the remaining bills.
- b and c are the new bills to combine. These bills must be discarded afterward.
- P is a **DECOMP_PROOF**(a, b, c,n,m)

Optimized Proofs for certain HCGCs

We have described proofs that work for an abstract HCGCs, without considering any property of an specific HCGC. All the presented proofs rely on the cut-and-choose method. We'll now present optimized proofs that work with the Poligh-Hellman HCGC.

Proof of knowledge of Encryption Key

<u>Def:</u> **KEY_PROOF**(b,a,k) is a proof that the owner knows a private key k that allows him to encrypt a into b, without revealing k.

There are several methods to achieve this proof, either interactively or non-interactively. There are other methods designed for the Pohlig-Hellman HCGC, such as the Schnorr's Id Protocol.

Proof of knowledge and equivalence of Keys

Def: **EQKEY_PROOF**(k , (a_1,b_1) , ... , (a_n, b_n)) which is a proof that each a_i encrypts to the corresponding b_i with a the same key k and that key is known to the prover, without revealing k.
To achieve this proof we can use the Chaum-Pedersen protocol.

Proof of Correct Shuffle

Def: SHUFFLE_PROOF($a_1, .. , a_n , b_1, .. , b_n , k_1, .. , k_n$) is a zero knowledge proof of correctness of the shuffle and re-encryption operation of $a_1, .. , a_n$ into $b_1, .. , b_n$ with keys $k_1, .. , k_n$.

As with KEY_PROOF, there are several methods to achieve this proof. And again, the most studied method is the cut-and-choose proof. The method to create the proof of shuffle of elements of M is similar to the method described in the GetShuffleProof procedure to create a proof of shuffles of bills.

Bill mixing

It's sometimes necessary to shuffle a set of bills together to increase anonymity and avoid traceability. To do so the shuffler publish the following message:

Def: **Bill mixing Message** = < $a_1, .. , a_n , b_1, .. , b_n , P$ >
Where:

- $a_1, .. , a_n$ are the original bills to be mixed. These bills must be discarded afterwards.
- $b_1, .. , b_n$ are a permutation and re-encryption of the bills $a_1, .. , a_n$ with a single key k.
- P = **GetBillShuffleProof**($a_1, .. , a_n , b_1, .. , b_n , k$) is a proof of correctness of the shuffle/re-encryption operation

3.4. Optimized Shuffle proofs

To allow the mixing of hundreds of bills without requiring high amounts of work by each verification node of the network, we present an optimized shuffle proof protocol and its corresponding non-interactive counterpart.

The optimized shuffle proof (GetBillShuffleProof_Linear) is based on a mixture of the Random-subset Test and the cut-and-choose protocol. In the interactive version, there is an source list of bills and a destination shuffle-encrypted list of bill. At each round the prover re-encrypts and shuffles the destination list of bills to create a temporary list of bills, using a secret round key. Then every source bill is decomposed in its components ($a_{s1}, a_{s2}^{s,,} a_t , a_v , a_{w,} a_u$) and all components are stored ordered in a vector. The verifier provides a random seed and a coin flip, and the prover generates a pseudo-random sub-set of indexes in the vector using the received seed and selects the components contained in those positions. Then the verifier multiplies all selected components and, depending on the flipped coin value:

a) shows a set of indexes in the temporary bill list and the result of multiplying all of them. Shows a

compound key (original encryption key multiplied by round key) that maps the multiplication of the selected source components to the multiplication of temporary components, where mapping is the Bill re-encryption function. Because of the Representation Problem, finding the a different subset that, being multiplied, has the same result is difficult. The verifier verifies these operations and accepts only if they are correct.

b) shows the round-key and the permutation used (this can be shown as a seed for a pseudo-random generator of permutations or as the full permutation). The verifier chooses a random subset of components (or the same subset that the seed generates), multiply them and check that their multiplication equals the multiplication of the elements in the temporary list that match the source sub-set by the permutation.

The probability that the prover cheats the verifier in each round is ¾ (higher than ½ as in the stander cut-and-choose protocol). This can be corrected simply by adding more rounds accordingly.

The protocol can be easily turned into a non-interactive proof using the Fiat-shamir heuristic. Each seed and the coin-flip in each round are obtained as outputs of a pseudo-random generator whose seed is obtained from a commitment hash. The commitment hash forces the prover to commit to every temporary list chosen. In the non-interactive proof, it's not necessary that each temporary list is explicitly part of the non-interactive proof. The commitment hash can be built as the concatenation of hashes of the temporary round lists, and all temporary list hashes are included in the non-interactive proof. If the coin flip requires the compound key to be shown (case a) then the temporary list of that round is revealed. If the coin flip requires the round key (case b) then the temporary list can be computed by the verifier using the round key and it can be checked against the corresponding hash.