Digital Cash Design Document

By Justin Fiedler and Steven Kappes

I. Abstract

In this document we describe a technique for implementing a digital cash system and show how it can be effectively used to handle anonymous purchases between entities while preventing misuse. Our technique is equipped to handle multiple levels of attacks both those that could exploit the protocol itself and those which are inherent to any networked system.

II. Entities

There are three entities in the design, the bank, customer, and merchant:

- Bank = B
  The bank manages all of the accounts of customers and merchants, in particular their balances. The bank handles the withdrawal and deposit of the digital cash by the customers and merchants.

  The bank will need to keep all deposited money orders, a list of the keys to open the selected identity strings, and which half of the identity string the keys apply to.

- Customer = C
  The customer can request digital cash from the bank, and then, use it to pay a merchant.

  The customer will need to generate money orders. For each money order, the customer will then need to store the encryption keys of the money orders, the encryption keys of the identity strings, and the blinding factors for the hashes of the money orders.

- Merchant = M
  The merchant accepts payment from customers and informs the bank to credit their account with the correct amount paid by the customer.

The merchant will need to retain the past money orders received from customers, along with the keys for the identity strings and which half each belong to. The stored money orders will be used to verify that the customer does not attempt to send the same money order to a single merchant twice.

III. Message Flow

To complete a money transfer from a customer to a merchant, a series of eight messages are required. All customers have a random string I_C which will be used to identify them. All messages that are sent to the bank will include the user's identity and password. This will allow the bank to verify who the user is. SSL will be used to provide secrecy, authentication, and data integrity.

The following notation is used to describe money orders:

MO_i = The i-th money order
$= U_s \| I_{i:1} \| \ldots \| I_{i:n} \| Amt$
Us = A unique string generated by C
Amt = Amount of the money order
$I_{i:j}$ = The j-th identity string for the i-th money order which is secretly split then encrypted with separate keys. The hash is appended to ensure commitment of the two values.
$= E_{K_{Li:j}}(I_{Li:j} \| H(I_{Li:j})) \| E_{K_{Ri\text{-}j}}(I_{Ri:j} \| H(I_{Ri:j}))$
$K_{Li:j}$ = Unique symmetric encryption key for the j-th identity string for the i-th money order
$K_{Ri:j}$ = Unique symmetric encryption key for the j-th identity string for the i-th money order
$I_{Li:j}$ = Random bit string the same length as the I_C
$I_{Ri:j} = I_{Li:j} \oplus I_C$

Message 1: C → B: $MO_{K1} \| \ldots \| MO_{Kn} \| BH_1 \| .. \| BH_n$

The customer first generates N money orders which are all encrypted with a symmetric key and the hash of the money orders which are blinded.

MO_{Ki}= The i-th money order encrypted with K_{MOi}

= $K_{MOi} (MO_i)$

K_{MOi}= Symmetric key used to encrypt the i-th money order

BH_i = Blinded hash of the i-th money order

= $H(MO_i)^* (Z_i)^e$

Z_i = A random number generated by C

e = Public key of B for RSA signing

Message 2: B → C: i

The bank responds by requesting C to provide the necessary information to decrypt the money orders, the identity strings, and unblind the hashes of all money orders except for the i-th money order. The bank uses this information to verify that the constructed money orders are legitimate.

i = Index in the range of 1 to N which specifies the money order B will sign. Therefore, C is required to send the values necessary to decrypt and unblind all other money orders. This value is constant through the remainder of the protocol.

Message 3: C → B: $\mathbf{K_{MO1} \| \ldots \| K_{MOi-1} \| K_{MOi+1} \| \ldots \| K_{MOn} \|}$
$\mathbf{Z_1 \| \ldots \| Z_{i-1} \| Z_{i+1} \| \ldots \| Z_n \|}$
$\mathbf{K_{Lm:1} \| \ldots \| K_{Lm:i-1} \| K_{Lm:i+1} \| \ldots \| K_{Lm:n} \|}$
$\mathbf{K_{Rm:1} \| \ldots \| K_{Rm:i-1} \| K_{Rm:i+1} \| \ldots \| K_{Rm:n}}$

The customer provides the requested symmetric keys for the money orders and identity strings, as well as the unblinding factors for the hashes. The bank will use these values to ensure the N-1 money orders are all of the same dollar amount, the identity string keys correctly decrypted the identity strings based on the appended hash, and money order hashes are correct.

$K_{Lm:j}$ = Symmetric encryption key at index j for money order m

$K_{Rm:j}$ = Symmetric encryption key at index j for money order m

Message 4: B → C: B{ BH_i }

If the money orders appear legitimate, the bank signs and returns the i-th blinded hash order which was not unblinded by C.

B{ BH_i }= Blinded hash for the i-th money order generated by C signed by B.

= $(H(MO_i) * (Z_i)^e)^d = H(MO_i)^d * Z_i$

d = Private key of B for RSA signing

Message 5: C → M: MO_i || B { H_i }

The customer unblinds the hash of the money order and sends it along with the corresponding money order to the merchant.

B{ H_i }= The hash for the i-th money order generated by C signed by B.

= $H(MO_i)^d * Z_i / Z_i = H(MO_i)^d$

Message 6: M → C: b_1 || ... || b_n

The merchant verifies the banks signature along with checking that the hash matches the sent money order. If both of these checks pass the merchant requests one of the two keys for each identity string in the money order i.

b_j = Value of either 0 or 1. Indicates which key, KLi:j or KRi:j, must be returned by C.

Message 7: C → M: $K_{xi:1}$ || ... || $K_{xi:n}$

The customer provides the keys for the requested halves of all identity strings. The merchant verifies that these keys are correct by decrypting and checking the identity string with its hash.

$K_{xi:j}$ = The left or right symmetric encryption key, based on x, at index j for money order i.

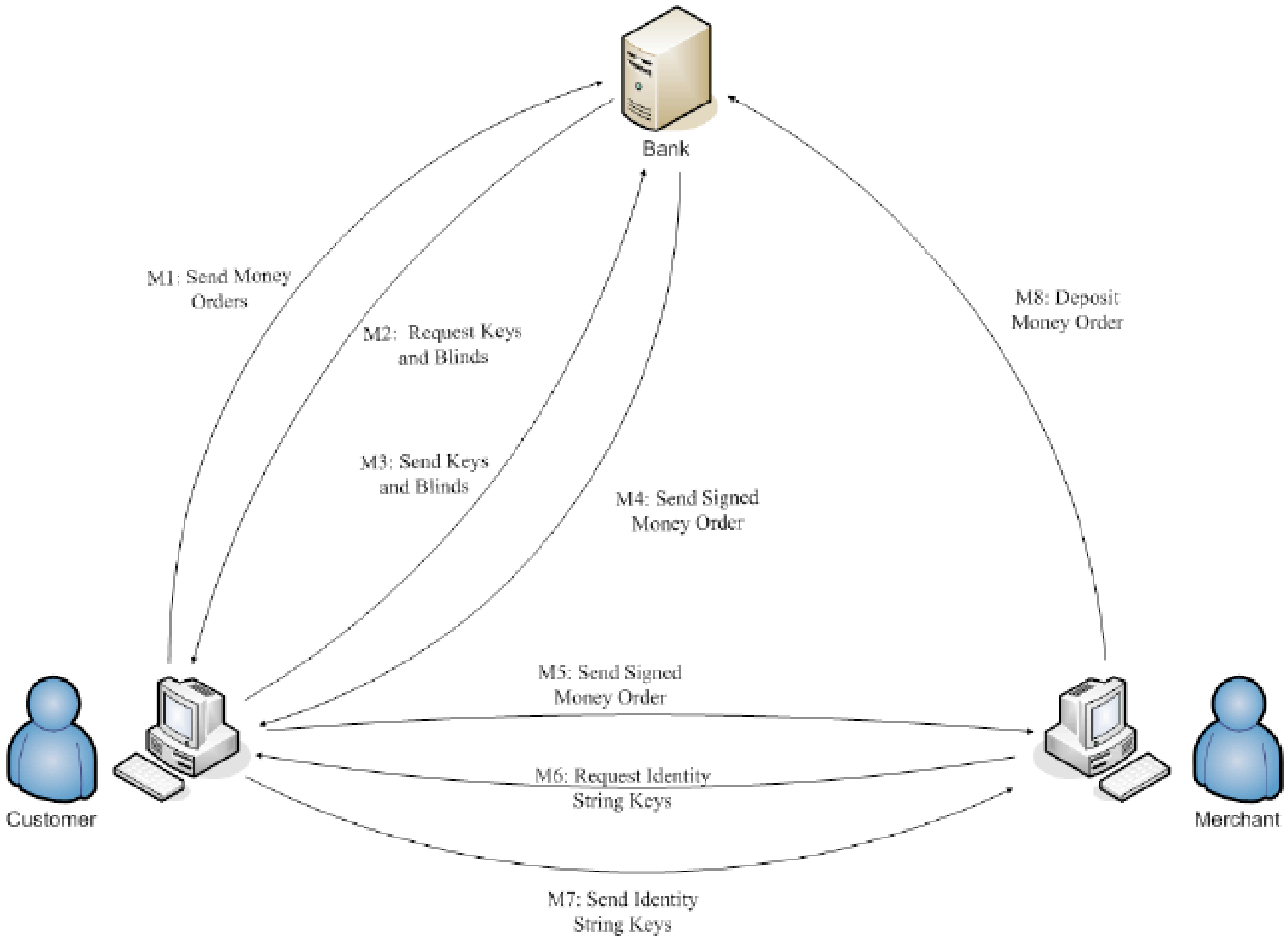
x = Either L or R based on the values of bj. This value is different for every j.

Message 8: M → B: $\mathbf{MO_i \,||\, B\{ H_i \} \,||\, b_1 \,||\, ... \,||\, b_n \,||\, K_{xi:1} \,||\, ... \,||\, K_{xi:n}}$

The merchant sends the money order to the bank, along with half the keys for the identity strings and which half they correspond to. The bank checks to make sure the money order has not been previously spent. If not, it checks that the money order is valid by checking the signed hash and the identity keys. If the money order is accepted, the bank deposits the given amount into the merchants account. If the money order has been spent before, the bank checks the identity strings and keys to determine who is to blame, the merchant or the customer.

IV. Architecture Diagram

The components are the same as the entities given in part one. There are no subcomponents. The diagram on the next page shows all necessary messages exchanged between the bank, customer, and merchant to withdraw and deposit a single money order.


Bank
M1: Send Money Orders
M2: Request Keys and Blinds
M3: Send Keys and Blinds
M4: Send Signed Money Order
M8: Deposit Money Order
M5: Send Signed Money Order
M6: Request Identity String Keys
M7: Send Identity String Keys
Customer
Merchant

V. Threat Model

Authentication, secrecy, and data integrity are provided by SSL. This removes many potential security threats without any extra work:

1. Stealing money orders as they are sent from entity to entity is protected through secrecy.

2. Impersonating the bank to steal money orders is prevented do to server authentication. This also makes it impossible for the user to attempt a man-in-the-middle attack by acting as a proxy between the client and server.

3. Replaying an encrypted money order is prevented by the fact that SSL is safe against replay attacks. This is augmented by the checks made by the digital cash protocol.

There are a number of threats within the digital cash protocol as well. Most of these threats are based on the variable N. In the protocol, the variable N specifies both the number of money orders sent to the bank during a request and the number of identity strings, which are secretly split, per money order. Increasing the size of N decreases the probability that these attacks will succeed. In our implementation, we set N to 20. This is a compromise between security and performance, as a larger N involves more computation as well as more bandwidth usage. These threats based on the variable N include:

1. The customer cheating the bank during a withdraw:

   Since the goal of this protocol is to ensure anonymity of the origin of money orders deposited by the merchant, the bank does not get to view the money order and corresponding hash which will be signed. Otherwise, the bank could identify the money once it is deposited.

   To protect against fraudulent withdrawals, the bank requires the customer to send N orders. The bank will be allowed to view N - 1 of those and will sign the remaining one if they are all valid. Consequently, the customer could exploit this fact to get the bank to sign a hash which does not belong to a money order sent to the bank or sign the correct hash to a invalid money order. An invalid money order could include either a greater amount than the checked orders, identity strings which do not identify the customer or both. The security against this attack is based on N. The customer will have a 1/N chance (5% in our implementation) chance of cheating the bank. With the recourse of banning the customer from the bank upon the first detection, the customer would be ill-advised to attempt this attack.

2. Reuse of digital cash:
   It is trivial for the bank to detect reuse of a money order by simply keeping a record of all the past deposited money orders. However, it is important that the bank bans the correct entity which reused the digital cash, either the customer or merchant. The variable N is used to determine this as well. Using the identity encryption keys revealed during deposit, the bank can determine the fraudulent entity.

   When the bank sees a money order deposited twice, it can check the identity encryption keys that are sent. If all of the keys sent are the same, the bank can infer that the merchant tried to cheat the bank. This is due to the fact the merchant cannot create the missing keys. Since the merchant sent the same identity encryption keys to correctly decrypt the identity strings, the merchant must be depositing the same money order twice. There is an outside chance that the customer is at fault in this case though. This could only happen if two separate merchants asked the customer to reveal the exact same identity encryption keys. The chance of this situation is extremely low with a probability of $1/2^N$ (approximately 0% in our implementation). Consequently, the bank will ban the merchant.

   If the keys are different, it is sure that the customer tried to spent the same money twice since the merchants would not have access to the different keys. In this case, the bank can decrypt both halves of the identity string for the keys that are different and determine the identity of the customer.

The other threats to this protocol is the chance of a customer or merchant to construct a valid money order without the money being withdrawn from their account. This could be attempted in two ways:

1. Finding a hashing collision between money orders:
   If the customer can find two money orders of different amounts which hash to the same value, the customer could send the smaller money order to the bank to sign and then use the signed hash to spend the larger amount. This prevention of this attack will be based on collision resistance of SHA-1. Strong collision resistance will prevent the user from finding two money orders which result in the same hash. Weak collision resistance will prevent the user from generating a money order which has a specific hash. In this case, even if a user could find a bit stream that produces a specific hash, the

user would also have to determine the encryption keys to produce the correct identity strings. As a result, it is infeasible to find a collision among money orders, as the attacker would have to revert to a brute force method.

2. Constructing a valid signed hash without the bank:
   The user could also attempt to find a valid hash without the help of the bank. The user could unsign a random bit stream with the bank's public key to produce the hash. However, the user could not create a money order for the specific hash based on the reasoning above. The user will also not be able to sign a hash due to the security of RSA signing. The user would have to have the bank's private key to sign the hash. Brute force could again be attempted, but is infeasible due to time constraints.

Finally, the user could attempt to send an money order with a signed hash that does not correspond to the money order. This will quickly be revealed by verifying the bank's signature with the hash of the money order.

VI. XML DTD

A single DTD is used to send all messages in the XML format. The format of the DTD is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT message (header, item*)>
        <!ELEMENT header (msgNum, customerID?, password?)>
                <!ELEMENT msgNum EMPTY>
                <!ATTLIST msgNum number CDATA #REQUIRED>

                <!ELEMENT customerID EMPTY>
                <!ATTLIST customerID id CDATA #REQUIRED>

                <!ELEMENT password EMPTY>
                <!ATTLIST password text CDATA #REQUIRED>

        <!ELEMENT item EMPTY>
        <!ATTLIST item tag CDATA #REQUIRED>
        <!ATTLIST item index CDATA #REQUIRED>
        <!ATTLIST item value CDATA #REQUIRED>
```

Each XML message will contain a message node which contains a header and multiple items. The header is comprised of a mandatory message number corresponding to the message order described in the architecture flow section. The customer ID and password will be used on all the messages from the customer or merchant to the bank to identify the user.

The item elements includes three attributes: the tag, index, and value. The tag is used to specify what is contained in this element, such as a money order string, encryption key, blinding factors, etc. The index is used to indicate different elements which contain the same tag. The value attribute is used to store the data of the element.

This format was chosen for its simplicity and expandability. This makes constructing and parsing intuitive for both the server and clients without requiring a bloated message format. For instance, message 3 contains N - 1 money order encryption keys, 2 * N * (N - 1) identity string encryption keys, and N - 1 blinding factors. If all of these elements were individually mapped in the DTD, the number of elements would increase significantly. This also allows the protocol to change the number of elements sent without rewriting the DTD.

VII. Future Extensions and Known Bugs

Albeit minor, there is one known issue in our implementation. Our hash function manually sets the most significant byte of the digest to be 1. This was necessary to for correct functioning when converting between bytes and BigInteger objects. The BigInteger object is required since the hash needs to be multiplied by a blinding factor. For unknown reasons, if the most significant bit of this byte becomes one, the BigInteger will not return the correct bytes after it is created, even when it is explicitly initialized as a positive number.

Therefore, this solution was chosen and ensures the BigInteger object behaves as expected. While this may make finding a collision between hashes easier, it is still unlikely.