Scrit: A distributed untraceable electronic cash system

Jonathan Logan and Frank Braun

2019-10-30 (draft)

Abstract

Scrit (secure, confidential, reliable, instant transactions) is a federated Chaumian ecash (see Chaum, Fiat, and Naor 1990). Coins in Scrit are so-called digital bearer certificates (DBCs) issued by mints. Scrit mitigates the issuer risk common in other DBC systems by employing n mints in parallel. It has the maximally achievable transaction anonymity, the anonymity set of a DBC equals or is greater than all DBCs ever issued in that denomination during the defined epoch. Transactions are extremely cheap and fast, the settlement is network latency bound leading to sub-second confirmation times.

Contents

1 Introduction

A problem of previous Chaumiam ecash systems has been their centralization in both a technical and a governance sense, caused by employing a single mint. This has exposed these systems to technical and legal risks and presented a single point of failure.

Furthermore, Chaumian ecash systems focus on the model of withdrawing ecash from *accounts* and depositing it into other accounts. This requires an undesirable setup phase for users.

Scrit removes the notion of accounts, it only has direct DBC-to-DBC transactions. Users do not have any standing relationship with the operators nor do

they possess any identifying authentication credentials. This both simplifies the system and removes a potential lever for censorship.

A classical Chaumian ecash system encodes the attributes of a DBC (for example, amount, denomination, and expiry) in the signed message of the DBCs (see Chaum, Fiat, and Naor 1990). Since the client controls the message, this poses a fraud risk that requires complex mitigation, which usually involved using either the user's identity or the user's holdings in his account as a collateral. In Scrit this fraud risk is removed by using the mint's signing key as the signifier of certificate attributes, similar to later implementations of ecash (see Schoenmakers 1998). That is, each DBC signing key (from the mint) is associated to a unique tuple comprised of amount, denomination, and expiry. A successful verification of a signature yields this tuple, the message contents are not authoritative concerning the value of a DBC. Since this removes the fraud risk in Scrit, no identification or account is necessary.

Scrit enables technical and legal distribution of DBC operations by parallel execution of transactions distributed over many separate mints. To accomplish this, we modify the classical construction of a DBC, which is composed of a message and a signature, and replace it with the definition of a DBC as consisting of a *unique* message and a *set* of signatures. Instead of relying on a unique value certified by a single mint, Scrit defines certification as consensus between mints expressed by independent mint signatures. The consensus is reached if a DBC carries enough signatures by different mints to reach a predefined *quorum*. That is, a DBC is valid, if it has at least m-of-n signatures, where m is the quorum and n is the number of mints (as described in detail further below).

Since Scrit operations are distributed over a set of mints, the question of governance arises. Technically, the solution of the governance question is outside the scope of the payment system Scrit itself, but we propose a simple governance solution based on Codechain, a system for secure multiparty code reviews, which is described in detail in the section on Governance.

Transactions in Scrit are extremely cheap and fast, especially compared to blockchain based systems. Mints do not have to synchronize at all to process transactions, which means the communication to all mints can be performed in parallel. This leads to network latency bound settlement times with sub-second confirmations. See the section on Performance for details.

2 Overview

Scrit serves the purpose of transferring value between users by employing a federation of third parties called mints. Value in Scrit is represented as a *digital bearer certificate* (DBC) which consists of a single-use unique message that is digitally signed by the mints, see section on DBCs.

To transact value the user sends a signed input DBC and a new output DBC message to the mints who record the signed input DBC as spent (in a database called *spendbook*, see section on Spendbook entries), sign the new output DBC message, and return it to the user. See section on Transactions.

For a transaction to be successful it has to be executed with a majority of all mints, see section on Quorum.

To control who can transact a DBC, Scrit employs digital signatures which public key is encoded in the DBC message. Transactions have to be signed by these keys. See section on Access Control Script (ACS).

Membership in the mint federation as well as authorized mint signature keys are contained in a *key list* that is coordinated by a *governance* layer. See sections on Key list and Governance.

Keys are only valid during their *signing epoch*, see section on Key rotation.

3 DBCs

DBCs are single-use digital coins in predefined denominations. The denomination, expiry, and currency of these coins are encoded by public signing keys employed by *mints* (the issuers of DBCs in Scrit). A

signature done by a mint guarantees the authenticity of a DBC. The spendbook of a mint guarantees uniqueness (and thereby prevents double spends).

DBCs consist of a message and a list of signatures. The message contains information for looking up signature public keys as well as information to enforce ownership and uniqueness. Values for key lookup are start of the signing epoch, amount, currency, and expiry, as well as signature algorithm. They refer to an entry in the *key list* (see Key list below). Furthermore, the ownership is encoded by a hash of an *access control script* (ACS) with which the mint verifies the user's authority to execute a transaction. The message also contains a random value for uniqueness. The list of signatures consists of at most one signature per mint in the network. Signatures contain the mint ID in addition to the cryptographic values of the signature itself.

Given the fields contained in the DBC (amount, currency, expiry, signature algorithm, and mint ID) the signing public key can be looked up from the system-wide published key list. Given the retrieved public key, the signature can be verified. This ensures that the values set in the message of the DBC match the signing key of the mint (otherwise the signature would be invalid).

3.1 Key list

Each mint publishes a list of its DBC signing keys. Per signing key the list contains the following information: amount, currency, signature algorithm, beginning and end of the signing epoch, the end of the validation epoch, and the corresponding unique public key. All entries are signed as a unit by both the long-term identity signature key and each unique DBC signing key contained in the list. This ensures that the private key corresponding to each public key contained in the list is actually controlled by the mint identified by the long-term identity signature key, which prevents the creation of forged DBCs. The association between certification values and the DBC signing key must be globally unique (which has to be verified by all clients and mints in the system). Without unique DBC signing keys it becomes impossible to count mint signatures. This can lead to faulty signature sets that yield an invalid DBC (a key shared between multiple mints) or to fraudulent certification of DBC properties (a key used to certify more than one set of properties).

4 Transactions

Scrit mints offer only three API calls to the Scrit clients: Perform a *transaction* (also called a *reissue*), a lookup in the spendbook (which records all spent DBCs), and one for retrieving the number of currently valid DBCs for a mint's signature key (to assess anonymity set sizes).

The spendbook writes entries in the order given below and aborts transaction processing when encountering a failure. All writes are successful if the value was not contained in the spendbook before and fail if the value is already known. A transaction works as follows:

1. Verify transaction: Verify ACS, verify mint signatures on input DBCs.
2. Test if transaction has already been added to spendbook, if yes return success and sign output DBCs.
3. Write server parameters to spendbook in the order contained in the transaction (if required for signature algorithm). If any parameter is known return failure and abort transaction (on first known parameter).
4. Write input DBCs to spendbook in the order contained in the transaction. If any input DBC is known return failure and abort transaction (on first known input DBC).
5. Write transaction hash to spendbook.
6. Sign output DBCs and return signature.

If a transaction contains any spent input DBCs after unspent input DBCs, the unspent DBCs will be recorded as spent and the transaction will abort without returning DBC signatures. This can only happen, if the client attempts to defraud the mint (or the implementer screwed up).

4.1 Transaction format

All transactions in Scrit are *reissue*-transactions. They take input DBCs and output DBC messages as well as parameters as input, and return output DBC signatures. Furthermore, transactions must fulfill conditions defined in the ACS referenced by the input DBCs.

Transactions consist of three blocks: A global set of input parameters, a global list of user signatures, and a mint local set of input parameters.

The global set of input parameters is sent to all mints and contains the following (see Figure 1):

- Start of the signing epoch, which refers to start of the key rotation epoch and must be globally coordinated between mints.
- Input DBCs: List of unblinded DBC messages, not including mint signatures.
- Root of parameter tree (see section Parameter tree below).
- List of access control scripts in the order of input DBCs.

The global list of user signatures contains one entry per input DBC that consists of all signatures that are required to fulfill the corresponding ACS. Each signature signs the corresponding hash of the input DBC and the hash of the global set of input parameters.

The mint local set of input parameters that is sent to a single mint contains only a list of lists of mint signatures and the corresponding path of the parameter tree (including the leaf), see section Parameter tree below. The list of lists has the same order as the input DBCs and contains the lists of the input DBC mint signatures. Usually such a list contains only the mint's own signature. Except in cases of mint recovery, see the section on Distribution below.

This transaction format limits the amount of signatures a client has to make, so that it does not depend on the number n of mints in the system. Furthermore, it simplifies the implementation of verification functions, because it only requires the parallel verification of list elements while limiting the impact of n on the required memory.

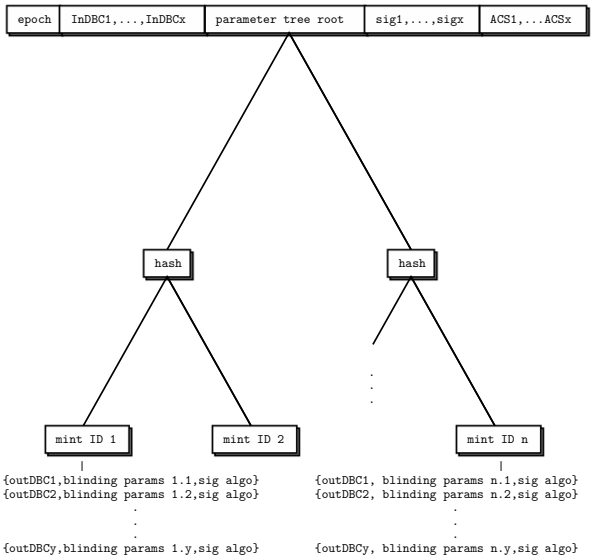

Figure 1: Transaction format (global set).

4.2 Parameter tree

The parameter tree contains per mint specific definitions of output. Each leaf is assigned to one mint and contains the mint ID and a list of tuples. A tuple contains a potentially blinded output DBC message, encrypted server blinding parameters (see section on Signatures below), and the signing algorithm to use. Furthermore, it contains values required for DBC signing key lookup (amount and denomination).

For non-blind signing algorithms the server blinding parameters are empty. If only non-blind signing algorithms are used in the outputs, the same leaf is revealed to all mints and the mint ID is set to a global constant referring to all mints.

The tree is encoded as a Merkle tree. During transactions leaf and path are revealed to the corresponding mint and are verified by it.

4.3 Spendbook entries

To enforce the uniqueness of DBCs (preventing double spend) each mint records the message of every spent DBC within one key verification epoch (that is, signing plus validation epoch). Furthermore, server blinding parameters have to be unique as well, which requires them to be recorded in the spendbook. In addition, recording the transaction itself allows idempotent operations.

For the spendbook Scrit uses a key-value store in which the following is recorded ('||' denotes the concatenation of values, '$a \rightarrow b$', the mapping of key a to value b):

- Transaction: T||Hash(Tx) $\rightarrow$ true
- Parameters: P||Hash(Param) $\rightarrow$ true
- DBC: D||Hash(DBC msg) $\rightarrow$ user sig||OOB

OOB refers to *out-of-band* data that can be generated by an Access Control Script (ACS).

The spendbook also records DBC messages in a hash chain for cryptographically secure ordering. The hash chain consists of:

$$CE_{n+1} = n + 1||\text{Date}||\text{Hash}(CE_n)||\text{Hash}(\text{DBC msg})$$

Clients can access all of these records through an open API.

4.4 Access Control Script (ACS)

Scrit mints enforce access control for DBCs through a parameter encoded in the DBCs which is called the *access control script* (ACS). Such an ACS can enforce that transactions using a certain DBC have to be signed by a user-controlled key. We define multiple access control languages which can be extended in the future to incorporate additional features.

Herein we define just two access control functions:

- $0x00$: No access control.
- $0x01$||Date||PubKey_a||PubKey_b: This ACS enforces that **before** Date the transaction must be signed by PubKey_a and **after and including** Date the transaction must be signed by PubKey_b. The special value 0 for Date enforces that PubKey_a must always sign the transaction.
- $0x02$–$0x$ff: Reserved for future use.

The standard transaction from recipient to sender constructs the ACS as follows:

1. Given: Elliptic curve, generator G.
2. Sender knows from recipient:
$$\text{PubKey}_r : aG$$
3. Recipient knows corresponding:
$$\text{PrivKey}_r : a$$
4. Sender generates temporary key pair:
$$b = \text{random}, \text{PubKey}_b = bG$$
5. Sender calculates shared secret:
$$s = \text{Hash}(\text{scalarMult(b,aG)})$$
6. Sender calculates transaction signing key:
$$\text{PubKey}_a = \text{scalarMult(s,aG)}$$
7. Sender constructs ACS as:
$$0x01||\text{Date}||\text{PubKey}_a||\text{PubKey}_b$$
8. Recipient calculates shared secret:
$$s = \text{Hash}(\text{scalarMult(a,bG)})$$
9. Recipient calculates signing key:
$$\text{PrivKey}_a = as$$
10. Recipient signs transaction.

If the recipient doesn't sign a valid transaction before Date expires, the sender can recover the DBC by signing a transaction with b (which he has to store).

The above construction prevents the mints from recognizing the recipient over multiple transactions and

thus preserves the anonymity of both sender and recipient. That is, the recipient PubKey_r can be a published constant without sacrificing anonymity.

Referenced state in an ACS refers to the mint's local state. In the case of Date this is the system time of the mint in UTC.

4.5 Evidence of payment

The combination of a publicly accessible spendbook that contains the user signatures of spent DBCs and the access control script allows for a sender to publicly demonstrate that he made a payment that was accepted by the recipient. Neither the sender nor the mint are able to forge this signature. This also allows the owner of a DBC to demonstrate if a mint has falsely claimed a DBC to be spent.

4.6 Protocol flow

Let's assume the sender has a DBC A for a given recipient constructed according to an ACS type $0x01$ as described above.

In order to perform a payment the protocol flow is as follows. The sender gives the DBC A to the recipient. The recipient reissues the DBC A to a DBC B, either immediately or before the ACS Date expires:

1. The recipient constructs a transaction with DBC A as input DBC, DBC B as output DBC, and signs it with the derived PrivKey_a.
2. The recipient talks to all n mints **in parallel**, sending **each** mint the same global set of input parameters of the constructed transaction, but sending each mint a **different** mint local set of input parameters (as described in the Transactions section above).
3. Each mint verifies the transaction independently of all other mints, signs the output DBC, and returns its signature.
4. The recipient collects the signatures from all mints over the output DBC B, combines them into a validly signed DBC B (given he received at least m valid signatures), and saves it in his wallet.

The protocol protocol flow for an ACS type $0x00$ is similar, but simpler, as shown in Figure 2.

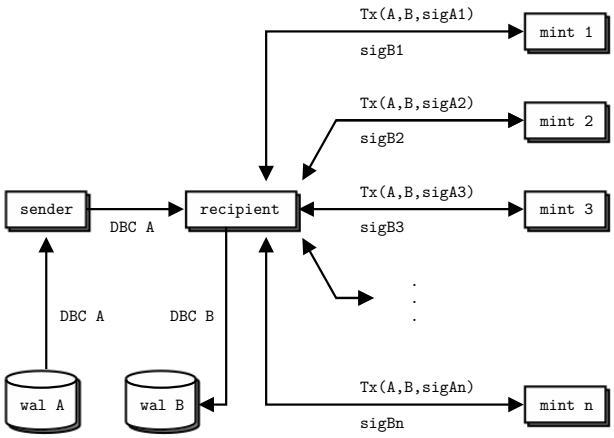

Figure 2: Protocol flow of a Scrit transaction with an ACS type $0x00$. Scrit clients talk to all mints in parallel.

5 Signatures

Scrit employs both blind and unlinkable as well as non-blind signature schemes.

The non-blind signature schemes are used for user signatures to fulfill access control scripts as well as for DBC signatures in scenarios where unlinkability of transactions is not a requirement.

For anonymous blind signatures that allow anonymous and untraceable transactions we use a ECC based blind signature scheme published by Singh and Das (2014). This scheme is based on ECDSA and employs user-generated blinding parameters as well as a single-use server-generated blinding parameter set consisting of Q and K. The server blinding parameters serve to protect the private key of the signer against attacks by the user. For the security of this scheme to hold these

parameters may not be reused. This complicates the mint operation in the sense that blinding parameters have to be recorded in the spendbook and have to be exchanged partially with the user. Scrit solves this issue by the mint encrypting K to its own temporary symmetric key and sending the encrypted K and the unencrypted Q to the user. During the transaction Q and the encrypted K are send back to the mint and the mint decrypts K, verifies it against the spendbook, and on success generates one signature.

While the unblind signature scheme provides anonymity to the users of the system it still allows the creation of a history of DBC transaction. Only by using the blind and unlinkable signature scheme does it become possible to preserve untraceability and thus increase the anonymity set of all DBCs to at least the DBCs issued during one signing epoch.

6 Key rotation

To be able to prune the spendbook and not having to keep signing keys secret forever, Scrit employs *key rotation* with disjunct signing epochs. The signing epoch determines which signing key is used at a certain point in time. After the end of a signing epoch follows a validation epoch in which DBCs can still be spend. Together the combination of signing and validation epoch comprise the verification epoch. Figure 3. visualizes the key rotation process.

All mints have their own singing keys, but the epochs are the same for all and have to be synchronized (see section on Governance).

Employing key rotation has two important implication:

1. Clients must go online before the verification epoch of the DBCs they hold ends and reissue them. Otherwise they will loose these DBCs.
2. After a validation epoch ended the total number of DBCs in circulation can be calculated with a spendbook audit and compared between the mints.

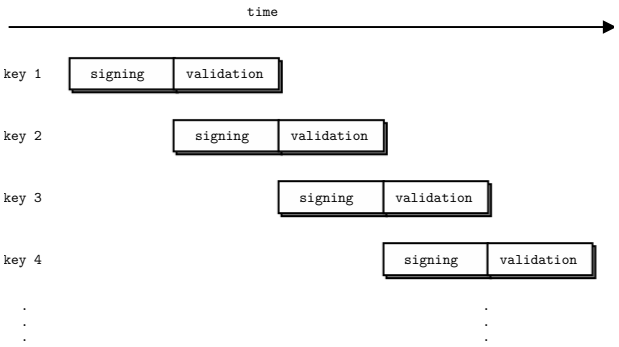

Figure 3: Key rotation with disjunct signing epochs.

7 Distribution

Scrit defines a valid DBC as a message signed by at least m-of-n mints that is **not** recorded in the mints' spendbook.

From this follows that a mint shall reissue a DBC if any of the following two rules is satisfied:

1. A DBC message is not found in the spendbook and the DBC message is signed by the mint itself.
2. A DBC message is not found in the spendbook and the DBC messaged is signed by at least m-of-n other mints.

For these rules to be sufficient the signing public keys for case 2. must belong to the same epoch and the signing keys must be globally unique. Furthermore, this requires that only one signing key per mint and per epoch exists.

m and n can therefore only change at the start of a signing epoch. However, n, the total number of mints, can be decreased at any time as long as it stays equal or greater than m. Signing epoch lengths can be changed, but must stay disjunct, meaning signing epochs may never overlap.

The signing epoch for output DBCs is enforced by the epoch field in the transaction so that no decisional ambiguity exists at signing epoch boundaries.

During normal operations a user only has to fulfill

rule 1. by sending one signature to the corresponding mint per reissue transaction. Only if not all signatures of all mints can be collected for a DBC it may become necessary to employ rule 2. in a subsequent transaction. Users' clients should always try to complete the set of signatures, otherwise failures of mints can cascade and invalidate DBCs due to a lack of signatures.

Transactions employing rule 2. are called *recovery transactions*. Accordingly, transactions employing rule 1., which is the normal case, are also called *non-recovery transactions*.

Temporary or permanent unavailability of single mints, as long as the quorum remains fulfilled, does not undermine the ability of Scrit to perform transactions. The later addition of new mints and the ability to add them to the quorum allows enough dynamism for a Scrit network to heal.

New mints do not have to know any spendbook of other mints in order to participate in the network. Only knowledge of the public signing keys is required.

Scrit is based on a quorum of mints certifying the validity of DBCs to users and other mints. This allows mints without prior knowledge of a DBC to accept it as valid as long as it is signed by enough other mints. However, this only holds as long as there is an upper bound of mints that are **changed** during the verification epoch of a DBC: No more than $x = 2m - n - 1$ mints may be replaced, added, or removed during that epoch to prevent a user from forging a DBC.

To simplify corner cases of verification in actual implementations we set the rule that a verification period may not be longer than the next signing period.

Changes to m are unproblematic because in transactions the m refers to input DBCs of the same or previous epoch while the m in the output DBCS always refers to the current signing epoch. However m must always be larger than $n/2$, as explained in section Quorum.

Given these constraints it is advisable that m is significantly larger than $n/2$ to have more flexibility with changes of n.

7.1 Change of monetary supply

Increasing the monetary supply consists of issuing new DBCs without spending existing DBCs. For this to be possible mints have to coordinate the DBC message to be signed. This prevents lower than quorum colluding mints from increasing the monetary supply.

Reducing the monetary supply requires the spending of DBCs without issuing new ones. For this DBCs can simply be reissued to an ACS that is provably unusable. These DBCs fall out of the circulation at the end of their verification epoch, effectively reducing the monetary supply.

7.2 Quorum

As mentioned above DBCs in Scrit contain multiple signatures and are valid if the number of signatures is equal to or greater than the quorum. The quorum has to be larger than $n/2$, since any quorum lower than the majority of mints would allow the user to multiply DBCs. The upper bound of the quorum is only limited by the resilience against mint failure that is acceptable. The higher the quorum, the more mints would have to collude in order to to allow dishonest behavior. The lower the quorum, the more mints can fail during operation without interrupting the system.

We suggest a minimum of $n = 10$ mints combined with a $m = 8$ quorum.

8 Governance

As mentioned before, the mints do not have to talk to each other to perform normal transactions (which are *reissue* = *spend* + *issue* operations). Either an unspent input DBC is presented to them with their **own** signature or an input DBC with enough signatures of **other** mints, such that the signatures reach quorum. The former is the normal case. The latter

can happen when a mint wasn't reachable during an earlier reissue operation or simply didn't exist yet.

However, this still leaves a few questions open: How is new money introduced into the system and the (optional) backing (a pure *issue* operation of new DBCs)? How is money removed from the system and the (optional) backing (a pure *spend* operation of DBCs)? How are new mints introduced into the system or existing ones removed from it? Under what rules do the mints operate? How does a client get to know which mints belong to the system and which changes are made, ideally in an automatic and cryptographically secure fashion? In short: How do we solve the problem of *governance*?

Scrit uses Codechain[1] as its governance layer. Codechain is a system for secure multiparty code reviews which establishes code trust via multi-party reviews recorded in unmodifiable hash chains. This makes it impossible for a single developer to add changes to the Scrit code base. Using Codechain tends to be a good idea for sensitive code like the Scrit client or the Scrit mint, but it is probably less clear how it could solve the governance problem. For the client and the mint the signers of the Scrit Codechain are the trusted Scrit developers.

To understand how Codechain can solve the governance problem three points are important:

1. A Codechain "repository" doesn't have to contain source code, although that is the most common use case. It could also just contain configuration data and text files.

2. The set of signers of a Codechain doesn't have to be the group of developers. It could also be another group, such as all the mints.

3. Codechain contains a mechanism called *secure dependencies* (see the specification[2] of *secure packages* for details) that allows to embed one Codechain into another, with potentially different sets of signers.

[1] https://github.com/frankbraun/codechain

[2] https://godoc.org/github.com/frankbraun/codechain/secpkg

We can combine these three points into a governance solution for Codechain:

- We have a "normal" Codechain for the Scrit client and mint, signed by multiple Scrit developers.
- We have a "governance" Codechain which contains configuration files and text files, comprising the governance layer of Codechain. The set of signers are all the mints (the number n of signers in Codechain) in the system and they "vote" on changes in the governance layer by signing changes to the "governance" Codechain. The necessary quorum (the minimum number of signatures m in Codechain) can be the same as the quorum for transactions or higher. Of course, the transaction quorum is also set in the governance Codechain. The configuration files contain all the mints which comprise the system, how they can be reached, what their signature keys are, and what the monetary supply is. Decisions to add miners, remove them, or change the monetary supply are recorded in the governance Codechain and are voted on by mints signing. The entire process is described in a "constitution" text file which is also part of the governance Codechain and is changed by the same mechanism.
- The "normal" Codechain for the Scrit client and the mint contains the "governance" Codechain as a secure dependency. That way the client and the mint can automatically and securely update the mint configuration, allowing to transparently add and remove mints from the system.

The whole design gives us a simple solution to the governance problem in Scrit:

- For normal operations (that is, transactions) the mints do not have to talk to each other at all, everything is done **automatically** by the clients talking to all mints separately (but in parallel).
- Governance change are decided on **manually** by the mint operator via signing changes to their governance Codechain, but are then **automatically** distributed to the corresponding Scrit clients and mints via secure dependency updates, as they are happening during regular secure package updates of the client or mint code.

The governance Codechain contains a definition file that includes start and lengths of epochs, the mint identity keys, the key lists, and a commitment of future DBC creation or destruction.

9 Wallets

Scrit wallets work differently than other cryptocurrency wallets, because they mostly revolve around transfer and reissuing of DBCs, and they don't necessarily have to sign anything. In the following we give some details on how mobile and hardware wallets for Scrit could work.

There are four connectivity scenarios to consider:

1. Sender and recipient are both online.
2. Sender is online and recipient is offline.
3. Sender is offline and recipient is online.
4. Sender and recipient are both offline.

9.1 Mobile wallets

We consider having a mobile wallet as a sender and a mobile wallet or POS terminal as recipient.

In scenario 1. (both online) the sender scans a QR code from the recipient containing the payment sum, the DBC public key of the recipient, and a URL where to upload the payment DBCs. The sender reissues the necessary DBC to reach the payment sum for the recipient's public key, creating assigned DBCs. He then posts these to the URL. The recipient checks locally that he hasn't seen these DBCs before (to prevent double spends) and reissues them again (possibly later). This gives the sender Evidence of payment, as described above.

In scenario 2. (only sender online) the sender scans a QR code from the recipient containing the payment sum, the DBC public key of the recipient, and configuration data for a local Bluetooth or WiFi connection to the recipient. The sender reissues the necessary DBC to reach the payment sum for the recipient's public key, creating assigned DBCs. He then opens up a local Bluetooth or WiFi connection to transfer them to the recipient. The recipient checks locally that he hasn't seen these DBCs before (to prevent double spends) and later reissues them. This gives the sender Evidence of payment, as described above.

In scenario 3. (only recipient online) the sender scans a QR code from the recipient containing the payment sum, the DBC public key of the recipient, and configuration data for a local Bluetooth or WiFi connection to the recipient. The sender opens up a local Bluetooth or WiFi connection to transfer unassigned DBCs to the recipient. The recipient immediately reissues them to prevent double spends. The recipient confirms the payment, however this does **not** give the sender Evidence of payment.

In scenario 4. (both offline) the sender scans a QR code from the recipient containing the payment sum, the DBC public key of the recipient, and configuration data for a local Bluetooth or WiFi connection to the recipient. The sender opens up a local Bluetooth or WiFi connection to transfer **previously assigned** DBCs to the recipient. The recipient checks locally that he hasn't seen these DBCs before (to prevent a double spends) and confirms the payments. This gives the sender evidence of payment, as described above, but only **after** the recipient reissued the DBC at a later stage.

In theory the transfer from the sender to the recipient could also be done via QR codes. But with a larger number of DBCs and/or mints this quickly reaches the size limitations of QR codes and is therefore not realistic in practice.

However, QR codes might be a good way to transfer a bunch of assigned DBCs to a recipient on paper, with the recipient scanning one DBC QR code after another. This gives us offline anonymous untraceable digital cash in paper form (assigned to a single recipient).

9.2 Hardware wallets

A simple hardware wallet would consist of a mass storage device, a display, and a single button. It basically handles scenario 3. (only recipient online) or 4.

(both offline) above. The mass storage device contains DBCs in different denominations. When connecting the hardware wallet to the POS terminal (via USB, NFC, or other means) the POS terminal requests a certain sum. The hardware wallet shows the requested sum on the display and waits for confirmation via a button press. Upon confirmation the hardware wallet would select the corresponding DBCs, transfer them to the recipient, and delete them. Depending on the scenario, the recipient would either reissue immediately (for unassigned DBCs) or later (for assigned ones).

Since a very simple hardware wallet cannot check the validity of DBCs we do not deal with change. When loading up hardware wallets the denominations are selected in a way to err on the side of smaller denominations and we can live with small overpayments in almost all real world payment situations (consider it a tip).

Such simple hardware wallets would be loaded with a trusted device. For example, an ATM that we trust (just as we trust cash ATMs) or with a desktop client running on a trusted computer.

10 Communication

Scrit does not define the means of communication between users. Sets of DBCs are simple strings that can be transferred between users by email, instant messaging, or any other means. They can also be printed as QR codes and used by the sender without a digital device present.

Communication for transactions between users and mints can be very efficient if the signature set required is small. For non-recovery transactions a single UDP package can usually contain the whole transaction. Since the system is idempotent no communication guarantees are required. On communication failure the transaction is simply repeated with the failing mint. To conceal the content of transactions against third parties the transaction package is encrypted to the mint's long-term and short-term public encryption keys.

Using single packet UDP requests and responses with encrypted payloads allows the development of censorship resilient mint networks.

For recovery or large transactions communication to the mint the client uses TLS over TCP.

Furthermore, the transaction format allows the relaying of transactions through active trustless proxies that efficiently distribute the transactions over all mints.

11 Performance

A usual transaction consists of the following operations:

1. Signature verification by the sender.
2. Signature generation by the recipient.
3. Two signature verifications by the mint.
4. Three spendbook operations by the mint.
5. One signature by the mint.
6. Signature verification by the recipient.

Hence the performance of Scrit is limited by two signature verifications, one signature creation, and three spendbook operations performed by the mint. These operations are easily distributable over multiple processors and hosts. Sharding of the spendbook can easily happen without complex commitment and synchronization schemes since spendbook operations are designed as failure on first known entry.

Current server-grade hardware can perform several thousand signing and verification operations and several hundred thousand spendbook operations per second.

Since single mints can easily be distributed over clusters of hardware and mints do not have to synchronize during transactions this system is linearly scalable as long as mints scale equally.

This allows for the creation of mints which have an upper transaction volume bound by connection bandwidth.

12 Backing

While Scrit does not define a backing layer, a potential one is a backing of mint payment infrastructure by Bitcoin as soon as efficient multi-signature algorithms (for example, Schnorr signatures) are implemented. This would allow to extend the control quorum from Scrit mints to their backing. At this point in time a Bitcoin backing is already possible with multi-signature addresses, but this would limit n to 15, because that is the current maximum for m-of-n multi-signature addresses in Bitcoin.

It is also reasonable to envision backing by fiat money, precious metals, or any other valuables. Enforcing sound backing operations is outside the scope of Scrit itself. It is also possible to operate a Scrit mint network without any backing at all.

12.1 Cartel theory

Cartels are colluding groups of system participants that conspire and coordinate to undermine the rules of the system. Cartels do not come into existence completely formed, but require communication and negotiation before they can become effective. During this time a cartel does not pose a threat to the system yet. Both during formation and operation a cartel is vulnerable to members that commit treason against it.

The lesson drawn from this has been to incentivize traitors against the cartel in order to make cartels more brittle and potentially undermine their formation.

One method to do this is to reward the traitor with collected penalties from other cartel members. It is conceivable that Scrit mint operators have to deposit a security which would be transferred to the traitor if he can provide evidence of the formation of a cartel.

In such a system, the n mints are divided into groups of size $g = n - m$ and each mint distributes its security equally over all of these groups. The funds of each of these groups is controlled by a $(g-1)$-of-g multi-signature address. As soon as a mint can present evidence of another mint's attempt to form a cartel, that mint's security is distributed to the witness while the cartel forming mint is excluded from n. The g mints of every group judge the evidence.

This method would allow to create an pseudonymous mint network. The only way to create a cartel in such a pseudonymous mint network is to sign the relevant communication which also creates proof of cartel forming activity. Unauthenticated communication towards this goal is indistinguishable from a member being tested towards his inclination for cartel membership. However, this method does not prevent Sybil attacks.

13 Conclusion

Scrit is a very fast, extremely cheap, and linearly scalable distributed untraceable electronic cash system with a flexible backing and governance structure. Employing a blind and unlinkable signature scheme for its DBCs makes Scrit censorship resistant against rogue mints. Its network communication protocol, with very short encrypted packages that is proxy capable, makes it censorship resistant against network filters.

It is not permissionless for mint operators (a new mint requires the permission of the quorum m of the existing mints to join), but it can be combined with permissionless backing like Bitcoin, creating a second-layer solution with very interesting properties, that make it a good fit for user-to-machine and machine-to-machine (micro-)payments.

Its trust model makes Scrit very suitable as a value transfer system, but it should not be viewed as a long-term store of value.

Scrit revives the concept of Chaumian ecash and adds federation to it, mitigating issuer risk. It allows to perform simple offline payments, a feature that to the best of our knowledge no other digital payment system has.

References

Chaum, D., A. Fiat, and M. Naor. 1990. "Untraceable Electronic Cash." In *Proceedings on Advances in Cryptology*, 319–27. CRYPTO '88. Berlin, Heidelberg: Springer-Verlag. http://dl.acm.org/citation.cfm?id=88314.88969.

Schoenmakers, Berry. 1998. "Basic Security of the eCash Payment System." In *State of the Art in Applied Cryptography: Course on Computer Security and Industrial Cryptography, Leuven, Belgium, June 3–6, 1997 Revised Lectures*, edited by V. Rijmen B. Preneel. Volume 1528 of Lecture Notes in Computer Science, Berlin.

Singh, Nitu, and Sumanjit Das. 2014. "Article: A Novel Proficient Blind Signature Scheme Using Ecc." *IJCA Proceedings on International Conference on Emergent Trends in Computing and Communication (ETCC-2014)* ETCC (1): 66–72. https://www.ijcaonline.org/proceedings/etcc/number1/17905-1417.