Digital Money and DBCs

Jonathan “smuggler” Logan

2018-11-12T19:20:33Z

This article as PDF[1].

It’s been more than a decade since I got involved in the design and development of digital payment systems. My focus has always been primarily that of a cryptoanarchist - privacy and functionality first, regulatory compliance second. Since I got involved in this field, a major innovation took place: Blockchain cryptocurrencies. This innovation injected hundreds of thousands of new people into the realm of cypherpunk ideas and online “less regulated” commerce. Thousands of new developers are now working on new tools and I applaud that. However, in engaging with many of these new faces I discovered that their understanding of what other technologies exist - or have existed - beyond the blockchain is wanting. In almost every conversation I find out that terms like “digital bearer certificate”, “verifiable book money”, “blind signatures” and a host of others are virtually unheard of. The generation change from pre-blockchain age to blockchain age has come with a lot of old knowledge being lost. But in my opinion it is knowledge that is still useful, and possibly even crucial. Hence I decided to write this text, touching upon some of these forgotten technologies, and specifically recording my own contributions in the field. May it help everybody to widen their box of thought and find inspiration for better solutions, and to gain a better understanding of the problems to be solved.

Digital Money

Let’s start with defining “digital money”. Digital money is the digital representation of money, which is in turn a medium of value transfer and storage. Digital money allows making payments and to keep value for a time. Digital money can represent currencies like the US Dollar or the EURO, or precious metals like gold or silver. A “digital currency” is something different: Instead of representing some other money, digital currencies themselves have a monetary function as in being able to create and regulate the supply of money. Digital currencies are digital money plus monetary control.

[1] /files/DigitalMoneyDBC.pdf

Digital Book Money

The most basic form of digital money is "digital book money", commonly known under the term "e-money" or "electronic money". This includes most of the payment systems we use today, from giro money on your bank account, to PayPal, credit cards and various internet payment systems. Digital book money works by the operator keeping a record of accounts and balances that are controlled by the users of the system. In the simplest form this is a simple table stating which users has how much money in their account.

Digital book money has a bad reputation in some circles because it does nothing to prevent operators from manipulating the money kept, invent it, steal it, or make transfers without user consent. This however is not a fair description of more advanced digital book money systems. More modern systems are verifiable and auditable. As an example, systems that derive from OpenTransactions implement mutually signed transactions and balances which allow both the operator as well as the user to present a cryptographically signed, fixed order, immutable history of transactions and the balances they result in. This works by each account keeping the hash of the last transaction, balance, and signatures by both user and operator of these values. Each new transaction includes the hash of the last transaction, the balance before the transaction is executed, the balance after the transaction is executed, additional transaction details (like the destination account of value transaction and its amount), and the signatures of both the operator and the user (account owner). This way both the operator as well as the user have a signed statement showing the state of an account which can be demonstrated to a third party in case of suspected abuse. Further developments in that sphere concern the auditability of the sum of all balances in a system by forming a merkle tree out of the accounts and their balances as the leaves, and then taking sums in the nodes of the tree. That merkle tree is then added into a hash chain and the head of that chain published. Now auditors that know the signed statements of some accounts can verify that the published accounting is probable true, and that the operator is not inventing or stealing money. Any account holder can verify that public accounting for himself, and publish a proof of any discrepancy if such exists. There have also been discussions about using bullet proofs to the same end. Taking these modern developments into account, digital book money systems present a very efficient, extremely cheap to operate, and publicly verifiable means to provide infrastructure for person-to-person transfers. They do however provide no privacy nor are they permission-less.

Blockchain Cryptocurrencies

Instead of operating a single centralized record of accounts and balances that is modified by a single entity, blockchain cryptocurrencies replicate those records

across many entities of unknown trustworthiness. Changes to the records are executed by those entities by applying new transactions to the current state of record, while adhering to a protocol that defines validity of transactions and access control to records (usually through the use of digital signatures). A complete history of valid transactions is distributed over all entities taking part in operating a blockchain cryptocurrency, allowing each to calculate the current state of records for the system.

The order in which transactions are processed is a crucial element of any system that changes global state. For systems operated by a single entity, that entity can define and assure order easily on a first come, first served basis. In a distributed system of untrusted parties, additional efforts must be taken to ensure that all parties execute transactions in exactly the same order. To this end, the system must select an authoritative party (the "leader") that defines the order of transactions. In blockchain based systems, this party is only authoritative for a short amount of time and replaced with an unpredictable other party for the next time slot. Various methods for leader selection exist. Some systems use election protocols, others use proof-of-stake or proof-of-work protocols. Common to all of these methods is that they result in one leader being selected to define the order of transactions for a short amount of time, leading to the synchronization of transactions and state for the whole system. Additionally, blockchain cryptocurrencies have a money-creation function which usually works by rewarding the leader by allowing it to create some defined amount of new money in return for its work. It is important to note that the definition of which rules apply to verify transactions, select leaders, and create new money are purely defined in software and become valid only due to the consensus of participating parties.

Digital Bearer Certificates

The remainder of this text deals with "Digital Bearer Certificates" (DBC for short). I'll try to describe both the technology itself as well as the history of DBCs as experienced by me.

A DBC is nothing else but a string of bits that are ascribed value. It directly represents the money, without reference to any account, instead it is the money itself. Consider a DBC to be a digital bank note that can be directly transmitted between parties, does not require any account to be linked to a user, nor a widely distributed global state of the system.

For this to work, a DBC needs to fulfill two requirements:

First, it needs to be verifiable as being a DBC issued by some party. This is usually done by the issuer signing the DBC with a key. Second, it needs to be unique. A string of bits can be copied without limitations, but only one of those

copies carries value. This is achieved by being able to exchange one DBC for another at the issuer, and recording the original DBC as being de-validated.

A typical DBC encodes the currency or asset in which it is denominated (the "Denomination"), the number of units of that denomination that it represents (the "Amount"), a random nonce, and a signature by the issuer.

A typical DBC system consists of two types of entities: The Mint, which issues the DBCs. It must be able to verify that DBCs presented to a mint have been issued by that mint, which is usually done by verifying the signature. It furthermore has to be able to ensure the uniqueness of a DBC, usually by keeping a record of DBCs that have been de-validated already. The record of de-validated DBCs is usually called "Spendbook". And the User that interacts with the Mint and other users by transmitting DBCs.

In a typical transaction, user A would send a DBC 1 to user B, user B would then communicate with the mint to exchange DBC 1 into a new DBC 2. The mint recording DBC 1 as being de-validated. Should user A try to exchange DBC 1 at the mint at a later moment, the mint would recognize it as de-validated, refusing the replacement. This basic protocol ensures that a DBC cannot be used more than once in a transaction.

DBC systems typically are extremely cheap to operate and allow very fast finalization of transactions, as well as allowing many transactions to be processed in a short time. Typically the time to finalization of a transaction only barely exceeds the network latency between the user and the mint, and mints running on commodity laptop hardware can typically deal with thousands of transactions per second. The cost of operation is determined mostly by once creating and once verifying a digital signature per DBC, storing a unique identifier for each DBC (around 32 bytes per DBC), and transmitting the DBC over the network twice (a DBC usually being smaller than 250 bytes). This makes DBC systems prime candidates for extremely fast and cheap micro payments.

Let's go into a little more detail.

Spendbook

Each mint has to maintain a record of DBCs that have been de-validated so to assure uniqueness of DBCs and thus prevent duplicate spending of a DBC. The spendbook is the only element of a Mint that has to be tightly synchronized and access to it has to be strictly serialized. Failure at this can lead to race conditions which allow double spending of DBCs. Spendbooks can be easily distributed and sharded to increase reliability and transaction performance. Spendbooks only implement a single function:

bool Append_iff_unknown(value) : Append a value to a spendbook if and only if that value is not yet present in the spendbook. Return true if the value has been appended, false if the value has not been appended.

The value added to the spendbook is usually a cryptographic hash of a DBC.

Signatures

A mint must be able to verify that a DBC was issued by it. To this end, DBCs carry a digital signature by the mint. On being presented a DBC, the mint verifies the signature and only then tries to record it to the spendbook. Only if signature verification and spending (Append_iff_unkown) are successful then DBC has been valid. Early experimental mints used simple HMACs with a secret value for signatures. Later developments used RSA, DSA and ECDSA signatures with published public keys that would allow users to pre-verify a DBC before sending it to the mint.

Blind signatures

Blind signatures allow for the signer to sign data in such a way that the signature can be publicly verified, but without the signer knowing what data it signed. This involves a protocol in which the user "blinds" the data, sends the blinded data to the signer, the signer creates the signature, and then the user un-blinds the data and signature and provides it to a third party for verification. The signature over the original data would show as being validly signed by the signer, without the signer knowing at the time of signature what data it signed. Blind untraceable signatures furthermore allow a signer to sign data so that it will not be able to know what data it signed, nor be able to find out when the signer made the signature even if both the un-blinded data and the signature are known to the signer afterwards. This allows for the construction of mints that can issue DBCs in such a way that the mint has no way to link input and output DBCs, nor be able to determine in any way when it signed a DBC or for whom. To emphasize this: Blind signatures allow the construction of DBC systems that allow for fully anonymous and untraceable transactions, verifiable by the user, without the mint having to be honest to maintain anonymity and un-traceability. However, blind signatures make the mint vulnerable to be defrauded by users. The blinding and un-blinding steps above are controlled by the user alone, with the Mint not knowing if the encoded denomination and amount is valid. The way to secure a Mint against this was usually to have the user provide many different blinded DBC candidates for signing, then the mint randomly requires the user to prove the content of all but one of those candidates, and then signing the remaining DBC if all other candidates were valid. In case a user would be discovered presenting a fraudulent DBC candidate to the Mint, some punishment would be exercised by the mint.

Mint functions

Fundamentally a mint only implements two basic functions:

- Issue: Create a new DBC by encoding denomination, amount and a random nonce, and sign it with the mints key.
- Spend: Destroy a DBC by verifying the signature, and recording the DBC in the spendbook.

These basic functions are not directly exposed to users. Instead, typical Mints implement these public functions:

- Reissue: Spend a user provided DBC, and issue a new DBC on success. Return the new DBC to the user.
- Split: Spend a user provided DBC, and issue two or more new DBC so that the sum of amounts of the new DBCs equal the user provided DBC. Return the new DBCs to the user.
- Combine: Spend the user provided DBCs (plural!). Issue a new DBC so that its amount equals the sum of amounts of the DBCs provided by the user. Return the new DBC to the user.

More advanced implementations would only expose a "Reissue" function to the user which would be able to deal with any number of user provided DBCs and any number of output DBCs.

Examples from history

Yodelbank

This DBC system was available on an IRC network frequented by cypherpunks in the early 2000s. Users could directly send each other DBCs through IRC, and interact with the Mint through IRC as well. Yodelbank offered DBCs that presented value in a variety of digital gold currency systems. Users could buy and sell DBCs for egold, 1MDC, pecunix and several other currencies. Yodelbank encoded DBCs in a human-readable format and signed them with HMAC. This allowed users to transact without the need for any special software, just copying and pasting short lines of data.

Digicash

Invented and founded by David Chaum. It used blind signatures to provide untraceablity for users. Users would open an account with Digicash and withdraw value in the form of blindly signed DBCs. The user could then transfer DBCs to other users which could then pay those in to their account. Digicash only broke the traceability between the sending and receiving user, but did not provide

anonymity to use the system. It also required software on the user's computer to be used.

eCache

My involvement with DBC systems started with being exposed to the Yodelbank system when being a resident of the Invisible IRC Project where it ran. I wanted a system that was more powerful and functional, and solved some shortcomings of it. Hence I ventured into implementing my own DBC software. Though I wrote the software, I never operated a public Mint due to legal reasons. Others however used my software to operate small mints that facilitated transactions in closed user groups, and even a small public mint that never caught any traction. With the advent of blockchain cryptocurrencies, this public mint disappeared. eCache did however have some influence on other systems, such as some ideas being picked up again by Vouchersafe and OpenTransactions. Both of these systems originate from the same group of users that discussed DBC systems on IIP and are still operational and in active use. No eCache system is operational as far as I know. There have been no changes to the design after 2008, though some new ideas will be touched on at the end of this text.

Auditable and Verifiable DBC mints

A fundamental issue with DBC systems is that they are hard to protect against dishonest mints. A mint can issue DBCs beyond the actual money it has in backing, or falsely claim that a DBC has already been spent before when presented by the user. Solving this requires a few additions to the DBC design:

DBC Expiry

Recording every spent DBC not just requires storing potentially a lot of data over time, but also increase the resources spent on verifying if a DBC has already been spent. For this reason eCache DBCs expire at some point after which they are invalid. Every eCache DBC thus encodes a future date at which the DBC will become invalid if not reissued before. The default expiry date was at least one year in the future. The spendbook can now be implemented by an expiry-indexed set of bloom filters that can simply be deleted after the expiry time has been reached.

HSM Signatures

Only recording spent DBCs (in contrast to recording issued and spent DBCs) requires that the signing key of the Mint is kept very secure. eCache solved that

by implementing the signature verification and signing operation in an HSM (Hardware Security Module) that would furthermore keep track of input and output amounts of a transaction. Furthermore the signing key would be rotated every few weeks. For each key, eCache tracks the total amount of value signed with it, subtracting from that amount with every signature verification. That way the Mint can make sure that it is not presented with DBCs generated from a stolen signing key.

Ownership

eCache DBCs encode information about who can spend the DBC, that is: Who the owner is. This is simply a hash of a public signing key that is used to verify any transaction in which that DBC is used. Any transaction command from the user to the mint has to be signed by the private key that corresponds to the public key encoded in the DBC, otherwise the mint will not process the transaction. This allows the mint to record a proof that the DBC has been spent legitimately by its owner, and not destroyed by the mint without user consent. Furthermore the eCache DBC allows a second owner which becomes valid after a defined date which is also encoded in the DBC. This means that eCache DBCs record an owner that can spend them before a user defined date X, and another owner that can spend it afterwards. Owners are only identified by public key, for every DBC a new owner key pair is generated, and usually not reused. In a mint transaction, all input DBCs must be owned by the same owner public key. Combining multiple DBCs with different owners will require that input DBCs are first reissued to the same owner individually, and then combined in a later transaction.

Publicly verifiable Signatures

Standard eCache DBCs are signed using ECDSA with a public key that is known to users. This allows the pre-verification of DBCs by users before even talking to the Mint.

Signed Mint responses

Any response by the Mint to a user is signed so it can be verified by anybody that it actually came from the Mint.

Idempotent API

Any time the same parameters for a transaction are sent from a user to the mint, the exact same response would be returned. This prevents newly created DBCs to be lost, but it also allows for the user to publish his transaction request and

have it be verified by anybody. Combined with signatures done with ED25519 and DBC templates (see below) this can be implemented trivially by only caching the result branch of a transaction, indexed by the hash of the transaction input. The transaction cache expires after a week, giving ample time to question the honesty of the mint, or recover from untimely loss of connectivity.

User-provided DBC templates

Instead of creating and encoding DBCs itself, eCache mints require the user to provide it with a template for newly to be signed DBCs that only lack the mint's signature. The mint only verifies the templates to be valid in the context of the transaction, and signs them accordingly. For each transaction eCache allows one or two input DBCs, and one or two output DBCs, and thus implements the reissue, split and combine functions in one flexible reissue operation. In the case of two input DBCs, the transaction includes three branches of output DBC templates for three possible states of the input DBCs:

1. Both input DBCs are unspent and valid.
2. The first input DBC is unspent and valid, the second is not.
3. The second input DBC is unspent and valid, the first is not.

Constructing transactions like this allows for lock-free writing to the spendbook (which also allows for the easy distribution of the spendbook) without the requirement of rollbacks or introducing complexities that can introduce race conditions.

Concealed unique values in templates

Since transactions are publicly recorded for verification and auditing purposes (see below), trivial tracing has to be prevented. For this reason the nonces and ownership included in the DBC templates are encrypted to a secret that is itself encrypted to the mint and included in the signed transaction. Both user and mint can decrypt the secret in the future to reveal the exact nonce and ownership in the output DBC, if that need arrives.

Public transaction history log

eCache mints publish a record of all transactions in an unchangeable, append-only data structure. It is implemented as a hash-chain in which every entry includes the hash of the previous entry. This prevents re-ordering of the history. The current head of the chain can be requested from the mint any time, returning the head, current time, and a signature of both. The complete history can also be downloaded and verified.

The log includes these entries:

1. Public key of the signing key pair that the mint uses. These entries include the time range in which the public key is valid.
2. Changes to the backing capital of the mint. If value is added or removed from the capital this is recorded.
3. Changes to the issued capital of the mint. If the total value of issued DBCs changes then this is recorded. The issue capital must always be less then the backing capital and the difference defines the maximum size of the spend pool (see below).
4. Spend transactions: The user-provided transaction, including all input DBCs, output DBC templates, encrypted shared secret and owner signature. Furthermore the mint adds the result branch selected (see above).
5. DBC spend: The spent DBC, including signature. Directly follows the Spend transaction.
6. DBC issue: The newly issued DBC but without signature.
7. Pool entry. See below.

The history log allows for the recording of proof of mint operations. Transactions now become publicly verifiable if the user so wishes. Should the mint be suspected of dishonesty, the user can publish his transaction. The result of the transaction, and if so requested even the output DBCs, can be validated publicly. Furthermore the current capital of the mint becomes verifiable, making it possible to audit its backing versus issued DBCs.

Spend-Pool

Not all transactions with the Mint are recorded immediately into the history log. Spend operations are instead kept secret in pool of unrecorded transactions. The total amount represented by DBCs spent in those transactions is lower than the difference between backing capital and issued capital. Whenever a spend transaction reaches the mint it will first be recorded in the spend pool. Furthermore a hash is derived from a mint-kown secret and the hash of the transaction and recorded in the history log. When the amount of capital in the spend pool reaches a defined threshold, or the total number of transactions in the pool reaches a defined threshold, a random transaction from the pool is selected, removed from the pool and recorded in the history log. The combination of history log, signed transactions, concealed unique values and spend pool allows public verification on demand, while making public tracing of transactions hard. Care must be taken by the users to use common DBC amounts to prevent tracing through unique amounts.

Half-offline capability

The combination of ownership and publicly verifiable mint signatures allows for transactions in which the recipient can do a check against double-spends without having to communicate with the mint. The recipient only has to keep record of

the DBCs sent to it that have a specific owner. If a DBC is presented twice a double spend is detected, if the DBC is unique and carries a valid signature by the mint the recipient can be sure that the DBC will be accepted by the mint for reissue at a later point in time. Since DBCs can include two owners that are exclusive to each other and cover two defined time spans, it is also possible to pre-generate DBCs to be used in an offline manner at a later point in time. The user simply generates DBCs for the intended recipient that after a certain time fall back to be reissue-able by the user's key. This allows transactions between two parties in which both can be offline for a defined, possibly very long, time span without needing to interact with the mint. This makes the system very usable for scenarios of mobile point of sales transactions and simple hardware implementations, such as in the case of payment cards, public transport tickets, etc. Furthermore it makes the system much less vulnerable against denial of service attacks. To clarify: There are only two cases which require one of the transaction parties to talk to the mint during the transaction: The recipient is unknown, or the payment amount is unknown. In all other cases the transaction can be structured so that no communication with the mint is required during the transaction.

Recovery

The public history log allows the recovery of the mint using only public data. The only DBCs that are at risk for double spend at that point are those in the spend pool. If the pool entry secret is known even those DBCs can be prevented from being double spent.

eCache anonymity

Blind signatures

One feature that can be implemented in DBC systems is true anonymity and untraceablity of transactions. eCache implemented a second class of DBCs that would carry blind signatures instead of standard ECDSA signatures. The limitation was that it was only possible to reissue ONE standard DBC into ONE blind DBC, and vice versa. To make this possible in a single transaction, only specific amounts were supported. Instead of allowing any amount being represented by a DBC, the mint had one key per denomination and amount. A published list of keys and their respective values could then be used by both the mint and recipients to determine what amount and denomination a blind DBC had, simply by testing the signature. This way a direct conversion between DBCs with standard or blind signature was possible without risk of the mint being defrauded. However, when challenged the mint could not possibly prove from the history log that a spent DBC was actually the result of a previously recorded issue operation, leaving the possibility that the mint could overissue

DBCs undetected. The user could however still challenge the honesty of the mint by revealing the used blinding parameters.

JarMix

The issue of auditability of blind signature DBCs lead to the creation of the JarMix protocol. The goal here was the preservation of untraceability and anonymity of the user's payment while at the same time being able to audit the mint's issuing behavior. To this end a new entity is introduced that is independent of the mint: The Mix. The Mix is an entity independent of the Mint that only serves the purpose of increasing the anonymity and untraceability of user payments. The mental model here is a jar of coins into which the user can put one coin in and get a random other coin out. For this end, the mix operates the "jar of coins", a pool of valid DBCs with unique ownership keys per DBC. A user contacts the mix and requests the swap of DBCs. The mix returns a signed statement containing the public keys of ownership for both the DBC in the pool and the expected DBC from the user. The user then creates a matching transaction for the mint, producing a DBC to the mix as owner. That transaction is encrypted to the mint and given to the mix. The mix appends its ownership public key and forwards the packet to the mint. The mint then decrypts the transaction, verifies that the ownership of the output DBC matches the attached ownership key of the mix, and returns the new DBC to the mix. It is important to note here that all responses by the mint are signed by the mint. On receiving the reply from the mint, the mix returns the previously agreed on DBC and ownership private key to the user. At this point the mint does not know which DBC the user received, and the mix does not know which DBC the user spent. Nor does the history log allow the mix from learning anything about the input DBC because that transaction is still kept in the spend pool (with ownership of output DBC being a concealed value). The user needs to trust the Mix to not spend the DBC on his own until it has been reissued. To keep Mixes honest, auditors can do test transactions through the mix, increasing the risk for dishonest mixes to be discovered. For increased privacy, and decreased risk of fraud by mixes, the user would reissue mix-DBCs through other mixes, with a short delay in between. The result of a chain of mix operations would be that no mix nor the mint could easily connect the original input DBC with the newly generated output DBC. At the end of a chain of mix transactions the user then reissues the last mix-DBC at the mint directly. The anonymity set and level of untraceability at that point depend on the combined trust in the mixes, and the number of mix transactions done by other users - similar to the level of privacy provided by mixes for communication. Various extensions have been described to extend JarMix so that the mix cannot double spend the DBC it returns to the user. They are beyond the scope of this document.

Fees

eCache (and JarMixes) simply subtracted a unit from the input amount based on the number of DBCs issued as a result of the transaction, allowing for trivial prediction of fees.

Usage of modern DBC systems

While eCache itself is not in use anymore, lessons learned from designing and developing the system have influenced other developments. One of these examples is Mute, a secure messenger developed in 2016. DBCs are very useful for micro payments. The requirements for processing, storage and bandwidth are very low, while at the same time allowing the system to operate for extended periods of time without the users having to communicate with the mint. And DBCs are just a few hundred bytes in size. For these reasons Mute used blind (untraceable, anonymous) DBCs to allow users to pay for the service on a per-message basis - using DBCs as digital stamps. Users would purchase some amount of the internal stamp currency, turn it into blind DBCs owned by any one of the Mute servers, and then include those DBCs into messages. The Mute servers would verify the Mint's signature and check for double spend. On successful verification, the message would be forwarded.

Multi-currency Mints and advanced ownership

DBC systems are flexible and easily extended to support further use cases. While the ownership feature described above is powerful, it can be adapted for more complex protocols. An example for that is the experimental extension developed for eCache.

Instead of simply recording the public key for transaction signatures, the hash of a "contract" was recorded in the DBC. A contract is a boolean expression describing the conditions under which a DBC will be reissued.

Furthermore the contract can apply to more than a single DBC transaction. eCache supported "combined" transactions in which two or more DBC transactions could be presented to the mint at the same time, the conditions of each of the input DBCs referring to the output DBC templates of the other transaction. This allows creating schemes that allow swapping the ownership of two certificates presented by two different users.

For example, a DBC issued for the denomination "gold" for owner A and another for the denomination "EURO" for owner B can be simultaneously processed if and only if the respective output DBCs would be "gold" for owner B and EURO for owner A. This essentially implements a swap or sales.

To facilitate processes like these eCache in addition implemented an "Issue Book" which cached newly issued DBCs created in combined transactions. It allowed anybody knowing the output DBC template to retrieve the signature for it. That way a user could pre-sign a transaction, give it to another user for combining it with his transaction and sending it to the mint, and then ask the Mint for the signature to be able to recreate the result of the transaction.

Issuer risk

A central question for digital money systems is that of the issuer risk, that is: Risks introduced by the issuer. These can include manipulation of issued capital outside the bounds of backing capital, the operation outside of protocol, and unavailability due to the mint ceasing operation or being forced to close down. It is the issuer risk that is the main problem that blockchain cryptocurrencies have strongly mitigated. For DBC systems, one possible approach would be to separate backing from issuing, and make backing access dependent on continuous successful verification of mint operations. eCache could be used in this context since it allows users to demonstrate mint dishonesty, and everybody to audit the issued capital.

Future

Watching and participating in the digital money sphere over the last decades is only part of the story, the other is an interest and excitement on where things will go. For me there seems to be an evolutionary path that has the "issuer risk" as the attractor. Issuer risk is the risks faced by a user of a digital money system of not being in direct possession of an intrinsic value, but instead rely on some entity for keeping and transmitting value.

In the beginning, issuer risk was overcome by trusted issuers and a legal system. The issuer would be liable to the users of his system, and the legal system would both regulate his behavior and practice, and also promise redress in case the issuer would defraud the users. Added onto that was the idea of issuer insurance, basically guaranteeing the deposits of users up to a certain amount by other issuers taking over the liability in case of issuer failure.

The next step was the conception of verifiable and recoverable systems. Here a combination of protocols and auditing entities would be able to verify the good behavior of an issuer at any time. Should an issuer misbehave, his reputation and business would effectively be blackballed. The users of the system would in that case profit from the separation of issuing and backing, or bonding/underwriting, and the ability of those systems to be recoverable in the sense that a third party could verify the state of transactions and pay out the deposits accordingly. What the inception of verifiable and recoverable systems underestimated is the

regulatory risk of providing payment systems. Regulators demonstrated that they had nothing of this independent money crap. Issuers were shut down, backing operators robbed, people sent to jail for decades. While verifiable and recoverable systems were able to deal with technical failure and fraud, they were no match for governments simply taking them down. And those systems failed to ever get popular when their operators were anonymous. Before solutions for these issues were implemented (and solutions do possibly exist), the next evolutionary step took place:

Trustless blockchain cryptocurrencies. Here the system itself says good-bye to any notion of backing or intrinsic value. Instead, only virtual fiat money was issued, by the system itself, held exclusively in the system, regulated purely by protocol implemented in software. Instead of trusting a single entity, or a system of institutions, the trust in these "trustless" systems is distributed over those parties that cooperatively but independently operate the system. These parties being, realistically, the miners. It needs to be kept in mind that "the miners" is no pre-defined group, it just happens to be the set of entities currently engaged in mining cryptocurrency, and that set is highly fluid. One miner disappearing can be replaced by a dozen new miners starting the activity - without any coordination whatsoever. This construction is believed by many to solve the issuer risk forever. I tend to disagree. Yes, it is for sure a much more resilient setup, but for me it is less then certain if that makes the system immune to regulatory attack, blocking, subversion. Yes, you will always be able to prove that you still have cryptocurrency, but its exchange value might be close to zero, and it might not be transmittable at all. It should be noted that in verifiable DBC systems the fact of still having valid DBCs is true even if the mint is taken down. And these DBC systems are recoverable: Another entity is able to just continue operating from the known public state. The main essential difference being that in blockchain cryptocurrencies the backing still remains, while in pure DBC systems that abstract another currency this would not be the case. One of the areas of research hence could be thinking about distributed monetary control for DBC currencies.

That is where we are at. We went from trusted, to verifiable, to possibly trustless. The word I am missing in that enumeration is "trustworthy". So far, we have failed to build systems that are worthy of our trust. When we have trustless however, do we really need trustworthy anymore? My view is that building trustless systems is very limiting from perspectives on complexity, economics, privacy and adaptability. Trustless systems are necessarily huge, needing many persons to cooperate according to the same protocol. This makes them vulnerable to error of specification and a changing environment. They are also inherently expensive to operate and likely will never be as fast and cheap a DBC systems. Let alone that they are likely never going to achieve the same strength of privacy - anonymity and untraceability - then blind DBC systems can reach.

I do however have an intuition that we might find ourselves evolving past trustless systems towards trustworthy systems relatively soon. Let me throw a few ideas

and sketches at you.

Black Box Computing

A few decades ago, people started working on something called "Trusted Computing". The idea there is simple (the technology isn't): Create computer hardware that can prove to a remote user that it is exactly one specific kind of computer, with one specific kind of hardware and hardware settings, running exactly one specific program. This means that you could verify from afar that a computer is doing exactly what you would expect. This is called "attestation". Then came the cloud. Now hardware manufacturers are incentivized to push trusted computing even further, and make it widely available in their server processors. The next step taken was encrypted RAM. Here the processor itself would encrypt all access to RAM, preventing even somebody with physical access to learn the contents of memory. The key to encrypt the RAM would be generated by the processor itself, and never be made available to any other hardware or software. Examples for this are SGX, SME and SVE. Combine attestation and encrypted RAM, and you find yourself with a remote box that can process secrets. Secrets that nobody could learn or manipulate other than through the software running on the box - software that itself would be attested. I call this "black box computing". The moment these technologies are mature (they aren't right now), we will be able to set up black box computers anywhere, remotely, and be sure that our software runs as expected, and our secrets are secure on them - even against physical attack. This leaves us with the question if we should trust the software running on those machines. And the answer is as always: Trust but verify. Amazing advances have been made over recent years in the field of formally verified software. Software for which we can make mathematical proves that it performs one specific activity, and no other, even when presented with evil input. Please sit for a second and think about this: A formally verified program, running on a remote trusted computer, with no access to RAM but by the verified program. Should we reach that stage, and I'm very confident we will, our environment for running digital money systems will change dramatically. We will again be able to rely only on verifiable and recoverable systems, and they will be able to easily move from machine to machine, control money supply, and adapt as needed. And these systems will be cheaper, faster, more private and more powerful than anything done with blockchains today. There will be DBC systems again.

Until then, there is more to do. Recently there have been developments to create distributed DBC mints that are resilient to byzantine failure of a minority of participating nodes. Distributed DBC mints that are able to do blind signatures or employ JarMix for untraceability. It will be exciting to see what comes out of this. I have the feeling that the time of DBCs is still to come.

This article as PDF[2].

[2] /files/DigitalMoneyDBC.pdf