On Central Bank Digital Currency: A composable treatment

István Vajda
TU of Budapest
vajda@hit.bme.hu

Abstract: Central Bank Digital Currency (CBDC) is in the phase of discussion in most of countries. In this paper, we consider the security issues of centralized retail CBDC. Our focus is on the design and analysis of the underlying cryptographic protocol. The main security requirements against the protocol are transaction anonymity and protection against tax evasion. The protocol provides security guarantees in case of the strongest model of an execution environment which is the general concurrent environment. We apply the Universal Composition (UC) methodology of Canetti [3],[4]. At the time of this writing, we are not aware of any published CBDC protocol with an aim to provide secure compositional guarantees.

Index Terms: Central Bank Digital Currency, cryptographic protocols, provable security, universal composition

1. Introduction

Today, cryptocurrencies like Bitcoin are massively used around the world. However, their value is highly volatile as they are purely speculative assets. This fact is slowly threatening the stability of financial systems. The CBDC may be the response of states to this challenge. However, CBDC is only in the phase of discussions in most countries [1].

Cryptocurrencies use distributed ledger technology (DLT), where the purpose is to establish a consensus across lots of parties without relying on a central party. In the case of a retail CBDC issued by a central bank distribution of the central bank's ledger could only increase transaction costs without benefits.

The number of central banks investigating CBDC grows steadily (e.g. existing Digital Yuan, planned Digital Euro, and Digital Dollar). There is currently no consensus on how a CBDC should be designed and what (security) features it should have. These questions are being intensively discussed. We want to contribute to this phase of discussions with our paper.

Our targeted application is centralized retail CBDC. We will call the ensemble of the central bank and the system of commercial banks an Exchange (E), without detailing its structure. We focus on the security of operations concerning the digital coin. The usual financial operations on the bank accounts are not part of the studied functionality, these are thought to be performed in the environment of the protocol. The users are called Customer(s) (C) and Merchant(s) (M). Parties (E, C, and M) run the (cryptographic) protocol operating on digital coins.

The scenario about the issue and usage of (retail) CBDC coins is as follows. A cus-tomer withdraws coins from its cash account intending to spend them in an online transacti-on at a merchant. The merchant deposits the received coin at the central bank, where it also sends information about the content of the transaction. Such a process would allow the state to tax transactions. However a malicious customer in an attempt to avoid taxation might give the coin to the merchant outside of the system, and then the transaction would be invisible to the authorities.

A CBDC implies a wide range of security issues. We focus on the security of the underlying cryptographic protocol that generates digital coins and performs operations on those. In our design, we will consider the following security requirements for a CBDC. The basic requirement is that an adversary should not be able to forge valid coins. The key feature is transaction privacy. The other CBDC-specific goal is thwarting tax evasion. Note, this goal is a challenge as it conflicts with transactional anonymity. Though money laundering attacks used to be considered a security issue in the case of a wholesale CBDC system, we will briefly discuss it to give a more complete picture.

A standard banking requirement is that a customer's money assets should not be damaged due to failures in a successful execution of the protocol (e.g. due to malicious premature aborts). We will also address this requirement in the discussion section.

Transaction privacy guarantees that private data cannot be collected about custom-ers. Specifically, the central bank only sees the total amount of money withdrawn and spent, where total means across the entire CBDC system. The commercial bank learns how much money a given client has exchanged for coins but does not see how much and where the client has spent it. We note in connection with the privacy issue that general data protection regulations have been introduced in Europe a couple of years ago (GDPR [10]).

The other CBDC-specific goal is to reduce the chance of successful tax evasion and money laundering. One component of protection is that a digital coin expires after a period of time. The second comes from the central bank's privileged abilities to monitor spending flows nationwide. The third component is a deterrent effect that the dishonest behavior of those involved in the attack puts themselves at risk.

Digital Euro (a project of the European Central Bank)) plans to safeguard users' privacy for lower-value transactions, while still ensuring that higher-value transactions are subject to mandatory AML/CFT (Anti-Money Laundering/Countering the Financing of Terrorism) protection. A similar example is the already existing Digital Yuan that aims to support anonymous small value payments but traceable larger amounts. Small value payments require only a phone number as an identity. In the case of higher value payments, users have to provide conventional proof of identity to open a wallet. The digital yuan project considers the measures against AML, CFT, and tax evasion as a future task.

The structure of the paper is as follows. In Section 2 we present related works and name our contributions. Section 3 provides an overview of the system and design considerations. In Section 4 we define our assumptions. Section 5 presents the formal definition of our CBDC ideal functionality as well as security implications of it. The short Section 6 refers to the real protocol as well as the blind signature and authorization ideal functionalities, all of which are detailed in the Appendix. Section 7 presents an example simulation. Section 8 discusses additional implementation and extension issues.

2. Motivation Behind this Work

The CBDC may be the response of states to the challenge posed by highy volatile crypocurrencies potentially threatening the stability of financial systems. However, there is currently no consensus on how a CBDC should be designed and what features it should have. We wanted to contribute to this phase of discussions with our paper.

The title of the paper refers to compositional treatment. As a nationwide system a large number of users may run (the coin operation sub-protocols of the) CBDC protocol simultaneously. Additionally, as the communication media will probably be a publicly accessible network (e.g. Internet), instances of other, potentially hostile protocols (i.e. protocols designed to attack the CBDC protocol) may also use the same communication space simultaneously. Our goal is to meet the security requirements in such an execution environment, called a general concurrent environment. The other goal was to be able to design and analyze in a modular way, called modular composition. This latter goal implies the simulation-base definition of security. For all these purposes, we choose the Universal Composability (UC) approach of Canetti [3], [4].

At the time of writing, we are not aware of any publications presenting a composable approach for the design and analysis of a provably secure CBDC protocol.

3. Contribution of the paper

The main contributions of the paper are as follows:

- we draw attention to the fact that the assumption of the general concurrent network model is the appropriate one for the designing of a CBDC protocol
- we proposed an ideal functionality for the CBDC task and we pointed out that all relevant security requirements are implied, such as transaction anonimity, unforgeability of digital coins and prevention of the intention to evade taxation (Theorems 1-3)
- we defined the necessary assumptions for the UC secure implementation of that ideal functionality
- we defined an UC-secure protocol for the CBDC task and provided a proof (Theorem 4)

4. Related work

A published and well-documented provably secure design we know about is protocol Taler [6]. The authors of protocol Taler gave the classic approach of D. Chaum with blind signatures [5] a new life, greatly expanding it, where the expansion means new types of coin operations and extended security guarantees. Recall, a blind signature allows a user to obtain a signature on a hidden message without the signer learning the message in question.

The authors of protocol Taler coined the notion of income transparency as a measure against tax evasion. Their corresponding measure is the Link protocol. The Link protocol threatens the users to lose exclusive control of coins that were received without being visible as income to the exchange (despite having the option to refresh them).

This protocol inspired our paper the most. The main similarities between our work and the Taler are as follows. Our approach is also based on the idea of blind signatures. In the task of thwarting tax evasion, we also use the idea of a threat of losing coins to enforce income transparency. Furthermore, operations on a coin (deposit and refresh) can only be performed with the consent of the party that withdrew the coin.

Protocol Taler uses game-based security notions. The design relies heavily on hash functions. Consequently, the authors could present security proofs only in the random oracle model (ROM), where in the analysis a hash function is replaced with a truly random function. Furthermore, they analyzed their protocol in a stand-alone network setting.

In contrast, we design a CBDC protocol that is provably secure in the general concurrent execution environment, that is – we believe - the realistic model for running a protocol over publicly accessible communication networks (like the Internet). Furthermore, there are several benefits if the design is modular. Accordingly, we use simulation-based security notions that naturally support secure composition. In particular, we use the techniques and theorems of the Universal Composition (UC) approach of Canetti [3],[4].

As a necessary „ingredient" of this approach, we define an ideal CBDC functionality. In this paper, we did not strive to provide the widest potential range of services and measures. Rather, we wanted to demonstrate the advantages of a design and analysis methodology in connection with a convincingly strong „list" of security guarantees. Nevertheless, the UC methodology can be applied to essentially any set of security requirements and execution environments.

Blind signature is at the core of the Chaumean approach we also follow. As we aim for a UC-secure CBDC protocol, we need a UC-secure blind signature algorithm. Blind signature schemes which require only one round of interaction (i.e. two moves) are called round-optimal. Lindell [13] showed that the construction of UC-secure blind signatures is impossible in the standard model (specifically, the assumption of the CRS (Common Reference String) setup [4] cannot be eliminated).

Fischlin [7] showed a construction for round-optimal blind UC-secure signatures. This construction guarantees blindness property even againts a malicious signer (bank) that chooses its keys maliciously. The essence of his tricky idea of this protocol is as follows. The client generates a commitment to the message. The signer signs this commitment using a standard signature scheme. The blind signature is a non-interactive zero-knowledge (NIZK) proof (given by the user) that the signed commitment opens to the message. The cryptographic assumptions are IND-CPA secure encryption scheme, non-interactive commitment, and non-interactive zero knowledge protocol (NIZK).

Fischlin's UC secure construction assumes non-adaptive corruptions. Kiayias [12] showed a construction for a practical UC-secure blind signature that is secure against adaptive adversaries as well.

In this paper, we considered a cryptographic technology to guarantee security requirements. However, we must see that the "everyday attacks" are typically not directed against underlying cryptographic protocols, and especially not against cryptographic primitives in those. (This is especially true if security is proven because in that case it is guaranteed that a successful attack against the protocol is impossible.) Instead, adversaries usually look for vulnerabilities in the (software) implementation. Recall, successful cyber-attacks against cryptocurrency implementations/systems.

Accordingly, the CBDC services, promised safe by the state, need to provide a high level of protection from cyberattacks. In the event of successful attacks, the recovery time should be short and the integrity of the data protected. Our paper is not about cybersecurity of CBDC implementations. For example, here we mention that authorization of payments prevents the classic phishing (Garrera [8]) and credit card fraud (Sahin [15]) attacks.

5. Assumptions and definition of security

A provable security approach includes clear definitions of the one's goals, clear definitions of one's assumptions as well as a rigorous justification (proof) of the claim that if the stated assumptions hold then the designed system meets the stated goals.

All parties are assumed to run efficient protocols only, i.e. parties are from complexity class PPT (Probabilistic Polynomial Time).

Adversarial model: We assume that dishonest customers and merchants (i.e. users) can be malicious. Examples of potentially malicious behavior are as follows: attempt to forge a coin; a customer and a merchant might collude for tax evasion; a malicious customer might try to deposit a coin ahead of the merchant that he has already given to the merchant as a payment in a successful transaction; a malicious merchant might obtain the price of goods without delivering them and being accountable under a contract; a malicious merchant might try to break the anonymity of a customer.

A dishonest Exchange can be semi-honest, that although it generates its messages according to the specification of the protocol but it wants to learn private information about the customers from the transcript of the executed protocol. In particular, it wants to break the transaction anonimity, i.e. to find a correlation between spending operations and the withdrawal or refreshing operations.

We assume static corruption, i.e. an adversary makes its decision about which participant to corrupt at the onset of the execution of the protocol and sticks to this decision throughout.

Network model: Communication channels between the parties are assumed to be secret, i.e. authenticated and private. Specifically, an authenticated channel here means the following. Customers can authenticate a merchant, however, the merchant will not know the real identity of its client in a transaction (gets to know just a „pseudonym"). Similarly, during the refreshing operation, the bank does not identify the customer (the bank only checks the validity of the coin).

However, when a coin is withdrawn from the Exchange or a coin is deposited at the Exchange, (full) authentication takes place in both directions, i.e. the Exchange identifies the client.

Formally, we will design and analyze a protocol that is a hybrid in a secret channel ideal functionality. (This can also be seen as a step of the modular design and analysis we follow in this paper.) Specifically, we assume a standard secret channel functionality ($F_{SC,1}$) and also a weaker one where no information is revealed about the real identity of the customer ($F_{SC,2}$). Accordingly, when we talk about a customer with identifier C, specifically we will consider a real identifier C1 or a pseudo identifier C2.

Consequently, a MIM network adversary can learn only the traffic information (network addresses of parties, timing, and bitsize of exchanged messages).

We assume a general concurrent setting. Such a setting models the real-world execution environment in a far more realistic way than the classic stand-alone definitions.

Trust assumptions: We will assume the availability of non-corruptible (i.e. trusted) functionalities in the system such as CRS setup, authorization functionalities running within the trusted core part of wallets, and standard PKI-like functionality for public keys (of the Exchange and the underlying communication channels).

Intractability assumptions: We assume the existence of a standard EU-CMA secure signature scheme and a UC-secure blind-signature scheme.

Security guarantees: Recall, the two approaches to the definition of security of protocols are the game-based and the simulation-based ones. We follow the latter one, in particular, the Universal Composition (UC) approach of Canetti. To save space, for the definition of UC-security, furthermore for the related fundamental notions and results we use, such as ideal functionality, hybrid protocols, universal composition (UC), joint-state universal composition (JUC), straight-line simulation as well as the UC and JUC theorems we refer to the original source [4]

6. Ideal functionality for the CBDC tasks

6.1 Components and operations of the system: an overview

The model of the (real) system under study consists of two main parts: the cryptographic subsystem and its environment. The cryptographic subsystem consists of the devices running the cryptographic protocol on behalf of the Customers, Merchantsk and the Exchange. The components of the environment are the corresponding parties as decision-making humans (customers, merchants, bank employees) and the usual banking infrastructure (managing the accounts of the bank's customers) as well as the communication network. For the sake of simple reference, we call the part of a participant running in the environment an external part.

The input/output (I/O) interface between the protocol and the environment is a key concept in the simulation-based definition of security. The I/O variables appearing at the interface of the Exchange are usual financial transaction variables (like customer's public identifier, type of operation, denomination of coin, time of expiration, accept/error messages). Users at their interfaces can additionally see their coins in some representation (stored in their wallets).

We consider the following type of actions with coins: withdrawal, spending, deposit, and refreshing. When a customer interacts with the Exchange it has to authenticate itself using its secret authentication key. The Exchange mints a new coin for the request of a customer having an account at the bank. The Exchange uses unique (public, secret) pairs of keys for dif-ferent denominations of coins. The essence of minting is a blind signature given by the bank on the blinded coin identifier using the secret denomination key. The blinding of the coin identifier guarantees non-traceability of the party when it spends the coin in a transaction. We will call a coin valid if the verification of the blind signature is successful. We say that a forging attack is successful if an adversary can produce a valid coin.

A coin is defined as coin= (d, ToE, coinId, coin_sign, status), where $d \in D$ is the value of denomination, D denotes the set of different denomination values, ToE stands for Time of Expiration, coinID is the unique identifier of the coin and coin_sign is the blind signature given by the Exchange. The value of the status variable can be fresh, spent, deposited, re-freshed and expired. A unique (public, secret) pair of keys is assigned by the withdrawer to each coin (coin owner's public and secret keys or simply coin-keys). The public key is rela-ted to the identifier (coinId) of the coin. The coin's secret key remains the secret of the ori-ginal owner of the coin throughout all operations. Disclosing it to the Exchange would void the guarantee of transaction anonymity. The owner of a coin can use the coin's secret key to authorize another party for a coin-related operation. A coin can only be deposited (by the merchant) if the customer authorized this action, technically by signing the contract (an appropriate mapping of the contract) related to a (successful) transaction. This authorization mustn't reveal the identity of the customer.

Coins expire periodically. If a coin is not deposited at the bank within a time interval it becomes lost to its keeper. It is a measure against tax evasion, in particular against the circulation of coins outside the system. Nevertheless, there are advantages of the refreshing of coins. When the owner of a coin wants to lengthen the lifetime of a coin, it can refresh it at the Exchange. Refreshing of a coin is the action when an old coin is replaced with a fresh coin using the same blind signature technique. On the one hand, the ability to refresh a coin reduces the cost of managing the bank account of customers. On the other, it weakens the expiration technique as a security measure. A compromise solution is to limit the number of updates and introduce such limits as a system parameter.

Each user has a digital wallet where the coins are stored and protocol components are executed. We assume that a wallet contains a trusted core element for the authorization functionality. The authorization functionality may only be activated by a party who is in the physical possession of the wallet. The trust assumption implies that the secret authorization key is ideally protected during this operation.

6.2 Definition of ideal functionality F_{CBDC}

Session identifier (sid) uniquely identifies an instance of F_{CBDC}. It is an abstract reference to the unique set of keys generated by the Exchange's key setup algorithms at the onset of the protocol run. Ideal functionality F_{CBDC} can be viewed as a set of subfunctionalities corresponding to different operations on coins (withdrawal, spending, deposit, and refreshing). Subfunctionalities of F_{CBDC} (sid) may run concurrently (recall, functionality F_{CBDC} models a nationwide financial service with lots of users active at the same time). Subsession identifier ssid=(P1, P2, ssid') uniquely refers to an instance of such a sub functionality, where P1 and P2 are the identifiers of the participants. Subsession identifier is an abstract reference to the instance of the keys corresponding to the underlying communication channel. For better comprehensibility when a customer C communicates over a fully authenticated channel we name it C1, otherwise, we use the notation C2.

When we write that ideal functionality „sends output" message to a party it is understood that it „sends private delayed output" in the original formalism of Canetti. We use the shorter wording so the lengthy definition is easier to read and all outputs are of the type of secret delayed (as we assume secret channels).

The identifier of the coin (coinId), in general, is an efficiently invertible deterministic mapping Id of the coin's public key, i.e. coinId=Id(AuthorVerif).

We will equivalently refer to a public key and the corresponding (binary representation) of the verification algorithm.

A state variable is assigned to each coin. This variable can take values fresh, spent, deposited, (refreshed, rfcnt) and expired. where a coin is in state (refreshed, rfcnt) after number rfcnt completed refresh operations. The maximum number of refreshes a coin can get is RFCNT.

For a clearer description of the functionality, checking the expiry of coins is implicit, i.e. not shown in the definition (of the ideal functionality). The functionality (in real life the real Exchange) periodically performs this check on the coin records it stores and chan-ges the status of expired coins to expired. We comment that making the expiration process explicit in the definition could be done by introducing a special trusted party called Timer. The algorithm of Timer is very simple: it sends an input message (Time, sid, time) to F_{CBDC} periodically.

The definitions of the ideal functionalities of the subprotocols follow in order. We use the related formal description language. We also append explanatory comments to the definitions. Definition of functionality starts with its name in bold and ends with a symbol □.

The definitions of the ideal functionalities of subprotocols consist of 1-3 rules. Each rule receives input only once. Additional inputs are ignored. Identifier (sid, ssid) identifies an instance of a subprotocol. The instance is invoked with the arrival of the input to the first rule. The consecutive inputs to the instance are expected to arrive at the rules in order. The last instruction of the last rule halts the instance. Any non-resolvable event leads to abortion of session (sid, ssid) (e.g. deposit request for a non-existent coin).

Ideal functionality Setup is shown in Fig.1. This subprotocol is called first when a new instance of F_{CBDC} is invoked. A pair of (public, secret) algorithms/keys are generated for each denomination of coins. Session identifier sid uniquely refers to the actual set of these pairs. The public components are output to the Exchange. (These components are made public by the external part of the Exchange.)

1. Upon receiving a value (***BlindSignKeysReq***, sid) from a party E, ideal functionality verifies that sid = (E, sid') for some fresh value sid'. If not so, then ideal functionality ignores the input, else it hands the message (***BlindSignKeysReq***, sid) to the adversary.

Upon receiving (***BlindSignKeys***, sid, $\{V_{BS,d}\}_{d\in D}$, $\{BlindSign_d\}_{d\in D}$) from the adversary, ideal functionality outputs (***BlindSignVerifKey***, sid, $\{V_{BS,d}\}_{d\in D}$) to E and it records (sid, $\{V_{BS,d}\}_{d\in D}$, $\{BlindSign_d\}_{d\in D}$) .

2. The list of valid and invalid coins (VC and IVC, resp.) is set to an empty list. □

Figure 1: Ideal functionality Setup

Ideal functionality Withdrawal is shown in Figure 2. Subprotocol Withdraw with subsession identifier ssid is used to withdraw a single coin of some denomination d. Customer C1 sends a request to E to withdraw a coin. The external part of E decides whether to allow the withdrawal. If the decision is positive, the functionality generates the authorization key pair. A blind signature is generated according to the rules of the blind signature ideal functionality F_{BS}. Instance F_{CBDC}(sid) halts according to the unforgeability condition of F_{BS}. Otherwise, the new coin is stored by the functionality, and customer C1 receives the fresh coin.

1. Upon receiving an input (***WithdrawRequest***, sid, ssid, d) from party C1, ideal functionality verifies (sid, ssid), ssid=(C1, E, ssid'). If not so, then ideal functionality ignores the input, else it sends an output (***WithdrawRequested***, sid, ssid, d) to party E.
2. Upon receiving an input (***WithdrawDecision***, sid, ssid, b) from party E, ideal functionality sends output (***WithdrawDecision***, sid, ssid, b) to party C1.
If b=0, then ideal functionality aborts subsession (sid, ssid), else proceeds as follows:
- generates a pair of algorithms (AuthorVerif, AuthorSign),
- sends output (***AuthorVerif***, sid, AuthorVerif) to party C1
- *if* C1 is honest *then* ideal functionality runs the blind signature protocol on input (m,$V_{BS,d}$), m=(d, coinId), coinId=Id(AuthorVerif) and obtains signature *coin_sign* ← Blind_sign(m,$V_{BS,d}$),
- *else*, it does:
- sends (***BlindSignRequest***, sid, ssid, m,$V_{BS,d}$) to the adversary.

Upon receiving an input (***BlindSign***, sid, ssid, m, coin_sign) from the adversary, ideal functionality F_{CBDC} ***halts*** if a coin (d, . ,coinId, coin_sign, .) has already been added to the list of invalid coins IVC. (unforgeability condition)
In both cases, ideal functionality proceeds as follows:
- stores coin= (sid, d, ToE, coinId, coin_sign, fresh),
- adds coinId to the list of valid coins (VC),
- stores coin-keys record (sid, coinId, AuthorVerif, AuthorSign),
- sends output (***Withdrawn***, sid, ssid, coin) to party C1,
- sends output (***Withdrawn,*** sid, ssid) to party E,
- halts subsession (sid, ssid). □

Figure 2: Ideal functionality Withdrawal

Ideal functionality Spending is shown in Figure 3. When a customer C2 pays a merchant M during a successful transaction, the functionality checks if the coin is spendable. Then, the functionality sends the coin together with a signature (depositPermit) to the merchant, where the signature authorizes the merchant to deposit the coin. Note, this sub-functionality does not check the validity of the coin. In this model, verification of the Exchange's signature is the task of the Exchange during a deposit operation.

1. Upon receiving an input (***Spend***, sid, ssid, transactionId, coin) from party C2, ideal functionality verifies (sid, ssid), ssid=(C2, M, ssid'). If not so, then it ignores the input, else proceeds as follows:
- verifies that the coin is spendable (status is fresh/refreshed)
- updates coin's status to *spent*,
- retrieves algorithm AuthorSign from coin-keys record (sid, coinId, AuthorVerif, AuthorSign),
- computes signature depositPermit ←AuthorSign (sid, ssid, transactionId, coin),
- stores transaction-record (sid, tansactionId, coin, depositPermit)
- sends output (***Spent***, sid, ssid, tansactionId, coin, depositPermit) to M,
- sends output (***Spent***, sid, ssid, coin) to party C2. □

Figure 3: Ideal functionality Spending

Ideal functionality Deposit is shown in Figure 4. When a deposit request arrives at the functionality, first it checks if the status of the coin is not deposited/expired. Next, it verifies the validity of the coin. This is followed by the verification of the authorization signature. If there is no recorded authorization signature depositPermit (within a transaction-record generated in a previous Spend operation) and the received signature successfully verifies then instance F_{CBCD}(sid) halts running (unforgeability condition). If all verifications are successful the external part of E decides about the acceptance of the deposit request.

1. Upon receiving an input (***DepositRequest***, sid, ssid, tansactionId, coin, deposit Permit) from a party M, ideal functionality verifies (sid, ssid), ssid=(M, E, ssid'). If not so, then ideal functionality ignores the input, else it proceeds as follows:
if the status of coin is deposited/expired then it aborts subsession (sid, ssid), else

if coinId is on the list of non-valid coins (IVC) then it aborts subsession (sid, ssid), else
if coinId is not on either the list of valid coins (VC) or the list of non-valid coins (IVC) then it adds it to the list IVC and aborts subsession (sid, ssid), else

if {there exists no transaction-record (sid, tansactionId, coin, depositPermit)}AND {AuthorVerif(sid, ssid, (transactionId, coin), depositPermit)=1}, then ideal functionality ***halts***, else
if {there exists no transaction-record (sid, tansactionId, coin, depositPermit)}AND {AuthorVerif(sid, ssid, (transactionId, coin), depositPermit)=0}, then ideal functionality aborts subsession (sid, ssid), else it
sends output (***DepositRequested***, sid, ssid, d) to party E.

2. Upon receiving an input (***DepositAccept***, sid, ssid, d) from party E, ideal functionality procceds as follows:

- updates coin's status to *deposited,*
- sends output (***Deposited***, sid, ssid) to party M,
- halts subsession (sid, ssid). □

Figure 4: Ideal functionality Deposit

Ideal functionality Refreshing is shown in Figure 5. Upon the arrival of a request for authorization of refreshing of a coin with status fresh/refreshed from a party C2 the functionality generates an authorization signature (refreshPermit). Upon the arrival of a refreshing request for a coin, the ideal functionality verifies the status of the coin, the validity of the coin, and the authorization signature, in order. Next, the functionality signals a refreshing request to (the external part of) party E. Upon receiving the permission ideal functionality generates the refreshed coin (by performing essentially the same operations as in case of withdrawal).

1. Upon receiving an input (***RefreshAuthorSignRequest***, sid, ssid, coin) from party C2, ideal functionality verifies (sid, ssid), ssid=(C2, E, ssid'). If not so, then it ignores the input, else it proceeds as follows:

- verifies that the status of coin is *fresh*/*refreshed*
- retrieves coin-key record (sid, coinId, AuthorVerif, AuthorSign),
- computes signature refreshPermit ←AuthorSign (coin),
- stores refresh-permit record (sid, coin, refreshPermit),
- sends output (***RefreshAuthorSign***, sid, ssid, coin, refreshPermit) to party C2.

2. Upon receiving an input (***RefreshRequest***, sid, ssid, coin, refreshPermit) from party C2, ideal functionality proceeds as follows:
if the status of coin is spent/deposited/expired OR rfcnt=RFCNT then it aborts subsession (sid, ssid), else

if coinId is on the list of non-valid coins (IVC) then it aborts subsession (sid, ssid), else
if coinId is not on either the list of valid coins (VC) or the list of non-valid coins (IVC) then it adds it to the list IVC and aborts subsession (sid, ssid), else

if {there exists no refresh-permit record (sid, coin, refreshPermit)} AND {AuthorVerif(sid, ssid, coin, refreshPermit)=1}, then ideal functionality ***halts***, else
if {there exists no refresh-permit record (sid, coin, refreshPermit)} AND {AuthorVerif(sid, ssid, coin, refreshPermit)=0}, then ideal functionality aborts subsession (sid, ssid), else
it sends output (***RefreshRequested***, sid, ssid, d, rfnct) to party E

3. Upon receiving an input (***RefreshPermit***, sid, ssid, b) from party E, it verifies message b. If b=0, then it aborts subsession (sid, ssid), else it proceeds as follows:

If C2 is honest then ideal functionality

- generates a pair of algorithms (AuthorVerif, AuthorSign),
- sends output (***AuthorVerif***, sid, AuthorVerif) to party C2
- runs blind signature protocol on input (m,$V_{BS,d}$), m=(d, coinId), coinId=Id(AuthorVerif) and obtains signature coin_sign ← Blind_sign(m,$V_{BS,d}$).

else it

- sends (***BlindSignRequest***, sid, ssid, d, coinId) to the adversary,
- upon receiving an input (***BlindSign***, sid, ssid, d, coinId, coin_sign) from the adversary, ideal functionality halts if (*d*, .,coinId, coin_sign, .) has already been added to the list of invalid coins IVC. (unforgeability condition)

In both cases, ideal functionality

- increments refresh counter (rfcnt) by 1
- replaces coin with coin= (sid, d, ToE, coinId, coin_sign, refreshed, rfcnt),
- adds coinId to the list of valid coins (VC),
- stores coin-key record (sid, coinId, AuthorVerif, AuthorSign),
- sends output (***Refreshed***, sid, ssid, coin) to party C2,
- sends output (***Refreshed,*** sid, ssid) to party E,
- halts subsession (sid, ssid). □

Figure 5: Ideal functionality Refreshing

6.3 Security properties implied by F_{CBDC}

In this chapter, we argue that the definition of ideal functionality F_{CBDC} implies the following (game-based) security properties: transaction anonymity, unforgeability of coins, andvthwarting tax evasion, We will also make comments on the protection against money laundering. Formally, we argue that provided one of the above security properties is not met (i.e. adversary can win in the corresponding game), the ideal and the real executions can be distinguished and the latter implies that no (efficient) implementation of the ideal functionality F_{CBDC} can be simulation-secure.We use the common notation Z for the environment of the protocol.

Transaction anonymity:
We use the game-based definition of transaction anonymity shown in Figure 6. In this game, the adversary is allowed to see all the messages exchanged between parties. This is not in contradiction to our assumption about secret communication channels. This latter assumption prevents a MIM adversary to see the (content of the) messages, however here we are considering the protection of the customer's privacy against dishonest parties.

Theorem 1: *Ideal functionality F_{CBDC} implies transaction anonymity.*
Proof: Assume there exists an efficient adversary (A) such that it can win the anonymity game with probability ½+∊, where ∊ is a non-negligible positive value. We define an efficient algorithm Z that can distinguish the real and the ideal system with an advantage ∊. Z simulates the oracle in the following game:

1. Z receives the public identifier of a pair of customers from the adversary.
2. Z sends withdrawal requests to its underlying system (real or ideal) in the name of the customers.
3. The system returns a pair of coins ($coin_0$, $coin_1$).
4. Z flips a coin with result b. Z sends pair of coins ($coin_b$, $coin_{1-b}$) to the adversary.
5. The adversary is allowed to ask the oracle (Z) to use the coins in different operations (spending, deposit, refreshing) and to see the corresponding transcripts.
6. Finally, the adversary outputs a guess *b'* on bit *b*. If $b^* = (b' \text{xor } b) = 0$, then Z outputs message "real", otherwise it outputs message "ideal".

Figure 6: Anonimity game

Z will have an advantage ∊ in the distinguishing game. It follows from the definition of ideal functionality F_{CBDC} that $P(b^*=0 \mid \text{ideal})=1/2$. It is so because the messages exchanged between parties do not reveal any information about the party who has withdrawn the coin. Indeed, the components/variables appearing in transcripts (such as pseudo-name C2 of the customer, coin (d, ToE, coinID, coin_sign), transactionId, depositPermit, refreshPermit*)* do not reveal any information about the real identity (C1) of the customer. On the other hand, from the success assumption on the adversary in the anonymity game, we get $P(b^*=0 \mid \text{real})=1/2+∊$. Therefore, the advantage of Z in distinguishing game is $\mid P(b^*=0 \mid \text{real}) - P(b^*=0 \mid \text{ideal})=1/2 \mid = ∊$. □

Unforgeability of coins:
Z simulates the oracle in the unforgeability game shown in Figure 7.

1. Upon a request from the adversary, Z "mints coin" for customer C with denotation d, chosen by the adversary. (Z executes the withdrawal operation playing the role of customer C). Z sends the coin to the adversary.
2. Eventually adversary outputs a fabricated coin, and Z validates it.
3. Z outputs "success" if the fabric is correct otherwise outputs "failure". □

Figure 7: Unforgeability game

Theorem 2: *Ideal functionality F_{CBDC} implies unforgeability of coins.*
Proof: A non-negligible advantage ∊ in breaking the property (against the real protocol) results in the same advantage in the distinguishing task. Indeed, the probability of successful forging of a coin is essentially zero in the ideal system. This is because the ideal blind signature functionality verifies as valid only those coins that are generated by the ideal functionality. □

Protection against tax evasion:
Ideal functionality F_{CBDC} provides conditional protection against tax evasion, depending on the behavior of the participants:

Theorem 3: *Ideal functionality F_{CBDC} implies conditional protection against tax evasion. There is no protection, if the participants in the attack cooperate and are completely trustworthy to each other. If the participants in the attack cannot trust each other, the attack is prevented by the risk that the protocol will allow the coins obtained during the illegal transaction to be stolen from each other.*

Proof: We consider a group of number m colluding dishonest parties ($P_1,\ldots,P_m$). In one of them, the parties can fully trust each other (group Ft), in the other they may be dishonest to each other (group Dh). First, suppose that there is no option for refreshing a coin. Furthermore, at the beginning of the game just one of the members (say P_1) has a fresh coin of value d in its wallet, all the other members have an empty wallet.

Clearly, to avoid financial loss, on the day of expiration of the coin, participant P_1 must hold the coin so it can deposit the coin on time. The total number of transactions (using the coin) depends obviously on their time duration and the length of the expiration period. For instance, party P1 buys 1 coin worth of goods from P2, then P2 from P3,… and finally Pi from P1. All of these transactions take place outside the system, without paying taxes. It is clear that for group Ft tax evasion scenario works successfully and it cannot be prevented.

However, this is not the case for group Dh. The point is that any member of the group involved in past transactions may decide to deposit the coin, i.e. to steal the value of the coin from its recent owner. Indeed, upon a transaction, full control over the coin will not be transferred and it is only shared. Note, as the members are dishonest the group cannot uniquely identify the one who broke "the rule".

Now, we consider the effect of the refresh option as well. This option extends the time interval available for dishonest parties for tax-avoiding transactions. If in the above example of the chain of transactions party P_i received the coin as payment and decides to refresh it, then party P_i acquires sole control of the (refreshed) coin. However, when this coin takes part in further transactions the coin becomes shared again.

In conclusion, the unavoidable risk of potential money theft limits the willingness to engage in transactions that avoid accountability. The refresh option does not eliminate this risk for dishonest parties. □

6.4 On protection against money laundering

Imagine that a huge amount of digital coins are stolen in a hacking attack. The thief wants to launder the stolen coins into bank account money. (We mention that this is a live example of a massive Bitcoin attack.)

We notice that in contrast to the tax evasion scenario it is not a realistic assumption that the owner of the coins will thwart money laundering attempts by depositing the coins in time before the thief. This is so because, if the owner notices such a large-scale hacker attack, he is likely to notify the authorities and not act independently.

We can distinguish two cases. In one of the cases the hacker cannot break into the authorization submodule, so the hacker cannot deposit the coins. Recall, we assumed such a trusted component within the wallet.

In the other case, the hacker gets full control of coins and no cryptographic protection remains. The protection against money laundering becomes the task of the external part of the Exchange. A realistic protection measure could trace and detect extremal patterns in the time stream of deposited values and it might limit the number of coins that can be redeemed per unit of time and per party.

The ideal protection rule could be the following: the Exchange only allows to deposit coins by a user the amount of which does not exceed the amount of money withdrawn minus the amount spent in legal transactions (by the user). Note, on the one hand, such a rule implicitly assumes that the production of goods has also been financed within the CBDC system, i.e. the system is closed in this respect. On the other hand, such a rule would exclude legal profits (on the side of the manufacturers of goods and the merchants). For such reasons such a strict approach does not seem to be „economically viable". It is not possible to define what a fair profit is, but it could be formally given as a parameter (of the functionality) and that could represent some percentage of the money withdrawn. The other approach (mentioned above) is to define the extremality in the time stream of deposited values extremal pattern. However, such details are already out of the focus of our present paper.

7. The protocol and its analysis

For the sake of clarity of the paper, we shifted the (also lengthy) definition of the real protocol to the appendix (Appendix 1). The UC compositional methodology helps to simplify the analysis by requiring the designer to construct a (straight-line) simulator only for a hybrid protocol instead of the real protocol. The F_{ATH}-, F_{BS}-hybrid protocol can be obtained with relatively minor modifications of the real protocol. The steps of the computation of a blind signature and its verification are replaced with sending the corresponding inputs to ideal functionality F_{BS}. Similarly, the steps of the computation of an authorization signature and its verification are replaced with sending inputs to ideal functionality F_{ATH}.

Our main claim is the following:
Theorem 4: *Under our assumptions (see in Section 4), our CBDC protocol (see in Appendix 1) UC-securely realizes ideal functionality F_{CBDC}.*

Proof: The main steps of the proof are as follows. First, we argue that the F_{ATH}-, F_{BS}-hybrid protocol UC-securely realizes ideal functionality F_{CBDC}. We assume that all communication takes place over a secret channel (F_{SC}-hybrid). So, actually it is an F_{SC}-, F_{ATH}-, F_{BS}-hybrid protocol. The task here is to show that a successful simulator exists (see the details in section 6.1). As the next step, we refer to the universal composition theorem of Canetti [4]. □

7.1 Simulation of the F_{ATH}-, F_{BS}-hybrid protocol

Several factors simplify the task of simulation. First, we notice that the authorization and blind signatures are the (only) cryptographic operations within the protocol. Furthermore, in the simulation of the hybrid protocol, these operations are simulated via the simulation of the corresponding ideal functionalities (including the simulation of the generation of keys).

The session identifier ((sid, ssid)) is composed of the main identifier sid and sub-identifier ssid. The main identifier (sid) connects all instances of sub-protocols that use the same instance of F_{BS}, i.e. the same set of Exchange's key pairs.

Sub-identifiers are unique to sub-protocol instances. This models that a fresh instance of communication channel (F_{SC}) is assigned to a new subprotocol instance (i.e. a fresh set of communication keys). Notice, this is a realistic assumption about the usage of the protocol. Indeed, on the one hand, different key sets are assigned to different communication directions (between C and E (withdrawal/refreshing), between C and M (spending), between M and E (deposit)). On the other hand, the same operations between the same parties are separated well in time and the parties establish fresh sessions for each such operation. (We comment, that the methodology can be extended to more general communication modes. Furthermore, subprotocols could be defined to handle several coins per session.)

The real question to consider regarding compositional technology is the shared use of functionalities F_{ATH} and F_{BS} by different subprotocol instances. Note, the same instance of F_{ATH} is shared by operations related to a given coin. Similarly, the same instance of F_{BS} is shared by all operations during the lifetime of an Exchange's key set. In other words, authorization and blind signature cause algorithmic dependence between different protocol instances.

There are two possible approaches for resolving this dependence. One of them is that dependent instances are analyzed together. Accordingly, we analyze together all subprotocol instances related to the life-cycle of a coin (dependent via a shared instance of F_{ATH}). For the sake of simplicity, we refer to such a set of subprotocol instances as a life-cycle instance.

Different life-cycle instances are dependent on the shared blind signature functionality F_{BS}. For resolving this dependence we use the Joint state UC (JUC) approach of Canetti [4]. Accordingly, under a certain condition, it is sufficient to prove the (base-) UC-security of a life-cycle instance, where each such instance has access to a dedicated, fresh (i.e. independent) instance of the blind signature functionality F_{BS}. The condition is the existence of a blind signature realization that UC-securely realizes the so-called multi-session extension ideal functionality. Such a („JUC-capable") realization can be obtained by the technique of prepending the ssid to the message before it is signed.

In summary, it is sufficient to prove the (base-) UC security of a life-cycle instance, where an instance is a hybrid in fresh F_{SC}-, F_{ATH}-, F_{BS}- subroutines. Specifically, we have to show the corresponding straight-line simulators. The task of simulation is simplified by the fact that subinstances within a life-cycle instance (of a coin) follow sequentially in time, thus the simulatability of the life-cycle instance can be demonstrated by showing the simulation for the component operations separately.

A straight-line simulation is shown for the (hybrid) Withdrawal subprotocol below. Each message transfer uses the F_{SC} communication functionality. For the clarity of presentation, this communication step is not shown explicitly. The simulation can be performed in a similar way for the other subinstances (i.e. without the need for additional technical tricks). For this reason, in this paper, we do not detail them to keep the focus and to keep the size of the paper reasonable.

The simulation is essentially straightforward because all cryptographic operations are within the ideal subroutines of the hybrid that these subroutines are simulated by the simulator. (Specifically, (usual) hard tasks for a UC-simulation, such as the extraction of inputs or the equivocality are missing.) The parties of the simulation are as follows: environment (Z), simulator (Sim), (real) adversary (A), and the parties of the protocol (C, M, E). The corruption scenario in the example simulation below is the following: party C is malicious and party E is honest.

The simulation consists of sending a relatively large number of messages. For the sake of clarity, the process of simulation is not given in standard text description, but using simplification notations. We use the following notations:
X(Y): party Y acts on behalf of X
Z/X: I/O port between Z and X
X $\rightarrow\!/$ Y: party X intends to send a message to Y, but the message is stopped and processed by Sim

The steps of simulation for the (hybrid) Withdrawal subprotocol are shown in Figure 8. in concise description.

Z/C →/ F_{CBDC}: (***WithdrawRequest***, sid, ssid, d) (ssid=(C1, E, ssid'))
Z(Sim) → C(A): (***WithdrawRequest***, sid, ssid, d)
C(A) → E(Sim): u=(sid, ssid, d')
Z(Sim) → F_{CBDC}: (***WithdrawRequest***, sid, ssid, d')
F_{CBDC} →Z/E : (***WithdrawRequested***, sid, ssid, d')
Z/E →F_{CBDC}: (***WithdrawDecision***, sid, ssid, b)
F_{CBDC} →/ Z/C: (***WithdrawDecision***, sid, ssid, b)
E(Sim)→ C(A): v=(sid, ssid, b)
C(Sim)→ Z/C: (***WithdrawDecision***, sid, ssid, b)
Sim: aborts if b=0 else proceeds
(F_{CBDC}: aborts if b=0 else proceeds)
C(A)→F_{ATH}(Sim): (***AuthorKeysReq***, sid, ssid)
F_{ATH}(Sim) → C(A): (***AuthorVerif***, sid, AuthorVerif)
C(A)→F_{BS}(Sim): (***BlindSign***, sid, ssid ,m,$V_{BS,d}$), where m=(d', coinId), coinId=Id(AuthorVerif)
F_{BS}(Sim) → C(A): (***BlindSign***, sid, ssid ,m, coin_sign)
F_{CBDC} → Sim: (***BlindSign***, sid, ssid ,m",$V_{BS,d}$), where m=(d', coinId"), coinId"=Id(AuthorVerif")
Sim → F_{CBDC}: (***BlindSign***, sid, ssid , m", coin_sign"),
F_{CBDC} →/ Z/C: (***Withdrawn***, sid, ssid, coin")
C(Sim) → Z/C: (***Withdrawn***, sid, ssid, coin")
(F_{CBDC} → Z/E: (***Withdrawn***, sid, ssid))
□

Figure 8: Simulation for the (hybrid) Withdrawal subprotoco

8. Conclusions and Future Scope

We proposed a protocol for the CBDC task such that it is provably secure in general concurrent network environment. We argued that such a strong execution environment model is the realistic one. We followed the universal composability approach in the design and analysis of the protocol. As part of this design we proposed an ideal functionality for the CBDC task and we pointed out that all relevant security requirements are implied, such as transaction anonimity, unforgeability of digital coins and prevention of the intention to evade taxation. We defined the necessary assumptions (including trust assumptions) for the UC-secure implementation of that ideal functionality.

We notice that a („standard") design based on the classic RSA-based blind-signature technology does not provide a simulation-based composable solution, not to mention universal composability. A critical point in this regard is the handling of hash mappings. The random oracle model is a standard mitigation step in this respect in case of a stand-alone design (with full domain hash, FDH). If we would like to „force" such a design direction in a concurrent setting an assumption about the availability of a global random oracle would be required which is far from any practical outcome concerning the CBDC. We will not analyze further this technological direction in this work.

Extensions:

As we mentioned in the introduction, in this paper we wanted to show the design of a UC-secure CBDC protocol for a CBDC ideal functionality guaranteeing convincingly strong security. We stress that this methodology can be used for other variants or extensions of the ideal functionality. These extensions could be done in future work. Here we give a brief overview of two ways extensions. One of them relates to the problem of mutually distrusting customer and merchant, the other is an extension to malicious exchange model.

The first problem relates to the secure timing of the exchange of the goods and the coin paid for. For the sake of clarity in paper size, our ideal functionality will not formally address this problem. However, informally, we give an idea for a corresponding extension of the protocol.

The exchange of goods and coin could happen in several rounds. First the customer sends the coin to the merchant without authorizing the merchant to deposit the coin at the Exchange. The merchant could send the coin to the Exchange to verify its validity. After successful verification, the Exchange records the coin together with the transaction ID. When the Exchange gives positive feedback, the merchant delivers the goods to the customer. In response, the customer sends its authorization signature to the merchant and the latter forwards it to the Exchange. Note, a dishonest customer will be forced to send the authorization signature, otherwise, the bank might use a forced deposit in favor of the merchant.

The model for the dishonest behavior of the Exchange could be changed to include malicious corruption. (It is not to say that the central bank as an institution would be malicious, but that the bank's computers running the CBDC protocol could have been corrupted by an attacker.) Here we mention two possible directions.

The first is to give security guaranties even in case of an Exchange such that it chooses its (blind signature) secret keys maliciously. This is an obvious extension as the generic framework of Fischlin [7] for constructing round-optimal UC-secure blind signatures guarantees blindness under malicious signer's keys.

Another important extension for malicious behavior is as follows. It is a standard banking requirement that a customer's money assets managed by the bank should not be damaged due to possible system failures, in particular, due to failures in the successful execution of the protocol. Let's consider a malicious attack when an adversary interrupts the execution of the protocol and prevents its finishing according to the specification. The requirement is that in the event of such an attack, the bank's customer should not suffer any financial loss.

Such an adversary could be a MIM network adversary or a malicious Exchange. Note, though we have assumed secret channels this fact does not prevent a MIM adversary from deleting messages from the channel (this is because the „traffic" remains to be seen by such an adversary). On the other hand, a malicious Exchange might abort the execution of an instance at an attackable point to prevent the other party from receiving its due messages.

For example, consider the withdrawal protocol. A corresponding security concern is that the external part of the Exchange (a commercial bank) deducts the value of the coin from the customer's account but the coin is not produced due to abortion. Note, the final operation of the withdraw subfunctionality is to send an acknowledgment message to the Exchange's output. (Before this, the coin is sent to the customer.) If the (honest) external part of the Exchange does not receive the ACK message, the account should be restored to its original state.

The general requirement is that the view of the current state of the account balance must remain in sync at the customer and the Exchange. Two key elements could support this goal. The customer should receive a notification message when any change has been made to its bank account, and until the (honest) customer has acknowledged the success of the coin operation, its account balance should have to be restorable to its original state.

Setups and their realization

A "plausible" practical implementation of secure communication links is protocol TLS. The TLS has UC secure implementation [9]. Standard PKI setup is required.

Recall, universally composable security guaranties that a UC-secure protocol behaves just like an ideally secure functionality in whatever arbitrary computational environment it is run. Of course, such a strong guarantee is not free. One component of the price is the assumption of the Common Reference String (CRS) setup concerning the UC-secure blind signature subroutine. The usual concern with such an assumption is that we assume the availability of an entity that is ideally non-corrupt. Nevertheless, the CRS setup is beginning to become accepted in (potentially large-scale) practical applications after its previous almost purely theoretical role. For example, the well-known cryptocurrency Zcash with z-Snark also requires a CRS setup [2].

By the adversarial model of this paper, the implementation of such a setup is less "problematic" as we assume an honest-but-(potentially) curious (i.e. semi-honest) central bank. In particular, such an assumption precludes manipulation of the distribution of CRS strings by the Exchange. The CRS module/server could be deployed at the site of the Exchange (central bank). Note additionally that by assumption customers have authenticated communication channel with the Exchange that can also be used to distribute CRS strings. Furthermore, the "help" provided by setup is only needed at the beginning of the protocol run, as once generated common reference string can be used for several signature generations by different users [7].

The AuthorSign algorithm is assumed to be run on a trusted module (implemented on a trusted firmware component within the wallet of a customer). This assumption means that no corruption adversary can influence the execution of these algorithms. The trusted module stores the secret authentication keys. Availability of such a level of security at a reasonable price (per wallet) may become a reality in case of mass demand, which can become real during the expected global deployment of CBDC technology over the next few years.

References

[1] R. Auer, R. et.al. (2021), 'Central bank digital currencies: motives, economic implications and the research frontier', BIS Working Papers, No 976

[2] Bitansky, N. et.al. (2012), 'From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again', Theory of Cryptography Conference, TCC 2013: Theory of Cryptography, pp. 315-333

[3] Canetti, R.(2001),'Universally Composable Security: A New Paradigm for Cryptographic Protocols', In 42nd FOCS, pp. 136-145

[4] Canetti, R. (2005), 'Universally Composable Security: A New Paradigm for Cryptographic Protocols'. http://eprint.iacr.org/2000/067, revision of 2005

[5] Chaum, D. (1983), 'Blind signatures for untraceable payments', In CRYPTO 1982, Plenum Press, pp. 199–203

[6] Chaum, D. et.al. (2021), 'How to issue a central bank digital currency', Swiss Nationalbank (SNB) Working Papers, 3/2021

[7] Fischlin, M. (2006), 'Round-Optimal Composable Blind Signatures in the Common Reference String Model', CRYPTO, August 2006

[8] S. Garera, S. et.al. (2007), ’A framework for detection and measurement of phishing attacks’. In: Proceedings of the 2007 ACM workshop on Recurring malcode, ACM 2007, pp. 1–8

[9] Gajek, S. et. al. (2008), ’Universally Composable Security Analysis of TLS’, In: Baek J., Bao F., Chen K., Lai X. (eds) Provable Security. ProvSec 2008. Springer LNCS 5324, pp. 313-328

[10] General Data Protection Regulation (EU) 2016/679

[11] Juels, A. et. al. (1997), ’Security of blind digital signatures’, In Advances in Cryptology – CRYPTO’97, Springer LNCS 1294, pp. 150–164

[12] Kiayias, A. and Zhou, H.S. (2007), ’Equivocal Blind Signatures and Adaptive UC-Security’, Workshop on Cryptographic Protocols (WCP'07), https://www.iacr.org/archive/ tcc2008/49480334/49480334.pdf

[13] Lindell, Y. (2003), ’Bounded-Concurrent Secure Two-Party Computation Without Setup Assumptions’, Proceedings of the Annual Symposium on the Theory of Computing (STOC) 2003, ACM Press, pp. 683–692

[14] Pointcheval, D. and Stern, J. (2000), ’Security arguments for digital signatures and blind signatures’, Journal of Cryptology, 13(3), pp. 361–396

[15] Sahin, Y. and Duman, E. (2010), ’An overview of business domains where fraud can take place, and a survey of various fraud detection techniques’. In: Proceedings of the 1st international symposium on computing in science and engineering, Aydin, Turkey. 2010

Appendix

A1: Definition of the (real) protocol

1. Upon receiving a value (***BlindSignKeysReq***, sid) from from Z, party E verifies that sid = (E, sid') for some sid'. If not so, then party E ignores the input, else it generates key set $\{V_{BS,d}, \text{BlindSign}_d\}_{d \in D}$ and sends a message (***BlindSignVerifKey***, sid, $\{V_{BS,d}\}_{d \in D}$) to Z. □

Figure 6. Exchanges's keys generation

1. Upon receiving an input (***WithdrawRequest***, sid, ssid, *d*) from Z, party C1 verifies (sid, ssid), ssid=(C1, E, ssid'). If not so, then party C ignores the input, else sends message u=(sid, ssid, *d*) to party E.
Upon receiving message u, party E makes local output (***WithdrawRequested***, sid, ssid, *d*) to Z.
2. Upon receiving an input (***WithdrawDecision***, sid, ssid, b) from Z, party E sends message v=(sid, ssid, b) to party C.
Upon receiving message v, party C1 makes a local output (***WithdrawDecision***, sid, ssid, b) to Z.
If b=0, then party C1 aborts sub-session (sid, ssid), else it proceeds as follows:

- generates a pair of algorithms (AuthorVerif, AuthorSign),
- makes a local output (***AuthorVerif***, sid, AuthorVerif) to Z
- runs blind signature protocol with party E on input (m,$V_{BS,d}$), m=(d, coinId), coinId=Id(AuthorVerif) and obtains signature coin_sign ← Blind_sign(m,$V_{BS,d}$),
- stores coin= (sid, d, ToE, coinId, coin_sign, fresh),
- stores coin-keys (sid, coinId, AuthorVerif, AuthorSign),
- makes a local output (***Withdrawn***, sid, ssid, coin) to Z,
- sends message w=(sid, ssid, d) to party E.

Upon receiving message w, party E makes local output (***Withdrawn***, sid, ssid) to Z. □

Figure 7. Withdrawal

1. Upon receiving an input (***Spend***, sid, ssid, *transactionId, coin*) from Z, party C2 verifies (sid, ssid), ssid=(C2, M, ssid'). If not so, then party C2 ignores input, else it proceeds as follows:

- verifies that coin is spendable (status is fresh/refreshed)
- updates coin's status to *spent*,
- retrieves algorithm AuthorSign from coin-keys record (sid, coinId, AuthorVerif, AuthorSign),
- computes signature depositPermit ←AuthorSign (transactionId, coin),
- makes a local output (***Spent***, sid, ssid) to Z,
- sends message p=(sid, ssid, tansactionId, coin, deposit Permit) to M.

Upon receiving message p, party M makes local output (**Spent**, sid, ssid, tansactionId, coin, deposit Permit) to Z. □

Figure 8. Spending

1. Upon receiving an input (***DepositRequest***, sid, ssid, tansactionId, coin, deposit Permit) from Z, party M verifies (sid, ssid), ssid=(M, E, ssid'). If not so, then it ignores input, else it sends message t=(sid, ssid, tansactionId, coin, deposit Permit) to party E.
Upon receiving message t, party E verifies that {coin is not deposited/expired} AND {coin's blind signature is valid}AND {authorization of deposit is valid} and if not so, then party E aborts subsession (sid, ssid), else it makes a local output (***Deposit_requested***, sid, ssid, d) to Z.
2. Upon receiving an input (***Deposit_accept***, sid, ssid, d) from Z, party E proceeds as follows:

- stores the coin with status *deposited,*
- *s*ends message t=(sid, ssid, tansactionId, coin, "deposited") to party M.

Upon receiving message t, party M makes local output (**Deposited**, sid, ssid) to Z. □

Figure 9. Deposit

1. Upon receiving an input (***RefreshAuthorSignRequest***, sid, ssid, coin) from Z, party C2 verifies (sid, ssid), ssid=(C2, E, ssid'). If not so, then ignore input, else it proceeds as follows:

- verifies that the status of coin is fresh/refreshed
- retrieves coin-keys record (sid, coinId, AuthorVerif, AuthorSign),
- computes refreshPermit ←AuthorSign (coin),
- stores refresh-permit (sid, coin, refreshPermit),
- makes a local output (***RefreshAuthorSign***, sid, ssid, coin, refreshPermit) to Z.

2. Upon receiving an input (***RefreshRequest***, sid, ssid, coin, refreshPermit) from Z, party C2 proceeds as follows:

- sends message s=(sid, ssid, coin, refreshPermit) to party E,
- upon receiving message s, party E verifies that {coin is not deposited/expired} AND {rfcnt<RFCNT}AND {coin's blind signature is valid}AND {authorization of deposit is valid} and if not so, then party E aborts subsession (sid, ssid), else it
- makes a local output (***RefreshRequested***, sid, ssid, d, ToE, rfcnt) to Z.

3. Upon receiving an input (***RefreshPermit***, sid, ssid, b) from Z, party E sends message (sid, ssid, b) to party C2.
Upon receiving message (sid, ssid, b), if b=0, then party C2 aborts the session (sid, ssid), else it proceeds as follows:

- generates a pair of algorithms (AuthorVerif, AuthorSign),
- makes a local output (***AuthorVerif***, sid, AuthorVerif) to Z
- runs blind signature protocol with party E on input (m,$V_{BS,d}$), m=(d, coinId), coinId=Id(AuthorVerif) and obtains signature coin_sign ← Blind_sign(m,$V_{BS,d}$),
- increments refresh counter (rfcnt) by 1
- replaces coin with coin= (sid, d, ToE, coinId, coin_sign, fresh),
- stores coin-keys (sid, coinId, AuthorVerif, AuthorSign),
- makes a local output (***Refreshed***, sid, ssid, coin) to party Z,
- sends message z=(sid, ssid, d) to party E.

Upon receiving message z, party E makes local output (***Refreshed***, sid, ssid) to Z. □

Figure 10. Refreshing

A2: Ideal functionality F_{BS}

Our definition (see in Figure 11) is an adaptation of Fischlin's blind signature functionality [7] with minor modifications, where we take into account that in our model a dishonest Exchange (a bank in Fischlin's wording) is semi-honest. In contrast, users can be honest or malicious (like Fischlin [7]). These changes somewhat simplify the rules of verification:

If a (message, signature) pair (m, S) is generated by the ideal functionality it gets a flag value f=1. If an honest party requests the signature, then it is generated by the Blind_sign protocol, else the adversary is asked to generate the signature. When a pair (m,S) arrives for verification such that the signature has not been generated by the ideal functionality and does not verify it gets a flag value 0. If later on the adversary attempts to „legalize" such a non-valid pair via requesting a signature for message m on behalf of a corrupted party, such an action would lead to an unresolvable contradiction (indeed, the adversary could set S as a"legal" signature). The only way to resolve the contradiction is to halt the execution of the instance.

1. Upon receiving a value (***Keygen***, sid) from a party E, ideal functionality verifies that sid = (E, sid') for some fresh value sid'. If not so, then it ignores the input, else it hands the message (***BlindSignKeygen***, sid) to the adversary.
Upon receiving message (***BlindSignKeys***, sid, $\{V_{BS,d}\}_{d\in D}$, $\{BlindSign_d\}_{d\in D}$) from the adversary, it outputs (***BlindSignVerifKey***, sid, $\{V_{BS,d}\}_{d\in D}$) to E and records (sid, $\{V_{BS,d}\}_{d\in D}$, $\{BlindSign_d\}_{d\in D}$) .
2. Upon receiving an input (***BlindSign***, sid, m, $V_{BS,d}$) from party C, it verifies sid=(E, sid'). If not so, then it ignores the input, else it proceeds as follows:
If C is honest then it does:

- runs blind signature protocol on input (m,$V_{BS,d}$) and obtain S $\leftarrow$ Blind_sign(m,$V_{BS,d}$),
- sends output (***Signature***, sid, S) to C,
- sends output (***Signature,*** sid) to E.

else it does:

- sends (***BlindSign***, sid, m, $V_{BS,d}$) to the adversary to obtain (***Signature***, sid, m, S),
- ***halts*** if (m, S, $V_{BS,d}$, 0) has been recorded before,
- sends output (***Signature***, sid, S) to C,
- sends output (***Signature,*** sid) to E.

In either case, it records (m, S, $V_{BS,d}$, 1).
3. Upon receiving an input (***Verify***, sid, m, S, V'_{BS}) from party C, verify sid=(E, sid'). If not so, then it ignores the input, else it does:

- if $V'_{BS} \neq V_{BS,d}$, then it ignores the input,
- if {there exists a record (m, S,$V_{BS,d}$, f)}, then it sends output (***Verified***, sid, m, S, f) to C,
- if there exists no record (m, .,$V_{BS,d}$, 1) then it records (m, S, $V_{BS,d}$, 0) and sends output (***Verified***, sid, m, S, 0) to C. □

Figure 11. Ideal functionality Blind signature

A3: Ideal functionality F_{ATH}

There is a unique correspondence between a pair of authorization algorithms/keys (AuthorVerif, AuthorSign) and a coin. The ideal functionality of authorization signature F_{ATH} essentially follows Canetti's F_{SIG} digital signature functionality. The definition of F_{ATH} is somewhat simpler (Figure 12). The signature generation rule does not have special rules for the honest and corrupt signers, respectively. The same is true for the verification rules. This follows from our assumed system model that the secret authorization algorithm runs within a trusted module (a trusted component of the wallet). This assumption implies that any party (honest or not) generates a valid signature when it runs the AuthorSign algorithm.

1. **Key generation**: Upon receiving an input (***AuthorKeysReq***, sid) from a party C, ideal functionality verifies that sid = (C, sid'), for some fresh sid'. If not so, then it ignores the input, else it does:

- generates a pair of algorithms (AuthorVerif, AuthorSign)
- outputs (***AuthorVerif***, sid, AuthorVerif) to C
- records (C, sid, AuthorVerif, AuthorSign).

2. Signature generation: Upon receiving an input (***AuthorSign***, *sid, m*) from C, it does:

- retrieves an algorithm AuthorSign from a matching record (C, sid, . , AuthorSign),
- computes σ= AuthorSign(m),
- records (C, sid, m, σ),
- outputs (***AuthorSignature***, *sid,* m, σ) to C.

3. Verification: Upon receiving an input (***AuthorVer***, *sid, m*, σ, AuthorVerif ') from a party M, it does:

- retrieves an algorithm AuthorVerif from a matching record (., sid, AuthorVerif , . , .),
- if { AuthorVerif (m, σ) = 1} AND {no entry (., ., m, σ) is recorded}, then outputs an "error" message to M and halts (an event of a successful forging), else it
- outputs (***AuthorVerified***, sid, m, σ, AuthorVerif ' (m, σ)) to M.

Figure 12. Ideal functionality Authorization