Collusion attacks and fair time-locked deposits for fast-payment transactions in Bitcoin

Xingjie YU
Singapore Management University, xjyu@smu.edu.sg

Shiwen Michael THANG
Singapore Management University, swthang.2015@mais.smu.edu.sg

Yingjiu LI
Singapore Management University, yjli@smu.edu.sg

Robert H. DENG
Singapore Management University, robertdeng@smu.edu.sg

Collusion attacks and fair time-locked deposits for fast-payment transactions in Bitcoin[1]

Xingjie Yu [*], Michael Shiwen Thang, Yingjiu Li and Robert Huijie Deng
School of Information Systems, Singapore Management University, Singapore
E-mails: stefanie_yxj@hotmail.com, swthang.2015@mais.smu.edu.sg, yjli@smu.edu.sg, robertdeng@smu.edu.sg

Abstract. In Bitcoin network, the distributed storage of multiple copies of the block chain opens up possibilities for double-spending, i.e., a payer issues two separate transactions to two different payees transferring the same coins. While Bitcoin has inherent security mechanism to prevent double-spending attacks, it requires a certain amount of time to detect the double-spending attacks after the transaction has been initiated. Therefore, it is impractical to protect the payees from suffering in double-spending attacks in fast payment scenarios where the time between the exchange of currency and goods or services is shorten to few seconds. Although we cannot prevent double-spending attacks immediately for fast payments, decentralized non-equivocation contracts have been proposed to penalize the malicious payer after the attacks have been detected. The basic idea of these contracts is that the payer locks some coins in a deposit when he initiates a transaction with the payee. If the payer double-spends, a cryptographic primitive called accountable assertions can be used to reveal his Bitcoin credentials for the deposit. Thus, the malicious payer could be penalized by the loss of deposit coins. However, such decentralized non-equivocation contracts are subjected to collusion attacks where the payer colludes with the beneficiary of the depoist and transfers the Bitcoin deposit back to himself when he double-spends, resulting in no penalties. On the other hand, even if the beneficiary behaves honestly, the victim payee cannot get any compensation directly from the deposit in the original design.

To prevent such collusion attacks, we design fair time-locked deposits for Bitcoin transactions to defend against double-spending. The fair deposits ensure that the payer will be penalized by the loss of his deposit coins if he double-spends and the victim payee's loss will be compensated within a locked time period. We start with the protocols of making a deposit for one transaction. In particular, for the transaction with single input and output and the transaction with multiple inputs and outputs, we provide different designs of the deposits. We analyze the performance of deposits made for one transaction and show how the fair deposits work efficiently in Bitcoin. We also provide protocols of making a deposit for multiple transactions, which can reduce the burdens of a honest payer. In the end, we extend the fair deposits to non-equivocation contracts for other distributed systems.

Keywords: Bitcoin, fair deposit, double-spending, collusion attacks, time-locked

1. Introduction

As a decentralized crypto-currency system, to eliminate the central bank, Bitcoin uses blockchain to take the role of a distributed ledger. This enables every participant to keep a copy of transaction records

[1]This paper is an extended journal version of our previous conference paper (In *IEEE Conference on Dependable and Secure Computing* (2017)).
[*]Corresponding author. E-mail: stefanie_yxj@hotmail.com.

which would classically be stored at central banks in traditional banking system. The distributed storage of multiple copies of the blockchain opens up possibilities for double-spending attacks. A double-spending attack is launched by a malicious payer, i.e., a payer who performs two separate transactions with two different payees transferring the same coins. Bitcoin addresses this problem by, in a sense, letting the entire network verify the legitimacy of the transactions, so that double-spending can be detected by other participants. The payee is advised to accept the transaction after the block containing his/her transaction is confirmed as at least k blocks deep in the consensus blockchain. This is because, if a payee waits for the transaction to advance into the blockchain a number of k blocks, then the probability that an attacker can build an alternative blockchain that reorganizes the public blockchain (which contains the double-spending transaction) drops exponentially with k [16].

Nowadays, Bitcoin is accepted worldwide in a number of fast payment scenarios, where the time between the exchange of Bitcoin currency and goods or services should be shorten to a few seconds. For example, different types of vending machines that accept Bitcoin were deployed a few years ago [10,31], and the number of Bitcoin vending machines continues to grow recently [30]. Other examples demanding Bitcoin fast-payment transactions include fast-food payments, e.g., in USA [36], Singapore [26] and Russia [33], and marketplace in-store payments, e.g., in South Africa [19] and Japan [12]. Although payees can be effectively protected from double-spending attacks if they accept the transactions till the blocks containing their transactions are at least k blocks deep, it requires payees to wait for far longer than a few seconds and is therefore impractical for fast payments.

However, if a payee accepts a transaction immediately, it is of limited value to detect the double-spending attack after a malicious payer has already obtained the goods or services due to the fact that Bitcoin users are anonymous and that users can hold multiple accounts. Researchers have demonstrated that unless appropriate countermeasures are integrated in current Bitcoin implementations, double-spending attacks on fast payments could succeed with overwhelming probability and can be mounted at low cost, thus compromising the trustworthiness and economic standing of Bitcoin [20]. It is important to protect payees from double-spending attacks without increasing transaction pending time in fast payment scenarios.

Although current Bitcoin transaction confirmation models cannot provide realtime prevention of double-spending attacks for fast payments, we still can penalize the dishonest payer after a double-spending attack has been detected. Ruffing et al. [34] proposed a non-equivocation contract which could be applied in Bitcoin to penalize the double-spending payer by loss of bitcoins. Their scheme is based on the idea of a time-locked Bitcoin deposit that can be opened by a beneficiary (i.e., a person who is supposed to detect a payer's double-spending and extract the payer's private key if the payer doubly spends) upon the detection of double-spending. After the deposit is confirmed by Bitcoin network, the payer sends an accountable assertion for each fast-payment transaction to the payee along with the transaction information, who forwards them to the beneficiary for storing. If the payer double-spends, the beneficiary can extract the payer's secret key with two conflicting assertions, then transfer all funds in the deposit to his/her own Bitcoin account. If the payer behaves honestly, he/she will regain full control of the deposit after the time-lock expires. Therefore, by setting aside a high-enough deposit, it is expected that the malicious payer would have no incentive to double-spend his/her fast-payment transactions.

However, such non-equivocation contracts are subjected to collusion attacks where a malicious payer can collude with the beneficiary to transfer the Bitcoin deposit back to himself/herself when he/she double-spends, resulting in no penalties. Moreover, this deposit is unfair to a victim payee (i.e., a payee who suffers from double-spending). This is because whether the payer colludes with the beneficiary or not, there is no compensation to the victim payee since the deposit is made for the beneficiary only. In a business perspective, the victim payee, who may be a service provider or a product seller, may lose valuable time and effort, and may result in a loss of profits. A solution for preventing double-spending in fast payment scenarios with the guarantee of compensation to the victim payee's loss is in demand.

Our goal is to design new protocols for making time-locked deposits to not only effectively defend against double-spending in Bitcoin fast payments, but also prevent collusion attacks and guarantee the compensation to a victim payee's loss. We first provide a solution to make a deposit for one transaction, which is first introduced in our conference paper [38]. In our protocol, the payer needs to create a time-locked deposit for his/her transaction with the payee, so that the payer could be penalized by the loss of his/her deposit if he/she double-spends within the locked time period. Our protocol ensures that the payee's loss is compensated in the case of double-spending.

In Bitcoin network, each transaction requires certain transaction fee to be paid to miners. To reduce the transaction fees for a honest payer who has never double-spent and has no intention of equivocating in the future, we design a protocol allowing a payer to make just one deposit for multiple transactions, which is an extension of our conference paper [38]. In particular, a payer can make a deposit for multiple transactions. If the payer double-spends any transaction among the transactions which are protected under the deposit, our protocol ensures that the payee of the double-spending transaction get a fair compensation from the deposit.

To make it more convenient for a payer and a payee to set up a deposit, the beneficiary of the deposit can be a randomly selected miner rather than an explicit beneficiary. Accordingly, we provide an extension of making deposits without any explicit beneficiary in this paper. We also extend our solution to non-equivocation contracts. Compared to the non-equivocation contracts proposed in [34], our non-equivocation contracts not only penalize the equivocating party but also compensate a victim party's loss. In addition, our non-equivocation contracts are resistent to the collusion attacks between the equivocating party and the beneficiary.

The rest of this paper is organized as follows. Section 2 introduces the preliminaries. Section 3 presents our threat model in Bitcoin network. Section 4 presents the design of a fair time-locked deposit for one Bitcoin transaction. Section 5 implements the setup and usage of fair deposits in Bitcoin network and evaluates its usability and efficiency. Section 6 introduces the design of fair deposits for multiple Bitcoin transactions. Section 7 provides an extension of our design to deposits without explicit beneficiaries. Section 8 introduces an extension of our scheme to non-equivocation contracts applicable to various distributed systems. Section 9 describes the related work. Section 10 concludes this paper.

2. Preliminaries

In this section, we introduce preliminary facts of non-equivocation contracts, time-locked deposit, and accountable assertion scheme, which are used throughout this paper.

2.1. Non-equivocation contracts

Ruffing et al. [34] proposed a non-equivocation (i.e., making conflicting statements) contract for distributed system to penalize the party who equivocates by losing Bitcoins. The protocol is based on the idea of a time-locked Bitcoin deposit that can be opened by a predefined beneficiary in the case of an equivocation. By setting aside a high-enough deposit with a beneficiary, it is expected that the malicious payer would have no incentive to equivocate. In their protocol, to use a time-locked deposit for a transaction, Party A creates a Bitcoin key pair and sets up the accountable assertion scheme with the same key pair. A then creates a deposit with a third party. The payee(s), Party B, waits till the deposit has been confirmed by the Bitcoin network and then receives the statement and the assertion from A. B then verifies the assertion and if it is valid, forwards it to the beneficiary. If the beneficiary detects an equivocation in two records specifying the context, statements, and assertions, he/she can extract A's secret key and use it to transfer the funds in the deposit to his/her own Bitcoin address. Else, A will regain full control of the deposit after the time-lock expires.

This protocol, however, may be subjected to a collusion attack since only the payer's signature is required in the output script of the deposit transaction. The malicious payer can collude with the beneficiary to transfer the Bitcoin deposit back to himself/herself when he/she double-spends, resulting in no penalties. On the other hand, even if the payer has double-spent the transaction and the beneficiary has behaved honestly, there is no compensation to the victim payee since the coins locked in the deposit can only be redeemed by the beneficiary.

2.2. *Time-locked deposits*

We create time-locked deposit based on a script command denoted by CheckLockTimeVerify (CLTV) which has been newly merged into Bitcoin Core. CLTV allows users to create a bitcoin transaction of which the transaction outputs are spendable only at some point in the future. As such, the coins sent in that transaction are time-locked until either a specified date, or until a certain number of blocks has been mined.

The payer who creates the time-locked deposit cannot transfer the coins in the deposit with his/her secret key before the lock time $T = T_{\text{net}} + T_{\text{conf}} + T_{\text{def}}$, where $T_{\text{net}} + T_{\text{conf}}$ is the safety margin of the lock time, and T_{def} is the additional lock time added on the safety margin. T_{def} should be decided based on the negotiation between the payer, the payee and the beneficiary if it is necessary. The safety margin ensures that the closing transaction has already been confirmed by the Bitcoin network before the deposit can be spent by the payer alone.

In particular, T_{net} should ensure sufficient time for the deposited transaction to be included in a block and broadcasted to the Bitcoin network. Since the proof-of-work (POW) of one block in the Bitcoin network takes 10 mins on average, $T_{\text{net}} = 10$ min can sufficiently ensure the broadcast of the deposited transaction [13]. T_{conf} should ensure sufficient time for the deposited transaction to be confirmed in Bitcoin network, i.e., the deposited transaction is out of danger of being double-spent. In the Bitcoin network, a transaction is considered as confirmed if it has been backed up by least six blocks. As suggested by Ruffing et al. [34], the six desired blocks that back up a transaction could be secured after 24 blocks have been added in Bitcoin network. This is because the arrival of blocks in Bitcoin network is Poisson-distributed, and the probability that fewer than six desired blocks have been found is $\Pr[X \leqslant 5] < 2^{-18}$ for $X \sim \text{Pois}(24)$, where X is an integer greater than 0. Hence, $T_{\text{conf}} = 240$ min is sufficient to secure the confirmation of the transaction. Throughout the paper, the lock time of a deposit should be longer than the safety margin $T_{\text{net}} + T_{\text{conf}} = 250$ min.

2.3. *Accountable assertion scheme*

An accountable assertion is a cryptographic primitive introduced by Ruffing et al. [34]. The idea of this promitive is to bind *statements* to *contexts* in an accountable way: if the payer equivocates, i.e., asserts two contradicting statements in the same *context*, then any observer can extract the payer's Bitcoin secret key and, as a result, use it to force the loss of the payer's funds. We use the accountable assertion scheme to detect double-spending transactions, and extract the secret key if double-spending happens. Accountable assertions are constructed based on chameleon hash function which is a collision-resistant hash function that allows a user to compute collisions efficiently using a trapdoor. It supports the extractability property where a deterministic polynomial time algorithm exists which reveals the secret key when a collision occurs. An accountable scheme includes four algorithms: key generation, assertion, verification and extraction. An accountable assertion scheme is defined as follows:

- $(apk, ask, auxsk) \leftarrow \mathrm{Gen}(1^{\lambda})$: The key generation algorithm outputs a key pair consisting of a public key apk and a secret key ask, and auxiliary secret information $auxsk$. It is required that for each public key, there is exactly one secret key.
- $\tau / \perp \leftarrow Assert(ask, auxsk, ct, st)$: The assertion algorithm takes as input a secret key ask, auxiliary secret information $auxsk$, a *context* ct, and a *statement* st. It returns either an assertion τ or $\perp$ to indicate failure.
- $b \leftarrow Verify(apk, ct, st, \tau)$: The verification algorithm outputs 1 if and only if τ is a valid assertion of a statement ct in the *context* st under the public key apk.
- $ask \leftarrow Extract(apk, ct, st_0, st_1, \tau_0, \tau_1)$: The extraction algorithm takes as input a public key apk, a *context* ct, two *statements* st_0, st_1, and two assertions τ_0, τ_1. It outputs either the secret key ask or $\perp$ to indicate failure.

3. Threat model

In our threat model, a payer can maliciously double-spend an input of a Bitcoin transaction, but he/she cannot control more than 50% computation power. Rosenfeld [32] has demonstrated that a malicious payer with less than 50% of the total computational power is able to perform a double-spending by brute force and a bit of luck. If a malicious payer controls more than 50% computational power, he/she can double-spend any transaction, including the deposit transaction. Even worse, such malicious payer has a chance of succeeding in rewriting the entire block chain, which would affect the whole Bitcoin network. Hence, the inability to control more than 50% computation power is one of the fundamental security assumptions of Bitcoin. We assume that no payer can break this assumption as well as other fundamental security properties of Bitcoin, such that a payer cannot break a payee's private key.

Ruffing et al. have proven the security of accountable assertion algorithm in [34], and proposed protocols to defend against non-equivocations in distributed networks, such as double-spending in Bitcoin. However, in their threat model, no collusion attacks are considered between a payer and a beneficiary. In our threat model, a payer may collude with a beneficiary. Although we assume that a beneficiary can collude with a payer, he/she does not risk to lose his/her incentive defined in the deposit, which is outlined in Section 4. Since Bitcoin users are anonymous, a payer may create an account and assign this account to act as a beneficiary.

In the case of making one deposit for multiple transactions, we allow the payer to collude with the beneficiary. Moreover, we also allow the payer to collude with both the beneficiary and any payee(s), or act as the beneficiary itself. In addition, the payer can create fake payees who are nothing but the payer him/herself for making some deposit transactions. More specifically, a payer can issue a deposit for multiple transactions and some of these transactions may transfer coins to his/her own addresses. In the case of making deposit without an explicit beneficiary, we allow the payer to play as a miner or collude with a miner.

4. Fair time-locked deposits

This section introduces and analyzes the design and usage of the fair time-locked deposits that thwart double-spending in Bitcoin transactions. We start with a simple case where a deposited transaction includes just one input and one output. Then we introduce the extension to a transaction with multiple inputs and outputs. In this section, we introduce the deposit with an explicit beneficiary; a deposit without an explicit beneficiary is discussed in Section 7.

4.1. Deposit for transactions with one input and one output

We first provide a solution to create a deposit for a transaction that contains one input and one output. Party A (i.e., payer) makes a payment to Party B (i.e., payee). Once A makes a transaction to B spending coins on B's services (or products), A creates a deposit for this transaction with beneficiary P. A should lock $d + \Delta$ coins in this deposit. If A doubly spends, d coins are used to compensate B's loss and Δ coins are transferred to P as his/her incentive to detect A's double-spending. The value of Δ is decided based on a negotiation between A and P.

To ensure that B's loss can be fully compensated once A doubly spends, the value of d can be set as the value of coins transferred to B in the deposited transaction. However, for a honest payer who has limited coins, such deposit value may obstruct his/her normal transactions. Hence, the value of d could also be decided based on a negotiation between A and B. A could lock relatively less coins in the deposit if B agrees. Considering that the negotiation between A and B requires additional user interactions, it may affect the usability.

To provide better usability and also decrease the amount of coins locked in the deposit for a honest payer, a minimum value of d could be specified in the implementations of our protocol. However, the set of this minimum value may cause some practical problems. If the minimum value of d is relatively low, e.g., 10% of the transaction value, a malicious payer who has sufficient coins can initiate a transaction transferring a large amount of coins (e.g., 10 coins) to the payee with a minimum deposit (e.g., 1 coin). In this case, the malicious payer may doubly spend without worrying the loss of the deposit. Meanwhile, the victim payee cannot get fair enough compensation from the deposit. If the minimum value of d is relatively high, e.g., 90% of the transaction value, the incentive for a malicious payer to double-spend would decrease to a low level. However, a honest payer who only has limited coins may feel burdensome for paying such deposits.

To impose a sufficient penalty on a malicious payer while decreasing the burdens on a honest payer, the minimum value of d could be decided based on a payer's reputation which is computed based on the payer's previous transaction behaviors. Generally, if a payer acts honestly in a transaction, his/her reputation increases accordingly. The reputation of a payer continuously increases along with the increasing number of his/her honest transactions. Otherwise, if a payer acts dishonestly in a transaction, e.g., double-spends, his/her reputation severely decreases. The minimum value of d varies according to a payer's reputation, and a payer with higher reputation could be entitled with a less minimum value.

Since Bitcoin network is anonymous, a malicious payer may initiate a double-spending transaction with a new Bitcoin account (which has no transaction records as a payer). Therefore, we suggest that, for all new Bitcoin accounts, the minimum value of d should be set as the value of the deposited transaction. Unfortunately, such reputation-based deposits cannot impose sufficient penalty on a malicious payer who has sophisticatedly behaved well in previous transactions with a maintained account and later doubly spent with this account. However, it increases the cost of double-spending attacks due to the transaction fees and overhead for maintaining a highly reputable account. It remains interesting to design the algorithms for calculating the reputation and generating the corresponding minimum deposit value based on previous works on reputation systems (e.g., [3,14,15,39]), which is however, out of the scope of this work.

This deposit is secured by A's secret key sk_A, and the corresponding public key is pk_A. Furthermore, the deposit is locked till some point T in the future. It means that even though A owns the secret key sk_A, he/she cannot redeem the deposit until time T. However, before time T, with the usage of accountable assertion scheme, it is possible for P to learn the secret key sk_A if A double-spends. Therefore, if A double-spends, P can recover sk_A, and then P and B can use their secret keys along with sk_A to redeem the deposit. Here, the role of the beneficiary P can also be performed by the payee B. Consequently, B should take all the responsibilities of the beneficiary.

The key pairs (pk_A, sk_A), (pk_B, sk_B), (pk_P, sk_P), and other Bitcoin key pairs involved in our protocols are all generated and managed in a standard way, e.g., by Bitcoin wallets, which is the same as the key generation and management in the current Bitcoin structure. A Bitcoin wallet is an app or a program that allows a user to send and receive Bitcoins. It also keeps tracking of the user's Bitcoin balance held in one or more bitcoin addresses and stores the user's transaction histroy. A Bitcoin wallet can be used to generate and manage private keys for the user. Some Bitcoin wallets also allow users to import their private keys generated from outside of the wallets. Since our protocols focus on the usage of private keys, the generation and management of private keys are out of scope.

4.1.1. Deposit setup

To create a deposit for the transaction in which A transfers coins to B, A should generate an accountable assertion. In particular, when creating the assertion, the *context ct* in this assertion is the transaction number of the previous output which the current input of the closing transaction is redeemed from. The *statement st* in the assertion is a random number generated by B. The assertion will be sent to B first for verification, and then sent to P who will detect A's double-spending. If P has received two different assertions generated under the same ct, P can confirm that A has double-spent the input. Then, P can recover sk_A using the two received assertions, and thus redeem the coins locked in the deposit. Figure 1 shows the message flow for setting up a deposit.

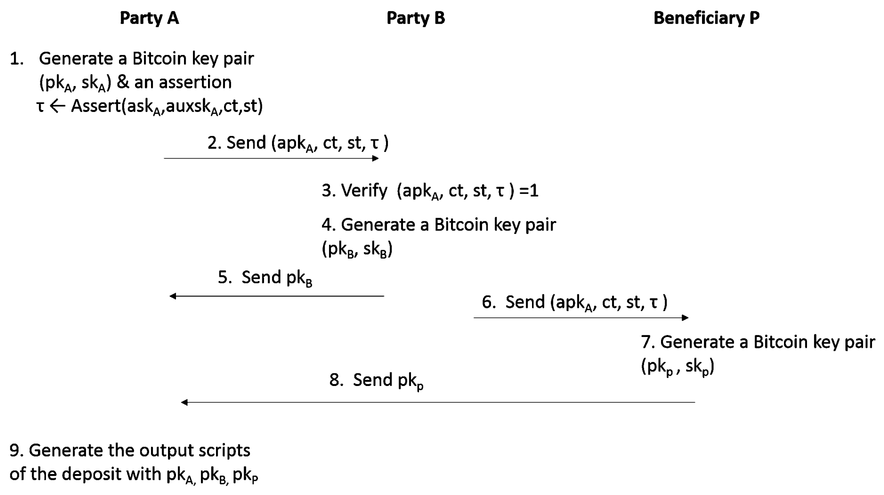

Party A
Party B
Beneficiary P
1. Generate a Bitcoin key pair (pk_A, sk_A) & an assertion $\tau \leftarrow$ Assert(ask_A,$auxsk_A$,ct,st)
2. Send (apk_A, ct, st, τ)
3. Verify (apk_A, ct, st, τ) =1
4. Generate a Bitcoin key pair (pk_B, sk_B)
5. Send pk_B
6. Send (apk_A, ct, st, τ)
7. Generate a Bitcoin key pair (pk_p, sk_p)
8. Send pk_p
9. Generate the output scripts of the deposit with pk_A, pk_B, pk_P

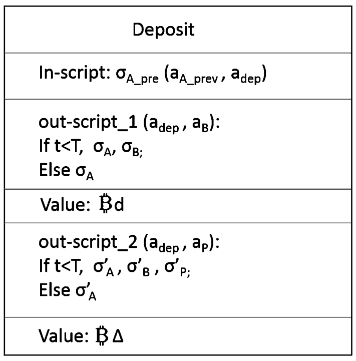
Fig. 1. Deposit setup for one transaction.

To generate an assertion, B generates a random number as st and sends it to A. After receiving st, A starts to generate the assertion. A first creates a Bitcoin key pair (pk_A, sk_A) which can be used to redeem the deposit after the expiry time T. Then, A sets up the accountable assertion scheme with the Bitcoin key pair (pk_A, sk_A). That is, A predefines the secret key of the chameleon hash tree $ask_A := sk_A$, creates the corresponding public key apk_A, and the auxiliary secret information $auxsk_A$ as specified in the key generation algorithm. Note that, $apk_A = (pk_A, z)$, where z is calculated based on chameleon hash values calculated in the key generation process, and $auxsk_A = k$, where k is generated by a pseudo-random function.

Next A uses the transaction number of the previous output as ct and the random number received from B as st to generate an assertion $\tau \leftarrow Assert(ask_A, auxsk_A, ct, st)$. However, not all transaction numbers can be mapped into the chameleon tree and thus be used to generate an assertion. If the transaction number cannot be mapped into the chameleon tree, A needs to re-generate a Bitcoin key pair and construct a new chameleon tree until ct can be mapped to a chameleon hash value in the tree. When the assertion is successfully constructed, A sends τ, apk_A, ct and st to B for verification.

After receiving τ, apk_A, ct and st, B first verifies ct and st. If ct and st are correct, B further verifies τ using apk_A, ct and st. If τ is valid, B generates a Bitcoin key pair (pk_B, sk_B) of his/her private address a_B and sends pk_B to A for defining the release condition of the deposit. Meanwhile, B sends the record (τ, apk_A, ct, st) to P for storing. After receiving the record (τ, apk_A, ct, st) from B, P generates a key pair (pk_P, sk_P) of his/her private address a_P and sends pk_P to A. Otherwise, if the verification of st fails, B can either ask A to regenerate an assertion or just cut off the transaction with A if he/she has enough reason to believe that A is malicious.

The deposit scripts are illustrated in Fig. 2. To ensure enough incentive for P as well as enough compensation to B, this deposit has two outputs when A double-spends: d coins are transferred to address a_B and Δ coins are transferred to address a_P. The release condition Π_B defines the requirements for transferring d coins to a_B. Π_B should ensure that such coins can only be redeemed with the authorization of B before the expiry time T. Hence, σ_A and σ_B are both required in Π_B. σ_A and σ_B are signatures on the transaction transferring d coins from a_{dep} to a_B with sk_A and sk_B, respectively. The release condition Π_P defines the requirements for transferring Δ coins to a_P. Π_P should ensure that such coins can only be transferred to a_P with the authorization of both P and B. Hence, σ'_A, σ'_B and σ'_P are all required in Π_P. Here, σ'_A, σ'_B and σ'_P are signatures on the transaction transferring Δ coins from a_{dep} to a_p with sk_A, sk_B and sk_P, respectively.



Deposit
In-script: σ_{A_pre} (a_{A_prev}, a_{dep})
out-script_1 (a_{dep}, a_B): If t<T, σ_A, σ_B; Else σ_A
Value: ₿d
out-script_2 (a_{dep}, a_P): If t<T, σ'_A, σ'_B, σ'_P; Else σ'_A
Value: ₿Δ

Fig. 2. Deposit script for one transaction.

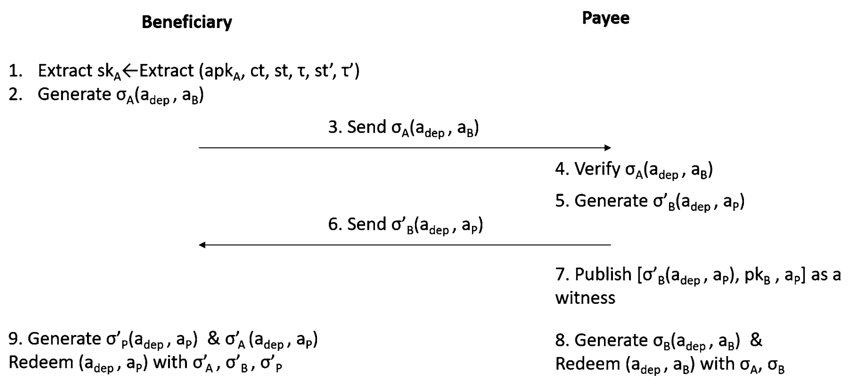

Beneficiary
Payee
1. Extract sk_A ← Extract (apk_A, ct, st, τ, st', τ')
2. Generate σ_A(a_dep , a_B)
3. Send σ_A(a_dep , a_B)
4. Verify σ_A(a_dep , a_B)
5. Generate σ'_B(a_dep , a_P)
6. Send σ'_B(a_dep , a_P)
7. Publish [σ'_B(a_dep , a_P), pk_B , a_P] as a witness
9. Generate σ'_P(a_dep , a_P) & σ'_A (a_dep , a_P) Redeem (a_dep , a_P) with σ'_A , σ'_B , σ'_P
8. Generate σ_B(a_dep , a_B) & Redeem (a_dep , a_B) with σ_A, σ_B

Fig. 3. Deposit usage for one transaction.

4.1.2. Deposit usage

If P detects the same ct in two different records (τ, apk_A, ct, st) and (τ', apk_A, ct, st'), it means that A has double-spent the input that is redeemed from the previous output ct. Then P uses τ and τ' to extract A's secret key sk_A and collaborates with B to transfer the coins locked in the deposit. Figure 3 shows the message flow of the deposit usage when A double-spends.

1. P uses the corresponding assertions τ and τ' to extract A's secret key sk_A.
2. P generates a signature σ_A on the transaction (a_{dep}, a_B) using sk_A.
3. P sends $\sigma_A(a_{\text{dep}}, a_B)$ to B.
4. B verifies $\sigma_A(a_{\text{dep}}, a_B)$ using the corresponding public key pk_A.
5. If $\sigma_A(a_{\text{dep}}, a_B)$ is a valid signature, B creates a signature $\sigma'_B(a_{\text{dep}}, a_P)$ on the transaction (a_{dep}, a_P) using the secret key sk_B.
6. B sends $\sigma'_B(a_{\text{dep}}, a_P)$ to P.
7. In case that B may not generate $\sigma'_B(a_{\text{dep}}, a_P)$ and send it to P, B needs to publish $(\sigma'_B(a_{\text{dep}}, a_P), pk_B, a_P)$ (either on his/her own bulletin board, on the blockchain or on some other "alt-chain") as a witness that enables everyone to check that his/her action was correctly performed.
8. B generates a signature on $\sigma_B(a_{\text{dep}}, a_B)$ on the transaction (a_{dep}, a_B) using his/her secret key sk_B. Therefore, B can transfer the d coins to his/her address a_B using $\sigma_P(a_{\text{dep}}, a_B)$ and $\sigma_B(a_{\text{dep}}, a_B)$.
9. After receiving $\sigma'_B(a_{\text{dep}}, a_P)$, P generates a signature $\sigma'_P(a_{\text{dep}}, a_P)$ on the transaction (a_{dep}, a_P) using his/her secret key sk_P and a signature $\sigma'_A(a_{\text{dep}}, a_P)$ using sk_A. Therefore, P can transfer the Δ coins to address a_P using $\sigma'_P(a_{\text{dep}}, a_P)$, $\sigma'_B(a_{\text{dep}}, a_P)$ and $\sigma'_A(a_{\text{dep}}, a_P)$ to satisfy the release condition.

4.1.3. Security analysis

Our design is resistent to collusion attacks by requiring B's private key to redeem the deposit when A double-spends. The collusion between A and P will not work due to the lack of B's secret key sk_B. In our design, P can transfer Δ coins to his/her private address only if B has signed the transaction (a_{dep}, a_P) with his/her secret key sk_B. Therefore, if P wants to gets the d coins in the deposit, he/she has to generate $\sigma_A(a_{\text{dep}}, a_B)$ and send it to B first.

If a malicious payer chooses one of his/her own accounts to act as the beneficiary, although the incentive locked in the deposit will be transferred to the payer's own address, he/she cannot transfer the compensation for the victim payee to his/her address due to the requirement of the victim payee's signature for redeeming the compensation. In this way, the payer still needs to transfer the same amount of coins as he/she has doubly spent to the victim payee. Hence, the payer achieves no benefit in such case through double-spending attacks.

Our design also ensures that B can get his/her compensation of d coins and P can get Δ coins as his/her benefits. This is because if P signs a transaction transferring less than d coins to B with sk_A, B will not sign the transaction (a_{dep}, a_P) with sk_B. Meanwhile, since P signs a transaction only transferring d coins to B, B cannot transfer all the coins in the deposit to his/her private address. Moreover, the requirement of publishing a witness also forces B to sign a transaction transferring Δ coins to P. If B refuses to sign the transaction (a_{dep}, a_P), P cannot get the incentive in this deposit. However, since B's misbehavior will be detected and broadcasted to the whole Bitcoin network, it is hard for B to find a beneficiary for his/her future transactions. The possible negative influence on his/her future transactions can force B to behave honestly (i.e., sign a transaction to transfer Δ coins to P with sk_B).

Although B has published $\sigma'_B(a_{\text{dep}}, a_P)$, we ensure that only P can transfer Δ coins that are locked in the deposit to his/her private address a_P by the additional requirement of $\sigma'_P(a_{\text{dep}}, a_P)$ in the release condition Π_P. If σ'_P are not required in the release condition Π_P, A may use his/her secret key to generate the signature $\sigma'_A(a_{\text{dep}}, a_P)$ to transfer the coins to his/her private address along with σ'_B which is published by B. On the other hand, if σ'_B is not required in the release condition Π_P, P may transfer all $d + \Delta$ coins to his/her private address if it recovers A's secret key sk_A or colludes with A, which could result in that B get no compensation.

4.2. Deposit for transactions with multiple inputs and outputs

For a transaction with multiple inputs and outputs, to ensure that any payee who suffers in double-spending can get compensation, deposits should be created for each payee, respectively. Note that the multiple inputs of the transaction may be made by several payers. In this case, the payers need to generate a Bitcoin key pair for the deposits by negotiation, so that the payers can be regarded as one payer who generates the Bitcoin key pair. Hence, in this section, we consider multiple payers as one payer.

4.2.1. Deposit setup

Party A (i.e., payer) makes a transaction with k payees, $B_1, B_2, \ldots, B_k$, in the deposited transaction, and Party P_i serves as the beneficiary for the deposit made to B_i. A should lock $d_{\text{mult}_i} + \Delta_{\text{mult}_i}$ coins in the deposit made to B_i with the expiry time T, where d_{mult_i} is the total value of coins that compensate to B_i if A double-spends, and Δ_{mult_i} is the incentive for P_i to detect the A's misbehavior. The value of Δ_{mult_i} is decided based on a negotiation between A and P_i, while the value of d_{mult_i} can be decided based on a negotiation between A and B_i. Since k payees are involved in the deposited transaction, it may be troublesome for the payer to negotiate with all payees. Hence, the value of d_{mult_i} could be simply set as the value of the coins transferred to B_i in the deposited transaction. To decrease the coins locked in the deposits for honest payers, a reputation system can be designed for calculating a minimum value of d_{mult_i}.

For a deposit made to B_i, the output script is similar to the deposit made for a transaction with single input and output. The release condition of d_{mult_i} coins requires signatures generated respectively under the payer's secret key sk_A and the payee's secret key sk_{Bi}. The release condition of Δ_{mult_i} requires signatures generated respectively under sk_A, sk_{Bi} and the respective beneficiary's secret key sk_{Pi}.

Since the double-spending of any input could result in the invalidation of all outputs, to make sure B_i can get d_{mult_i} coins from the deposit, sk_A should be able to be recovered from the double-spending of any input. Therefore, A should generate an assertion for each input and send all the assertions to P_i. All the assertions should be generated under the same chameleon tree where $ask_A := sk_A$.

To create deposits to all payees, A first generates n accountable assertions τ_j, $j \in \{1, 2, \ldots, n\}$ for n inputs respectively. To generate τ_j, ct_j is the transaction number of the previous output which the jth input is redeemed from. st_j is a random number decided by all the payees. For example, st_j can be generated by hashing the conjunction of all random numbers generated by each payee with MD5 function.

To make a deposit to B_i, A sends the n records: $(apk_A, ct_j, st_j, \tau_j)$, $j = (1, 2, \ldots, n)$, to B_i for verification. If all the records are verified as valid, B_i sends all these records to the respective beneficiary P_i.

4.2.2. *Deposit usage*

After receiving n records $(apk_A, ct_j, st_j, \tau_j)$, $j = (1, 2, \ldots, n)$, if P_i detects a different assertion under the same ct_j, he/she can recover sk_A and generate a signature σ_{Ai} on the transaction that transfers d_{mult_i} coins to B_i from the deposit using sk_A. If σ_{Ai} is verified by B_i, B_i generates a signature σ'_{Bi} on the transaction that transfers Δ_{mult_i} coins to P_i from the deposit using sk_{Bi} and sends σ'_{Bi} to P_i. Then B_i also needs to publish a witness. After that, B_i generates a signature σ_{Bi} on the transaction that transfers d_{mult_i} coins to B_i from the deposit using sk_{Bi}. Therefore, B_i can use σ_{Bi} and σ_{Ai} to satisfy the release condition of the d_{mult_i} coins. Meanwhile, after receiving σ'_{Bi} from B_i, P_i create signatures σ'_{Pi} and σ'_A on the transaction that transfers Δ_{mult_i} coins to P_i from the deposit using sk_{Pi} and sk_A, respectively. Then P_i can use σ'_{Bi}, σ'_{Pi} and σ'_{Ai} to satisfy the release condition of the Δ_{mult_i} coins locked in the deposit.

Although we introduce the solution to make a deposit to the payee B_i with the beneficiary P_i, the payer could create the deposits for different payees with the same beneficiary. This is because, the records sent to each P_i are the same and the sk_A used in all deposits is the same. Hence, if a beneficiary recovers the sk_A, he/she can generate σ_{Ai} on the transaction $(a_{\text{dep}}, a_{B_i})$ using sk_A for all B_i, $i = (1, 2, \ldots, k)$. Furthermore, instead of respectively making k deposits, the payer can create only one deposit to all k payees. In this case, the deposit transaction should contain $k + 1$ outputs. If the payer double-spends, k outputs transfer the compensations to k payees, respectively, and the other output transfers the incentive to the beneficiary. The release conditions of the compensations locked in this deposit require the signatures of the payer and the respective payee, which is the same as in the deposit separately made to the respective payee.

4.2.3. *Security analysis*

The deposits for transactions with multiple inputs and multiple outputs provide the same security as the deposits for transactions with single input and output. When A double-spends, our design requires the collaboration among the beneficiaries and the victim payees to withdraw the deposits. For $i \in \{1, 2, \ldots, k\}$, the collusion between A and P_i will not work since B_i's secret key sk_{B_i} is required in the release condition. Our design also ensures that B_i can get his/her compensation of d_{mult_i} coins, since B_i will not sign the transaction (a_{dep}, a_P) with sk_B if P_i signs a transaction transferring less than d_{mult_i} coins to B_i with sk_A. Meanwhile, our design also grantees the incentive for P_i. Since P_i signs a transaction transferring only d_{mult_i} coins to B_i, B_i cannot transfer all the coins in the deposit to his/her private address. Moreover, B_i is required to sign the transaction transferring Δ_{mult_i} coins to P_i's private address and publish a corresponding witness which forces B_i to behave honestly.

Our design can prevent collusion attacks between different beneficiaries. If A sends the record $(apk_A, ct_j, st_{j0}, \tau_{j0})$ to B_i, $i \in \{1, 2, \ldots, k\}$, and the record $(apk_A, ct_j, st_{j1}, \tau_{j1})$ to B_{i+1}, these two different records will be sent to P_i and P_{i+1}, respectively. Then, P_i can collude with P_{i+1} to recover sk_A, even if A has not double-spent the input with the transaction number ct_j. In our design, all payees receive the same records $(apk_A, ct_j, st_j, \tau_j)$, $j = (1, 2, \ldots, n)$, from the payer. Hence, for the same ct_j, all beneficiaries receive the same st_j from the respective payee. The usage of the same st_j can prevent P_i recovering sk_A through colluding with other beneficiary.

5. Implementation and evaluation

In this section, we describe how our fair deposits for one transaction can be implemented in Bitcoin network. We use a deposit created for a transaction with single input and output as an example. We also evaluate the validation of our scripts and the effectiveness of the accountable assertion algorithm used in our design. To make deposits for a transaction with multiple inputs and outputs, the scripts for each deposit are basically the same as a deposit made for a transaction with single input and output. Hence, the scripts validation provided in this section can also demonstrate the validity of the deposit transactions created for the transactions with multiple inputs and outputs.

5.1. Implementation

We implement the accountable assertion algorithm based on the codes published online [35] by Ruffing et al. In the deposit setup phase, the accountable assertion is generated and verified using the assertion algorithm $Assert(ask, auxsk, ct, st)$ and the verification algorithm $Verify(apk, ct, st, \tau)$ respectively. In our implementation, most of the parameters used by Ruffing et al. [34] are left intact. Particularly, we use HMAC-SHA256 to instantiate the pseudorandom function F, SHA256 to instantiate the collision-resistant hash function H, and HMAC-SHA256 with fixed keys to instantiate the random oracles L and S. The height of the tree $l = 64$ and the arity $n = 2$ are the same as that were used by Ruffing et al. [34].

However, to generate an assertion, we customize the *context* and the *statement* used in the accountable assertion algorithm. In our implementation, the *context ct* is the Bitcoin address of the input of the transaction that has been deposited. This Bitcoin address can identify the respective input in Bitcoin network. As we introduced in Section 4, in each Bitcoin transaction, instead of Bitcoin address of the input, the transaction number of the previous output which the input of the deposited transaction is redeemed from can also be used to identify this input. The transaction number is constructed by the value of *prevTx* (which is the hash identifying the previous transaction) and the value of *index* (which is the index of the respective output in that transaction). Considering that the *prevTX* and *index* can all be found out by tracing the Bitcoin address, we just use the input Bitcoin address of the deposited transaction as the *ct* in our implementations.

There are two ways to represent the Bitcoin address – hex format and Base56Check encoding. In our implementation, we use a 20-byte hexadecimal Bitcoin address as *ct*. The bitcoin address can also be represented in Base56Check encoding in other implementations with some minor modifications on our codes. To use the Base56Check encoding, the *ct* needs to support a variable length since Base56Check encoding ranges from 25–34 characters. In our implementaion, the statement *st* is 32-byte random number r_{st} generated by the payee and sent to the payer for generating the assertion.

In the deposit usage phase, the beneficiary extracts the payer's secret key sk_A using the key extraction algorithm $Extract(apk, ct, st, st', \tau, \tau')$. After extracting sk_A, the beneficiary calculates the address a_B, a_P and generates a signature on the transaction (a_{dep}, a_B) using sk_A. In our implementation a Bitcoin address a is formed from the public key of an ECDSA key pair in the formal way by hashing the public key with SHA-256 first and RIPEMD-160 subsequently, prepending a version number, and appending a checksum for error detection. Thus, the address a_B where the coins in the $output_B$ will be transferred to is derived from the payee's public key pk_B, and the address a_P where the coins in the $output_P$ will be transferred to is derived from the beneficiary's public key pk_P. After generating the signature on the transaction (a_{dep}, a_B) using sk_A, the beneficiary sends a_B, a_P and the signature to the payee. If the payee verifies a_B, a_P and the signature on the transaction (a_{dep}, a_B) as valid, he/she generates a signature on the transaction (a_{dep}, a_P) using sk_B and sends it back to the beneficiary.

The Bitcoin script language supports a *CHECKSIG* operation that reads a public key and a signature from the stack and then verifies the signature against the public key on a message that is derived in a special way from the current transaction. This (and its multi-sig version) is the only operation that performs signature verification. In our deposit transaction, the outputs require the verification of signatures against specific public keys. In the output scripts, we make use of a *IF–ELSE* structure, so that if the payer double-spends, the victim payee and the beneficiary access to the deposit, locking out the payer. Otherwise, if the payer acts honestly, he/she can get the deposit back. We design the pubkey script (*scriptPubKey*) for the deposit transactions as follows, where *Output_B* denotes the output redeemed and controlled by the payee and *Output_P* denotes the output redeemed and controlled by the beneficiary:

- Output_B:
  IF
  DUP HASH160 <PK_B hash> EQUALVERIFY
  CHECKSIGVERIFY
  ELSE
  <Lock Time> CHECK_LOCKTIMEVERIFY DROP
  ENDIF
  DUP HASH160 <PK_A hash> EQUALVERIFY
  CHECKSIG
- Output_P:
  HASH160 <redeemScript hash> EQUAL
  where the *redeemScript* is:
  IF 2 PK_P, PK_B 2 CHECKMULTISIGVERIFY
  ELSE
  <Lock Time> CHECKLOCKTIMEVERIFY DROP
  ENDIF
  <PK_A> CHECKSIG

Correspondingly, once the beneficiary has detected the payer's double-spending, the required signature script (*scriptSig*) would be as follows:

- For Output_B
  <σ_A> <PK_A> <σ_B> <PK_B> OP_1
- For Output_P
  <σ_A> OP_0 <σ_P> <σ_B> OP_1 <redeemScript>

where the payer's signature is obtained from the beneficiary's extraction of the payer's private key.

If the payer has not double-spent, he/she can gain full access to his/her deposit using the following *scriptSig*:

- For Output_B
  <σ_A> <PK_A> OP_0

- For Output_P
  <σ_A> OP_0 <redeemScript>.

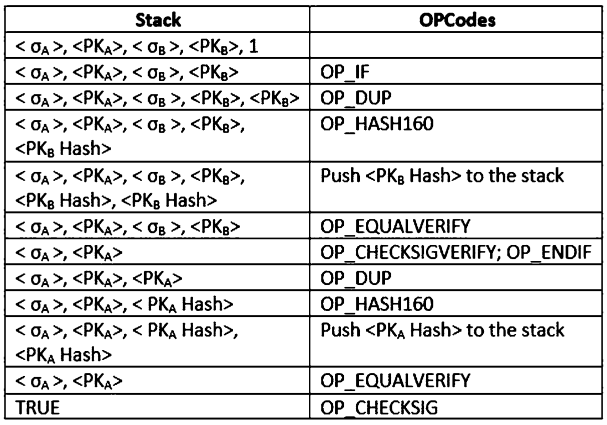
We write the scripts using Bitcoin script language. However, since the *IF_ELSE* structure used in our scripts is not considered as standard by Bitcoin Core's IsStandard() or IsStandardTx(), the transactions that use our scripts would not be considered as standard transactions. Although Bitcoin nodes running in the default settings may not accept, broadcast, or mine non-standard transactions, the transactions that use our scripts can still be supported in the current Bitcoin network without any modifications on Bitcoin structures. This is because Bitcoin nodes that accept non-standard transactions have already existed in the Bitcoin network. The transactions that use our scripts can be broadcasted to Bitcoin nodes which accept non-standard transactions and finally be confirmed by the Bitcoin network. In the future, if the Bitcoin network is updated with a standardization of the *IF-ELSE* structure due to its increasing usage, the transactions that use our scripts will also be supported by Bitcoin nodes running in the default settings.

5.2. Validation and evaluation

5.2.1. Scripts validation

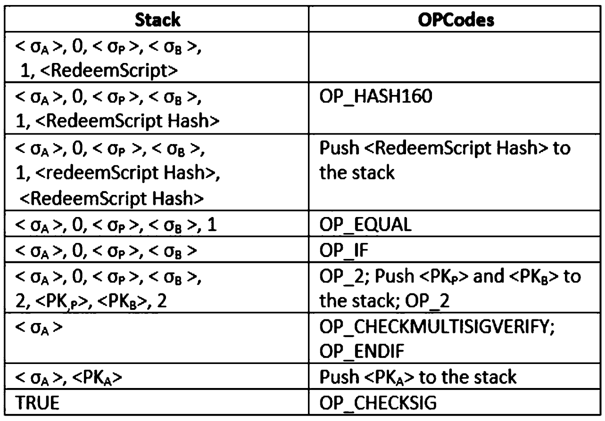
We show the validity of our scripts by observing the evaluation of the respective *scriptSig* and *scriptPubKey* in the stack. According to the Developer Guide released by Bitcoin Project [7], if false is not at the top of the stack after the *scriptPubKey* has been evaluated, the transaction is valid (provided there are no other problems with it). To test whether the transactions that transfer the coins locked in the deposit are valid, *scriptSig* and *scriptPubKey* operations are executed one item at a time in the evaluation stack, starting with the payer's *scriptSig* and continuing to the end of the *scriptPubKey* provided by whom redeems the deposit.

Figure 4 and Fig. 5 show the evaluation stack during the validation of *scriptPubKey* provided by the payee and the beneficiary, respectively, if the payer double-spends. We can see that at the end of the computations, the return value at the top of the stack is TRUE, affirming that our scripts can be performed successfully and the transactions using these scripts are valid. Detailed description of these validation processes are given in Appendix. We also evaluate the stack during the scripts validation in the event that the payer acts honestly and the corresponding evaluation stacks over time are given in the



Stack	OPCodes
< σ_A >, <PK_A>, < σ_B >, <PK_B>, 1	
< σ_A >, <PK_A>, < σ_B >, <PK_B>	OP_IF
< σ_A >, <PK_A>, < σ_B >, <PK_B>, <PK_B>	OP_DUP
< σ_A >, <PK_A>, < σ_B >, <PK_B>, <PK_B Hash>	OP_HASH160
< σ_A >, <PK_A>, < σ_B >, <PK_B>, <PK_B Hash>, <PK_B Hash>	Push <PK_B Hash> to the stack
< σ_A >, <PK_A>, < σ_B >, <PK_B>	OP_EQUALVERIFY
< σ_A >, <PK_A>	OP_CHECKSIGVERIFY; OP_ENDIF
< σ_A >, <PK_A>, <PK_A>	OP_DUP
< σ_A >, <PK_A>, < PK_A Hash>	OP_HASH160
< σ_A >, <PK_A>, < PK_A Hash>, <PK_A Hash>	Push <PK_A Hash> to the stack
< σ_A >, <PK_A>	OP_EQUALVERIFY
TRUE	OP_CHECKSIG

Fig. 4. Evaluation stack during the Output_B script validation when the payer double-spends.



Stack	OPCodes
< σ_A >, 0, < σ_P >, < σ_B >, 1, <RedeemScript>	
< σ_A >, 0, < σ_P >, < σ_B >, 1, <RedeemScript Hash>	OP_HASH160
< σ_A >, 0, < σ_P >, < σ_B >, 1, <redeemScript Hash>, <RedeemScript Hash>	Push <RedeemScript Hash> to the stack
< σ_A >, 0, < σ_P >, < σ_B >, 1	OP_EQUAL
< σ_A >, 0, < σ_P >, < σ_B >	OP_IF
< σ_A >, 0, < σ_P >, < σ_B >, 2, <PK_P>, <PK_B>, 2	OP_2; Push <PK_P> and <PK_B> to the stack; OP_2
< σ_A >	OP_CHECKMULTISIGVERIFY; OP_ENDIF
< σ_A >, <PK_A>	Push <PK_A> to the stack
TRUE	OP_CHECKSIG

Fig. 5. Evaluation stack during the Output_P script validation when the payer double-spends.

Table 1

Accountable assertion performance evaluation

Operation	AVG Time (ms)	
	[34]	Our Fair Deposits
Assertion generation	9	18
Assertion verification	4	8
Key extraction	N/A	36

Appendix. The results demonstrate that the transactions that redeem the deposit using our scripts are also valid when the payer acts honestly.

5.2.2. *Performance evaluation*

Comparing with the current Bitcoin transaction mechanism, our solution causes additional overhead by performing the accountable assertion algorithm. The overhead is mainly caused by assertion generation, assertion verification and key extraction. Hence, we evaluate the overhead caused by these operations and present our experiments in details. Moreover, we also present a comparison on the overhead between our solution and the first accountable assertion algorithm [34].

For each round of the experiments, we first generate a Bitcoin key pair as the payer's Bitcoin key pair and a Bitcoin address as the *context*. The Bitcoin key pairs used in each round are all generated using OpenSSL 1.0.1h. The corresponding Bitcoin addresses are calculated with the *RIPEMD160()* and *SHA256()* commands in the OpenSSL C++ library. We then generate and verify an accountable assertion using the generated Bitcoin key pair and the *context*, and record the required time for assertion generation and verification respectively. After that, we generate another assertion using the same *context* and different *statement*, and then extract the Bitcoin private key from these two conflicted assertions. In addition, we also record the required time for the key extraction. We run the experiments for 50 rounds and the average time for assertion generation, assertion verification and key extraction are recoded in Table 1.

The experiments are performed on a 2.4 GHz (Intel Core i5-4258U) machine with a DDR3-1600 MHz RAM. Ruffing et al. [34] also evaluate the overhead of their design caused by assertion generation and assertion verification. Comparing with [34], our design needs more time to generate and verify an assertion. This is because in our design the size of the *context* and the *statement* grow by a significant amount. We use a 20-byte hexadecimal Bitcoin address as the *context* and a 32-byte random number as the *statement*, while the *context* is 8-byte and the *statement* is 3-byte in [34]. However, the computational overhead of our design is still millisecond-level, hence it is still manageable and acceptable.

In other computing environments, the overhead of our solution and the overhead of [34] would be different from our evaluation due to the different computation power of computing devices. Compared to the common computing devices for Bitcoin transactions, the device we evaluated, i.e., a 2.4 GHz (Intel Core i5-4258U) machine with a DDR3-1600 MHz RAM, is not at the high end. If our solution is performed on a device with higher computation power, the overheard would further decrease. In addition, our evaluation demonstrates that the overhead of our solution is in the same order of magnitude as that of [34].

In terms of communication, comparing to the current Bitcoin transaction mechanism, the transfer of assertions in our solution causes additional communication overhead. The size of assertion in our solution is the same as the solution in [34]. Note that a chameleon hash value is a point on the secp256k1 curve and thus requires less than 33 bytes in the compressed form. A random input of the chameleon hash function is a 32-byte integer in the underlying field of the curve. The assertion in our solution is a sequence of 64 chameleon hash values and chameleon hash random inputs. Therefore, it takes $64 \times (33 \text{ bytes} + 32 \text{ bytes}) = 4160$ bytes.

6. Extension to a deposit for multiple transactions

In this section, we propose a solution for creating a deposit for multiple transactions to reduce the transaction fees. In our solution, a payer who has decided to perform n transactions could make one deposit with the expiry time T for the n transactions. After the deposit is confirmed by the Bitcoin network, the n transactions will be initiated and broadcasted to the Bitcoin network for confirmation. If the beneficiary detects a double-spending transaction among the n transactions before the expiry time T, the victim payee of the double-spending transaction will get compensation from the deposit.

6.1. Deposit setup

Party A (i.e., payer) makes n transactions with n payees, $B_1, B_2, \ldots, B_n$, and Party P serves as the beneficiary who is responsible for detecting A's double-spending. The value of coins locked in the deposit is $d' = \sum_{i=1}^{n} d_{Bi} + \Delta \times n$, where d_{Bi} is the value of coins used to compensate B_i's loss if A double-spends the ith transaction, and Δ is the incentive for the beneficiary to identify a double-spending transaction. The value of Δ is decided based on a negotiation between the A and P. The value of d_{Bi} should ensure that each payee who has suffered in a double-spending transaction can get sufficient compensation of his/her loss from the deposit. Hence, the value of d_{Bi} can be set as the value of the ith transaction. The value of d_{Bi} can also be decided based on a negotiation between A and B_i, though it may be troublesome for A to negotiate with all n payees. To decrease the coins locked in the deposit for a honest payer, a reputation system can be designed for calculating a minimum value for d_{Bi}.

Compared to a deposit for one transaction, the setup of a deposit for multiple specific transactions requires a pre-negotiation process between the payer and each payee as shown in Fig. 6. For simplicity, we use the case that each deposited transaction only has one payee as an example.

To create a deposit for n transactions, A first creates a Bitcoin key pair (pk_A, sk_A). Also, A sets up the accountable assertion scheme with the Bitcoin key pair (pk_A, sk_A). That is, A predefines the secret key $ask_A := sk_A$ of the accountable assertion scheme and creates the corresponding public key

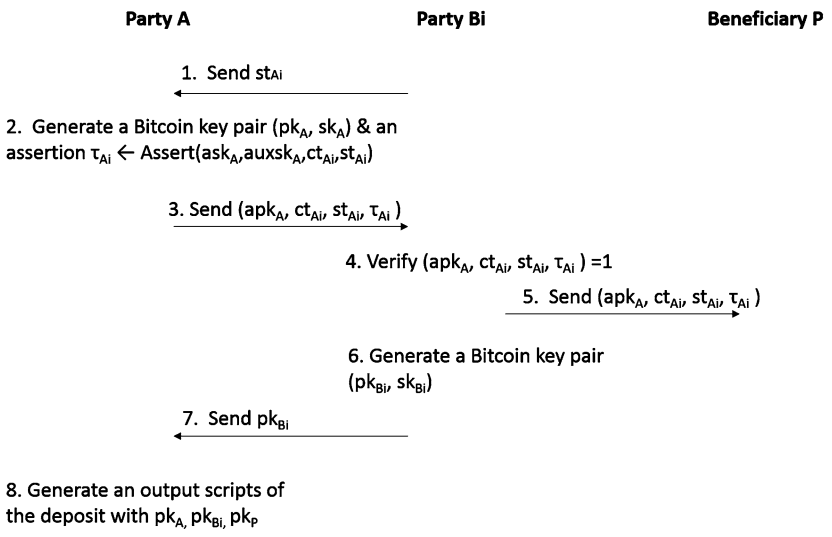
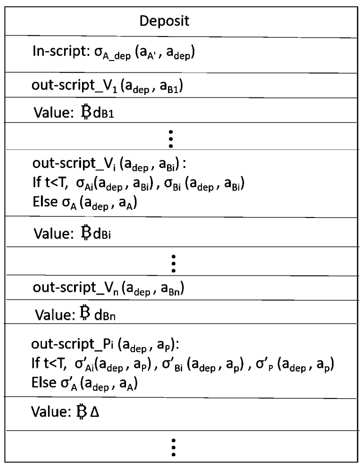

Party A
Party Bi
Beneficiary P
1. Send st_{Ai}
2. Generate a Bitcoin key pair (pk_A, sk_A) & an assertion $\tau_{Ai} \leftarrow$ Assert(ask_A,$auxsk_A$,ct_{Ai},st_{Ai})
3. Send (apk_A, ct_{Ai}, st_{Ai}, τ_{Ai})
4. Verify (apk_A, ct_{Ai}, st_{Ai}, τ_{Ai}) =1
5. Send (apk_A, ct_{Ai}, st_{Ai}, τ_{Ai})
6. Generate a Bitcoin key pair (pk_{Bi}, sk_{Bi})
7. Send pk_{Bi}
8. Generate an output scripts of the deposit with pk_A, pk_{Bi}, pk_P

Fig. 6. The message flow of deposit setup for multiple transactions.



Deposit
In-script: σ_{A_dep} ($a_{A'}$, a_{dep})
out-script_V$_1$ (a_{dep}, a_{B1})
Value: ₿d_{B1}
⋮
out-script_V$_i$ (a_{dep}, a_{Bi}) :   If t<T, σ_{Ai}(a_{dep}, a_{Bi}), σ_{Bi} (a_{dep}, a_{Bi})   Else σ_A (a_{dep}, a_A)
Value: ₿d_{Bi}
⋮
out-script_V$_n$ (a_{dep}, a_{Bn})
Value: ₿ d_{Bn}
out-script_Pi (a_{dep}, a_P):   If t<T, σ'_{Ai}(a_{dep}, a_P), σ'_{Bi} (a_{dep}, a_p), σ'_P (a_{dep}, a_p)   Else σ'_A (a_{dep}, a_A)
Value: ₿ Δ
⋮

Fig. 7. Deposit script for multiple transactions.

apk_A, and the auxiliary secret information $auxsk_A$ as specified in the key generation algorithm. Note that $apk_A = (pk_A, z)$. For $i \in \{1, 2, \dots, n\}$, to generate the assertion τ_{Ai}, B_i generates a random number and sends it to A as st_{Ai}. Then, A uses the transaction number of the output which the input of the transaction with B_i is redeemed from as ct_{Ai} to generate τ_{Ai} along with st_{Ai}.

Then A sends the record $(apk_A, ct_{Ai}, st_{Ai}, \tau_{Ai})$ to B_i. After verifying that $Verify(apk_A, ct_{Ai}, st_{Ai}, \tau_{Ai}) = 1$, B_i sends the record to the beneficiary P for storing. Meanwhile, B_i generates a Bitcoin key pair (pk_{Bi}, sk_{Bi}) and sends pk_{Bi} to A. After receiving $pk_{B1}, pk_{B2}, \dots,$ and pk_{Bn} from all n payees, A asks P for his/her public key to generate the output scripts of the deposit. Then P generates a bitcoin keypair (pk_P, sk_P) and sends pk_P to A. After that, A defines the output script of the deposit as shown in Fig. 7 and broadcasts the deposit transaction in the Bitcoin network for confirmation.

The release conditions of *Output_B*$_i$ (i.e., the output to compensate B_i's loss) and *Output_P*$_i$ (i.e., the output to incentivize the beneficiary to detect the double-spending of the transaction made to B_i) are similar with *Output_B* and *Output_P* introduced in Section 5.1. The design of the scripts for *Output_B*$_i$ and *Output_P*$_i$ is referred to the scripts for *Output_B* and *Output_P*, respectively given in Section 5.1, and the required public keys in the output scripts should be changed to corresponding public keys required in *Output_B*$_i$ and *Output_P*$_i$, respectively.

This deposit has $n+n$ outputs. Among these outputs, n outputs ensure that d_{Bi} coins can be transferred to B_i's private address a_{Bi}, where $i = 1, 2, \ldots, n$, if A double-spends the transaction with B_i. If A acts honestly, d_{Bi} coins can be transferred to A's private address a_A after the expiry time T. The other n outputs ensure that, if P detects that A double-spends the transaction with B_i, $i \in \{1, 2, \ldots, n\}$, Δ coins can be transferred to P's private address a_P. Otherwise, if A has not double-spent the transaction with B_i, Δ coins can be transferred to A's private address a_A.

In this deposit, we design n outputs each of which transfers Δ coins to P's private address. It can motivate P to detect as many double-spending transactions as possible, since each output requires a signature generated using a victim payee's private key. Only if P has detected that A double-spent the transaction with the victim payee, the victim payee will sign the transaction transferring Δ coins to P's private address using his/her secret key. Thus, P can transfer Δ coins to his/her private address.

6.2. *Deposit usage*

When P detects that A double-spends the transaction with B_i, $i \in \{1, 2, \ldots, n\}$ before the expiry time T, he/she can work with party B_i to transfer d_{Bi} coins from the deposit to B_i's private address and Δ coins to P's private address. Figure 8 shows the message flow of the deposit usage after P detects that A has double-spent the transaction made with B_i. The usage of the deposit can be described as follows:

1. If P detects different st_{Ai} in two records with the same ct_{Ai}: $(apk_A, ct_{Ai}, st_{Ai_0}, \tau_{Ai_0})$ and $(apk_A, ct_{Ai}, st_{Ai_1}, \tau_{Ai_1})$, P uses the assertions τ_{Ai_0} and τ_{Ai_1} to extract A's secret key sk_A.
2. P creates a signature σ_{Ai} on the transaction (a_{dep}, a_{Bi}) using sk_A.
3. Then P sends σ_{Ai} to B_i.
4. B_i verifies σ_{Ai} using the corresponding public key pk_A.
5. If σ_{Ai} is valid, B_i creates a signature $\sigma'_{Bi}(a_{\text{dep}}, a_P)$ on the transaction (a_{dep}, a_P) that transfers Δ coins to P's private address using the secret key sk_{Bi}.
6. B_i sends $\sigma'_{Bi}(a_{\text{dep}}, a_P)$ to P.

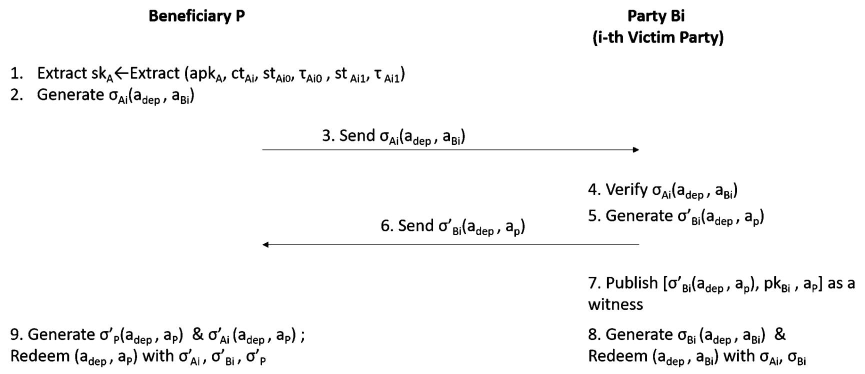

Beneficiary P
Party Bi
(i-th Victim Party)
1. Extract sk_A ← Extract (apk_A, ct_{Ai}, st_{Ai0}, τ_{Ai0} , st_{Ai1}, τ_{Ai1})
2. Generate $\sigma_{Ai}(a_{dep}, a_{Bi})$
3. Send $\sigma_{Ai}(a_{dep}, a_{Bi})$
4. Verify $\sigma_{Ai}(a_{dep}, a_{Bi})$
5. Generate $\sigma'_{Bi}(a_{dep}, a_p)$
6. Send $\sigma'_{Bi}(a_{dep}, a_p)$
7. Publish [$\sigma'_{Bi}(a_{dep}, a_p)$, pk_{Bi} , a_p] as a witness
8. Generate $\sigma_{Bi}(a_{dep}, a_{Bi})$ & Redeem (a_{dep}, a_{Bi}) with σ_{Ai}, σ_{Bi}
9. Generate $\sigma'_P(a_{dep}, a_P)$ & $\sigma'_{Ai}(a_{dep}, a_P)$; Redeem (a_{dep}, a_P) with σ'_{Ai} , σ'_{Bi} , σ'_P

Fig. 8. Deposit usage for multiple transactions.

7. In case that B_i may not generate $\sigma'_{Bi}(a_{\text{dep}}, a_P)$ and send it to P, B_i needs to publish (σ'_{Bi}, pk_{Bi}) (either on his/her own bulletin board, on the blockchain or on some other "alt-chain") as a witness that enables everyone to check that his/her action was correctly performed.
8. Then B_i generates a signature $\sigma_{Bi}(a_{\text{dep}}, a_{Bi})$ on transaction (a_{dep}, a_{Bi}). Therefore, B_i can transfer the d_{Bi} coins to his/her private address using σ_{Bi} and σ_{Ai} to satisfy the release condition.
9. After receiving $\sigma'_{Bi}(a_{\text{dep}}, a_P)$, P generates a signature $\sigma'_P(a_{\text{dep}}, a_p)$ on the transaction (a_{dep}, a_p) using his/her secret key sk_p and a signature $\sigma'_{Ai}(a_{\text{dep}}, a_P)$ using sk_A. Therefore, P can transfer the Δ coins to his/her personal address.

6.3. Security analysis

Our design is resistent to the collusion attacks between the payer and the beneficiary by requiring B_i's private key to transfer d_{Bi} coins to B_i's private address and transfer Δ coins to P's private address when A double-spends the transaction made with B_i. The collusion between A and P will not work due to the lack of B_i's secret key sk_{Bi}. In the cases where the payer chooses one of his/her own accounts to act as the beneficiary, although the incentive locked in the deposit could be transferred to the payer's own address if he/she double-spends some of the deposited transactions, he/she cannot transfer the compensation to the victim payees to his/her address due to the requirement of the victim payees' signatures to redeem the corresponding compensation.

Our design is also resistent to the collusion attacks between payer A, beneficiary P and a malicious payee $B_q, q \in \{1, 2, \dots, n\}$. In such collusion attacks, only $d_{Bq} + \Delta$ coins in the deposit can be controlled by the attackers, i.e., payer A, beneficiary P and a malicious payee $B_q, q \in \{1, 2, \dots, n\}$. If the payer A has double-spent the transaction made with a payee and wants to transfer the corresponding coins locked in the deposit to his/her private address, the real victim's signature is still required. Hence, even if the payer has created some "fake" payees for the deposit, he/she still cannot grab the coins deposited for the real transactions and the real victim's compensation is still guaranteed.

7. Extension to deposits without an explicit beneficiary

In this section, we provide an extension to deposits of which the beneficiary is a randomly selected miner rather than an explicit beneficiary. In this case, each miner can perform as a beneficiary of the deposit and detect A's misbehavior of double-spending. For simplify, we only design the protocol of making a deposit without an explicit beneficiary for a transaction that only contains one input and output. Based on some further extensions of the protocol introduced in this section, a deposit without an explicit beneficiary can also be made for a transaction with multiple inputs and outputs, and even for multiple transactions.

7.1. Deposit design

Let Party A denote the payer, Party B denote the payee, and d denote the value of the transaction between A and B. To enable each miner monitor A's behavior and recover A's secret key sk_A if A double-spends, B needs to publish the received record (apk, ct, st, τ) to all the miners who have the incentive to store it. The release condition of B's compensation locked in the deposit remains the same as a deposit with an explicit beneficiary. Thus, if A double-spends, the miner who reveals A's secret key sk_A signs the transaction that transfers d coins to B's private address with sk_A and sends the signature to B. After receiving a signature from a miner, B first verifies the signature. If the signature is verified as valid, B can transfer the compensation to his/her private address with signatures generated with sk_B and sk_A. The expiry time of a deposit without an explicit beneficiary is still $T = T_{\text{net}} + T_{\text{conf}} + T_{\text{def}}$.

Unlike in a deposit with an explicit beneficiary, in a deposit without an explicit beneficiary, the release condition of the incentive for the beneficiary should not require a signature generated with the beneficiary's secret key. This is because the payer cannot define the output script with the beneficiary's public key which is indeterminate at the time of creating the deposit. Therefore, the release condition of the incentive requires signatures on the transaction that transfers Δ coins to the beneficiary's (i.e., the random miner's) private address respectively using the payer's secret key and the payee's secret key. Hence, after receiving a signature on the transaction (a_{dep}, a_B) generated with sk_A from a miner, the payee B should generate a signature on the transaction $(a_{\text{dep}}, a_{\text{miner}})$ using his/her secret key sk_B and sends this signature to the miner. After that, B also needs to publish a witness as an evidence of his/her honest behavior. Then the miner generates a signature on the transaction $(a_{\text{dep}}, a_{\text{miner}})$ using A's secret key sk_A to redeem the incentive locked in the deposit together with the signature received from B. The design of the scripts is referred to the output scripts given in Section 5.1. Particularly, the required public keys in the scripts of the output that ensures the incentive to the beneficiary should be changed to pk_A and pk_B.

7.2. Security analysis

Since the accountable assertion is published to all miners, any miner who wants to get the incentive can recover A's secret key sk_A if A double-spends. However, only one miner can finally get the incentive. There are two different ways to decide which miner can finally be the beneficiary. In the first way, only the first miner who recovers sk_A and sends the payee a signature generated from sk_A can get the incentive. The payee only generates a signature on the transaction that transfers the incentive to the miner who is the first one to send the payee a signature on the transaction (a_{dep}, a_B) generated with sk_A. In the second way, the miner who includes a transaction that transfers the incentive to his/her own account in a block and adds the next block to the consensus blockchain will claim the incentive. Hence, for every miner who sends a signature on the transaction (a_{dep}, a_B) using sk_A to the payee, the payee generates and sends the miner a signature on the transaction that transfers the incentive to the miner.

Compared to the first way, both the payee and miners consume more computation powers in the second way. Therefore, the computation overhead is relatively lower in the first way. However, the first way is subject to an attack where a malicious payer plays as a miner or colludes with a miner. It is easy for the payer who plays the role of a miner or colludes with a miner to be the first one to send a signature on (a_{dep}, a_B) with sk_A if the network latency is not considered, since such miner does not need to recover sk_A. Even if the payer plays as a miner or collude with a miner, he/she can only transfer the incentive for the beneficiary to his/her own address and the victim payee can still get fair enough compensation due to the requirement of the victim payee's signature to redeem the coins that were deposited for the payee.

In addition, the first way may be unfair to other miners if the payee selects a particular miner as the beneficiary and colludes with it. In particular, the payee could ignore the signatures on (a_{dep}, a_B) with sk_A sent by other miners and keep waiting for the signature sent by the particular miner. The collusion between the payee and a particular miner would not cause an honest payer's loss, since the particular miner can only recover the payer's secret key if the payer double-spends. However, such collusion is unfair to other miners who have dedicated to monitor the payer's behavior and should have the fair chance for the incentive. To prevent such unfairness, each miner who sends a signature on (a_{dep}, a_B) with sk_A to the payee should publish a witness as the evidence of sending the signature. It enables everyone to check the publication time of a witness, and thus be aware of the first miner who sends the signature to the payee.

In the second way, although the malicious payer A can participate in the mining process, only if A controls a majority of the computation power in the network, he/she can ensure that the transaction that transfers the incentive to his/her address can be confirmed by the network. Hence, the computation power required for successfully transferring the incentive to A's address in the second way is much more than the first way. However, as we discussed in our threat model (Section 3), A cannot control a majority of the computation power in the network, which is one of the underlying assumptions for security of the Bitcoin network.

In both two ways, we do not need to prevent A from pre-mining the transaction that transfers all coins in the deposit to his/her address as discussed in [34]. This is because signatures generated using sk_B are required for transferring the coins in the deposit to both the miner and the payee if the payer double-spends.

8. Extension to non-equivocation contracts

Besides preventing double-spending in Bitcoin, our deposit scheme can also be used for creating non-equivocation contracts in various distributed systems that employ public append-logs to protect data integrity, e.g., in cloud storage and social networks. Our extension to non-equivocation contracts can provide integrity protection for distributed systems and enhance the security of the systems equipped with other data protection schemes, such as data encryption [4,28,29,37] and access control [24,25]. For instance, in a system that requires users to trust in a service provider for data integrity, the service provider may choose to equivocate and show different users different states of the system. With the non-equivocation contract built based on our scheme, if the service provider equivocates to two users, i.e., shows different states of the systems, it will be penalized by losing the funds locked in a deposit and the user who suffers from the equivocation will get compensation from the deposit. This section explains how to extend our deposits to non-equivocation contracts.

8.1. Non-equivocation contracts between two parties

Non-equivocation contracts between two parties-the sender A and the receiver B, are also built on the idea that with accountable assertions, it is possible to learn the key sk_A if the sender A equivocates. The sender A creates a time-locked deposit as a guarantee for his/her honest behavior. The deposit is secured by the sender's secret key sk_A; the corresponding public key is pk_A. Furthermore, the deposit expires at some point T in the future. That is, even though A owns the secret key sk_A, he/she cannot access the funds in the deposit until time T. Before time T, accessing the funds requires cooperation between the receiver B (or parties if appropriate) and a predefined beneficiary P. Therefore, A defines the output scripts of the deposit with B's public key pk_B and P's public key pk_P as introduced in Section 4.1.1, and the design of the scripts is referred to the output scripts given in Section 5.1. Hence, the receiver B and the beneficiary P will be given the funds if A equivocates.

Once the deposit is confirmed by the Bitcoin network, party B is ready to receive an accountable assertion generated under *statement st* and *ct* from the sender A. Here *st* should be a unique statement associate with the unique *context ct*. In an example scenario, a service provider violates the linearity of the system by showing contradicting states to different users, which can be considered as an equivocation. Although clients can cryptographically verify the append-only property, i.e., that a new system state is a proper extension of an old known system state, a malicious server can still provide different extensions to different clients. To build non-equivocation contracts in this scenario, the *context* ct is a revision number of the state, and the *statement* st is a digest of the state itself at this revision number.

The accountable assertions scheme allows the user A to produce assertions of statements st in contexts ct under the public key pk_A. If A behaves honestly, sk_A will stay secret, and A can use it to withdraw the deposit once time T has been reached. However, if A equivocates to some honest users B and C, i.e., A asserts two different statements $st_0 = st_1$ in the same *context* ct, then P can use st_0, st_1, ct and the two corresponding assertions received from B and C and to extract the sender's secret key sk_A. Then B and P can use sk_A together with his/her credentials to withdraw the deposit and thereby penalize the malicious sender A.

8.2. Non-equivocation contracts between multiple parties

Non-equivocation contracts can also be built between the sender A and multiple receiving parties B_i, $i = (1, 2, \dots, n)$. The sender A creates a Bitcoin key pair (pk, sk). Also, A sets up the accountable assertion scheme with the Bitcoin key pair (pk, sk) as introduced in Section 2.3.

Then A collects the receivers' public keys pk_{Bi}, $i = (1, 2, \dots, n)$, and the beneficiary P's public key pk_P. A then creates a deposit of d' coins with expiry time T using these pk_{Bi}, $i = (1, 2, \dots, n)$, pk_P, and pk_A as introduced in Section 4.2. The value of coins locked in the deposit should be equal to $d' = \sum_{i=1}^{m} d_{\max_i} + \Delta \times m$, where $d_{\max_i}$ is the ith maximum value among the n transactions, and Δ is the incentive for the beneficiary to detect the non-equivocation. After that, every party B_i, $i \in \{1, 2, \dots, n\}$ expects to receive asserted statements from A waits until the transaction that creates the deposit has been confirmed by the Bitcoin network. The usage of this deposit can be described as follows:

1. Whenever A is supposed to send a statement st_i to different protocol parties in a *context* ct_i, party A additionally sends an assertion $\tau_i \leftarrow Assert(ask, auxsk, ct_i, st_i)$.
2. Each payee B_i verifies that $Verify(apk, ct_i, st_i, \tau_i) = 1$ and the expiry time. B_i ignores the message if any of the checks fail. Otherwise, B_i sends the record $(apk, ct_i, st_i, \tau_i)$ to the beneficiary P, who will store it.
3. After that, if P detects an equivocation in two records $(apk, ct_i, st_{i_0}, \tau_{i_0})$ and $(apk, ct, st_{i_1}, \tau_{i_1})$, he/she uses the corresponding assertions to extract A's secret key $sk \leftarrow Extract(apk, ct_i, st_{i_0}, st_{i_1}, \tau_{i_0}, \tau_{i_1})$. Then P cooperates with receive parties to transfer the funds in the deposit as introduced in Section 6.2.

9. Related work

In this section, we summarize the related work in non-equivocation contracts, incentivized computation in Bitcoin, and reputation systems.

9.1. Non-equivocation contracts

Non-equivocation contracts are a form of smart contract [8,9,21]. To ensure that a secret key obtained through equivocation is indeed associated with funds, every party that should be prevented from equivocating is required to put aside a certain amount of funds in a deposit [1,6,8,23]. In the deposit schemes, the funds are time-locked in the deposit, i.e., the depositor cannot withdraw them during a predetermined time period. On the other hand, deposits with explicit beneficiaries and payment channels are possible to be made even without time-locked features [2,8].

9.2. Incentivized computation in bitcoin

In Bitcoin network, the proof of work are undertaken by all miners who are rewarded for validating blocks by incentive coins. However, being the first to successfully verify a block (i. e., being the first to finnd a valid nonce) happens only with a very small probability. Miners therefore often group into mining pools where multiple miners contribute to the block generation conjointly. Multiple different payout functions are used for sharing the profits in mining pools [32]. However, If the controlled supply of coins continues as specified, approximately in the year 2032 the reward will be less than 1 BTC, and in the year 2140 it will be down to zero. According to [11], this kind of deflation is a self-destruction mechanism. It puts the security of crypto currencies at risk by driving of miners. Whether the transaction fees will suffice to compensate the decreasing reward and to provide the necessary incentive for miners remains unclear and is controversially discussed [22]. It is obvious that our scheme provide a new way to reward the miners with coins.

9.3. Reputation systems

Credit systems where users are rewarded for good work and fined for cheating (assuming a trusted arbiter/supervisor in some settings) are proposed in [5,18]. Fair secure computation with reputation systems was considered in [3]. Particularly, in Bitcoin network, it is possible to trace transactions back in history. Therefore, even if a double-spending transaction is successful, the blockchain allows nodes to recognize double-spendings and to identify the tainted coins [17]. The victim will likely keep an eye on these coins and track their flow. Other traders might not be willing to accept tainted coins, because they will always be associated with a fraud. This leads to blacklisting and whitelisting considerations. Moser et al. provided first thoughts on quantifying and predicting the risks that are involved [27].

10. Conclusion

In this paper, we propose fair deposits against double-spending for Bitcoin transactions. The fair deposits can be used to prevent the collusion attacks between the payer and the beneficiary, and guarantee the compensation to the payee's loss. We first provide a solution to make a deposit for one transaction, including both the transaction with single input and output and the transaction with multiple inputs and outputs. We analyse the performance of our fair deposits and show that it has an acceptable overhead. Considering that double-spending transactions happen with a low probability in Bitcoin, we extend our fair deposits to multiple transactions, so as to reduce the payer's cost per transaction. To allow any random miner to perform as the beneficiary of a deposit, we introduce an extension of the fair deposits without an explicit beneficiary. We also extend our fair deposits to non-equivocations contracts to prevent equivocations in other distributed systems.

Acknowledgment

This work was supported by the Singapore National Research Foundation under NCR Award Number NRF2015NCR-NCR003-002. We thank the associate editor and the two reviewers whose insightful comments and suggestions helped improve and clarify this manuscript.

Appendix. Scripts validation

To test wether a transaction that transfers the coins locked in the deposit is valid, we evaluate the stack over time. During the script validation, the *scriptSig* is prefixed to the beginning of the *scriptPubKey* and executed. The operations are executed one at a time, starting with the *scriptSig* and continuing to the end of the *scriptPubKey*.

A.1. Output_B scripts validation

Figure 9 shows the evaluation of *scriptPubKey* provided by the payee when the payer double-spends; and the process can be described as follows:

- The elements of the payee's *scriptSig* are added to an empty stack. Because they are all just data, nothing is done except adding them to the stack.
- From the payer's *scriptPubKey*, OP_IF is executed first, popping the top most value from the stack. The statements in the OP_IF will execute if the popped value is not False, otherwise the statements in the OP_ELSE will be executed. Since it is True, the statement block under OP_IF will be next to be executed.
- Next, the OP_DUP operation pushes a copy of the data currently at the top of the stack, therefore creating a copy of PK_B.
- OP_HASH160 then pops the top most element and pushes a hash of it onto the stack. This creates a hash of PK_B.
- The script then pushes to the stack a copy of the hash of the payee's public key that was received before the deposit was made. At this point, there should be two copies of PK_B hash at the top of the stack.

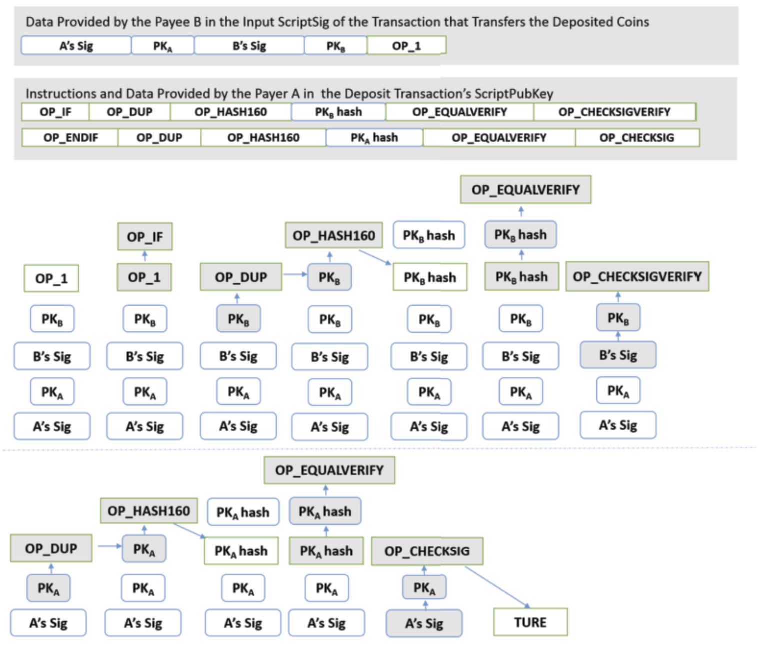

Data Provided by the Payee B in the Input ScriptSig of the Transaction that Transfers the Deposited Coins
A's Sig
PK_A
B's Sig
PK_B
OP_1
Instructions and Data Provided by the Payer A in the Deposit Transaction's ScriptPubKey
OP_IF
OP_DUP
OP_HASH160
PK_B hash
OP_EQUALVERIFY
OP_CHECKSIGVERIFY
OP_ENDIF
OP_DUP
OP_HASH160
PK_A hash
OP_EQUALVERIFY
OP_CHECKSIG
OP_EQUALVERIFY
OP_CHECKSIGVERIFY
OP_CHECKSIG
TURE

Fig. 9. Evaluation stack during the Output_B script validation when the payer double-spends.

- Then, OP_EQUALVERIFY, the equivalent of executing OP_EQUAL followed by OP_VERIFY, checks the two values at the top of the stack. Here, it checks if the two hashes at the top of the stack are equal. If they are indeed equal, True will be pushed to the stack, and False otherwise. At this point, it pops the top value of the stack (the result of OP_EQUAL) and checks it. If the value is False, it immediately terminates evaluation and the transaction validation fails. Otherwise, the validation proceeds with the next operation.
- OP_CHECKSIGVERIFY then checks the signature provided by the payee with the public key. If the signature matches the public key and was generated using all of the data required to be signed, OP_CHECKSIGVERIFY will allow the next operation to continue. Similar to OP_EQUALVERIFY, if the signature does not match the public key, the evaluation terminates and the transaction validation fails.
- Similar to what was done with PK_B, OP_DUP, OP_HASH160, and OP_EQUALVERIFY is then executed on PK_A.
- Finally, OP_CHECKSIG will check the signature of the payer and her public key. OP_CHECKSIG will push the value True onto the top of the stack if they are indeed matching.

If false is not at the top of the stack after *scriptPubKey* is executed, or if validation does not terminate prematurely, the transaction is deemed valid.

In the event of no double-spending, the payee's secret key cannot be recovered by the beneficiary and the payee. Then, the *scriptSig* will be provided by the payer, containing her signature and her public key. The evaluation of the scripts is similar to that in the equivocation case, except that the OP_ELSE statement block will be executed. Figure 10 shows the evaluation of the stack in this case; and this process can be described as follow.

- By putting OP_0 at the top of the stack, the OP_ELSE statement block will be executed.
- The lock time is then pushed to the stack and the OP_CHECKLOCKTIMEVERIFY will execute. This operation takes the top most element of the stack and compares it to the transaction's nLock-

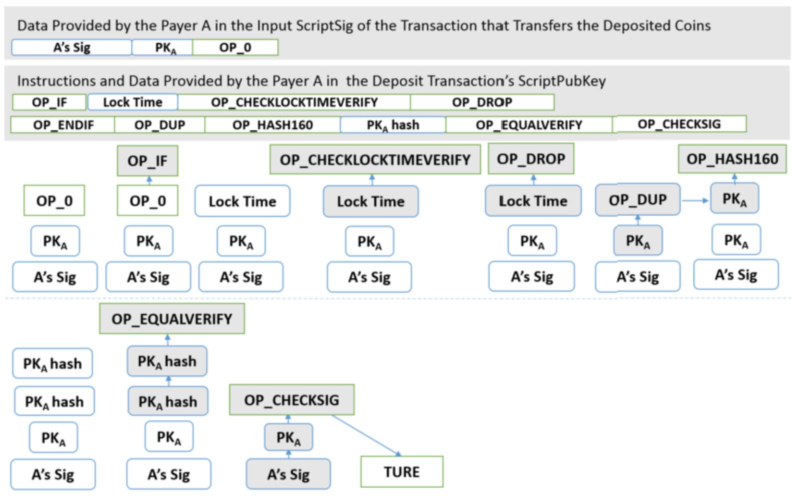


Fig. 10. Evaluation stack during the Output_B script validation when no double-spending.

Time field. If the top most element is greater than nLockTime, the validation of the script continues. If the top most element is smaller than nLockTime or negative, or larger than 500,000,000, or if the stack is empty, the transaction fails.

- Since lock time is not removed by OP_CHECKLOCKTIMEVERIFY, OP_DROP will remove lock time.
- The rest is similar to the last few operations given in the equivocation case.

A.2. Output_P scripts validation

We now examine the stack execution for Output_P scripts. In the case of an equivocation, similar to Output_B, the OP_IF statement block is needed. Note that for Output_P, the scripts make use of the Pay-to-Script-Hash (P2SH) structure, where the *redeemScript* is first hashed and checked using OP_EQUAL before validating *scriptPubKey*. Figure 11 shows the evaluation of *scriptPubKey* provided by the beneficiary when the payer double-spends; and the process can be described as follows:

- The beneficiary's *scriptSig*'s elements are pushed to an empty stack. Because they are all just data, nothing is done except adding them to the stack.
- OP_IF then pops OP_1, the top most element of the stack currently. This allows execution of the OP_IF block.
- Then, OP_2, PK_P, PK_B, and another OP_2 are pushed to the stack to prepare for OP_CHECKMULTISIGVERIFY.
- OP_CHECKMULTISIGVERIFY compares the signatures against the public keys until it finds a ECDSA match. The check process is repeated till all signatures have been checked or if there are not enough public keys remaining to produce a successful result. Care must be taken to place the signatures and public keys in order to prevent failure. Due to an off-by-one error, OP_CHECKMULTISIGVERIFY takes another element of the stack, OP_0 in this case, which will be consumed but not used in this operation. After checking the signatures, it returns True if the

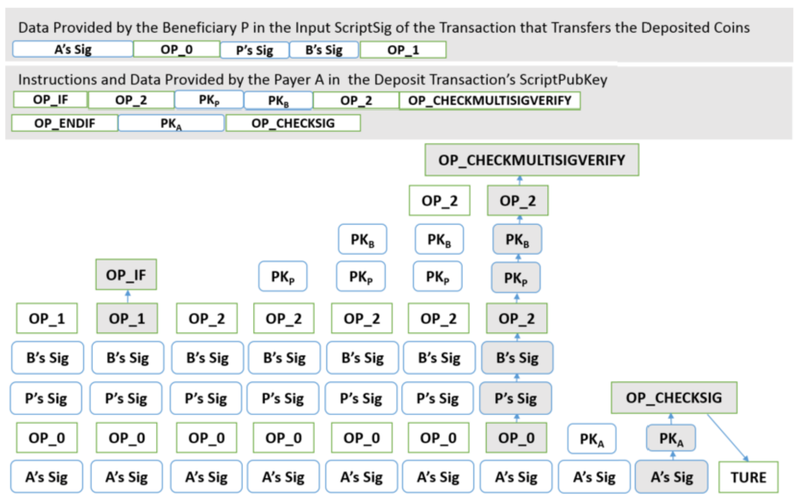

Data Provided by the Beneficiary P in the Input ScriptSig of the Transaction that Transfers the Deposited Coins
A's Sig
OP_0
P's Sig
B's Sig
OP_1
Instructions and Data Provided by the Payer A in the Deposit Transaction's ScriptPubKey
OP_IF
OP_2
PK$_P$
PK$_B$
OP_2
OP_CHECKMULTISIGVERIFY
OP_ENDIF
PK$_A$
OP_CHECKSIG
OP_CHECKMULTISIGVERIFY
OP_IF
OP_CHECKSIG
OP_2
PK$_B$
PK$_P$
OP_1
B's Sig
P's Sig
OP_0
A's Sig
PK$_A$
TURE

Fig. 11. Evaluation stack during the Output_P script validation when the payer double-spends.

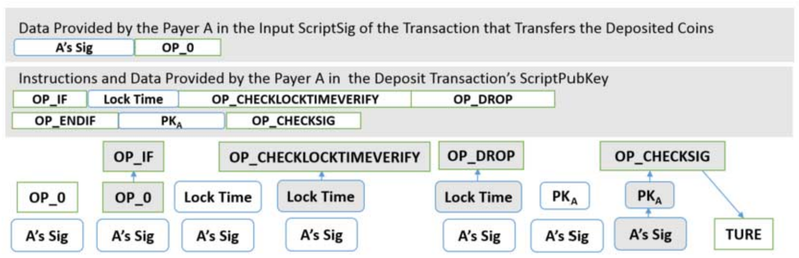


Fig. 12. Evaluation stack during the Output_P script validation when no double-spending.

signatures check out, and False if they do not. It then executes OP_VERIFY, which pops the result of the signature checking and allows the validation to proceed if it is True and terminates the validation process if it is False.

- PK_A is pushed to the stack, where OP_CHECKSIG will consume it with the signature left on the stack to output the final script validation result.

If the payer is honest, he/she will be able to obtain full control over the deposit, including the deposit locked for the beneficiary. Figure 12 shows the evaluation stack during Output_P scripts validation if the payer does not double-spend. The process can be described as follows:

- By putting OP_0 at the top of the stack, the OP_ELSE statement block will be executed.
- The lock time is then pushed to the stack and the OP_CHECKLOCKTIMEVERIFY will execute. This operation takes the top most element of the stack and compares it to the transaction's nLockTime field. If the top most element is greater than nLockTime, the validation of the script continues. If the top most element is smaller than nLockTime or negative, or larger than 500,000,000, or if the stack is empty, the transaction fails.
- Since lock time is not removed by OP_CHECKLOCKTIMEVERIFY, OP_DROP will remove lock time.
- Finally, OP_CHECKSIG will check the signature of the payer and her public key. OP_CHECKSIG will push the value True onto the top of the stack if they are indeed matching.

Therefore, similar to Output_B in the case of no equivocation, the evaluation of the stack for Output_P will return True if the payer prepares *scriptSig* accordingly, resulting in successful deposit redeem.

References

[1] M. Andrychowicz, S. Dziembowski, D. Malinowski and L. Mazurek, Secure multiparty computations on bitcoin, in: *Security and Privacy (SP), 2014 IEEE Symposium on*, IEEE, 2014, pp. 443–458. doi:10.1109/SP.2014.35.

[2] M. Andrychowicz, S. Dziembowski, D. Malinowski and Ł. Mazurek, How to deal with malleability of bitcoin transactions, 2013, arXiv preprint arXiv:1312.3230.

[3] G. Asharov, Y. Lindell and H. Zarosim, Fair and efficient secure multiparty computation with reputation systems, in: *Advances in Cryptology – ASIACRYPT 2013*, Springer, 2013, pp. 201–220. doi:10.1007/978-3-642-42045-0_11.

[4] B. Balamurugan and P.V. Krishna, Extensive survey on usage of attribute based encryption in cloud, *Journal of emerging technologies in web intelligence* **6**(3) (2014), 263–272.

[5] M. Belenkiy, M. Chase, C.C. Erway, J. Jannotti, A. Küpçü and A. Lysyanskaya, Incentivizing outsourced computation, in: *Proceedings of the 3rd International Workshop on Economics of Networked Systems*, ACM, 2008, pp. 85–90. doi:10.1145/1403027.1403046.

[6] I. Bentov and R. Kumaresan, How to use bitcoin to design fair protocols, in: *Advances in Cryptology – CRYPTO 2014*, Springer, 2014, pp. 421–439. doi:10.1007/978-3-662-44381-1_24.

[7] Bitcoin Project, Bitcoin developer guide. https://bitcoin.org/en/developer-guide#stratum.

[8] Bitcoin Wiki, Providing a deposit. https://en.bitcoin.it/w/index.php?title=Contracts&oldid=50633\sharpExample_1:_Providing_a_deposit.

[9] V. Buterin, A next-generation smart contract and decentralized application platform. https://github.com/ethereum/wiki/wiki/White-Paper.

[10] coindesk, 6 Cool Machines that Accept Bitcoin. https://www.coindesk.com/6-cool-machines-accept-bitcoin/.

[11] N.T. Courtois, On the longest chain rule and programmed self-destruction of crypto currencies, 2014, arXiv preprint arXiv:1405.0534.

[12] C. Darryn Pollock, Japan's Electronics Marketplace Starts Adopting Bitcoin. https://cointelegraph.com/news/japans-electronics-marketplace-starts-adopting-bitcoin.

[13] C. Decker and R. Wattenhofer, Information propagation in the bitcoin network, in: *IEEE P2P 2013 Proceedings*, IEEE, 2013, pp. 1–10.

[14] R. Dennis and G. Owenson, Rep on the roll: A peer to peer reputation system based on a rolling blockchain, *International Journal for Digital Society* **7**(1) (2016), 1123–1134. doi:10.20533/ijds.2040.2570.2016.0137.

[15] A. Dorri, S.S. Kanhere, R. Jurdak and P. Gauravaram, Lsb: A lightweight scalable blockchain for iot security and privacy, 2017, arXiv preprint arXiv:1712.02969.

[16] J. Garay, A. Kiayias and N. Leonardos, The bitcoin backbone protocol: Analysis and applications, in: *International Conference on the Theory and Applications of Cryptographic Techniques*, 2015, pp. 281–310.

[17] A. Gervais, V. Capkun, S. Capkun and G.O. Karame, Is bitcoin a decentralized currency? 2014.

[18] P. Golle and I. Mironov, Uncheatable distributed computations, in: *Topics in Cryptology – CT-RSA 2001*, Springer, 2001, pp. 425–440. doi:10.1007/3-540-45353-9_31.

[19] B.K. Helms, South Africa's Second Largest Supermarket Chain Pick n Pay Trials Bitcoin Payments. https://news.bitcoin.com/south-africas-second-largest-supermarket-chain-pick-n-pay-trials-bitcoin-payments/.

[20] G.O. Karame, E. Androulaki and S. Capkun, Double-spending fast payments in bitcoin, in: *ACM Conference on Computer and Communications Security*, 2012, pp. 906–917.

[21] A. Kosba, A. Miller, E. Shi, Z. Wen and C. Papamanthou, Hawk: The blockchain model of cryptography and privacy-preserving smart contracts, Technical report, Cryptology ePrint Archive, Report 2015/675, 2015. http://eprint.iacr.org.

[22] J.A. Kroll, I.C. Davey and E.W. Felten, The economics of bitcoin mining, or bitcoin in the presence of adversaries, in: *Proceedings of WEIS*, Vol. 2013, Citeseer, 2013.

[23] R. Kumaresan and I. Bentov, How to use bitcoin to incentivize correct computations, in: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2014, pp. 30–41.

[24] C. Langaliya and R. Aluvalu, Enhancing cloud security through access control models: A survey, *International Journal of Computer Applications* **112**(7) (2015), 8–12.

[25] C.-C. Lee, P.-S. Chung and M.-S. Hwang, A survey on attribute-based encryption schemes of access control in cloud environments, *IJ Network Security* **15**(4) (2013), 231–240.

[26] B.I.S.J. Lin, A café opened in Singapore that accepts bitcoin instead of cash and has a cryptocurrency ATM for people running low. https://www.businessinsider.com/cashless-bitcoin-cafe-singapore-2017-12/?IR=T.

[27] M. Möser, R. Böhme and D. Breuker, Towards risk scoring of bitcoin transactions, in: *Financial Cryptography and Data Security*, Springer, 2014, pp. 16–32.

[28] J. Ning, Z. Cao, X. Dong, H. Ma, L. Wei and K. Liang, Auditable s-times outsourced attribute-based encryption for access control in cloud computing, *IEEE Transactions on Information Forensics and Security* **13**(1) (2018), 94–105.

[29] J. Ning, X. Dong, Z. Cao, L. Wei and X. Lin, White-box traceable ciphertext-policy attribute-based encryption supporting flexible attributes, *IEEE Transactions on Information Forensics and Security* **10**(6) (2015), 1274–1288. doi:10.1109/TIFS.2015.2405905.

[30] PELICOIN, Bitcoin Vending Machines: The Next Bitcoin Machine You'll See Everywhere. https://www.pelicoin.com/blog/bitcoin-vending-machines.

[31] B.J. Redman, The Evolution of the Bitcoin Vending Machine. https://news.bitcoin.com/evolution-bitcoin-vending-machine/.

[32] M. Rosenfeld, Analysis of bitcoin pooled mining reward systems, 2011, arXiv preprint arXiv:1112.4980.

[33] RT Question More, First restaurant in Russian capital accepts payment in bitcoin. https://www.rt.com/business/394406-russian-restaurant-accepts-bitcoin/.

[34] T. Ruffing, A. Kate and D. Schröder, Liar, liar, coins on fire!: Penalizing equivocation by loss of bitcoins, in: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2015, pp. 219–230.

[35] T. Ruffing, A. Kate and D. Schröder, Implementation of accountable assertion scheme. http://crypsys.mmci.unisaarland.de/projects/PenalizingEquivocation/.

[36] SpendBitcoins, Fast Food Restaurants that accept bitcoin in United States. http://spendbitcoins.com/places/c/fast-food/.

[37] P. Xu, S. He, W. Wang, W. Susilo and H. Jin, Lightweight searchable public-key encryption for cloud-assisted wireless sensor networks, in: *IEEE Transactions on Industrial Informatics*, 2017.

[38] X. Yu, M.S. Thang, Y. Li and R.H. Deng, Fair deposits against double-spending for bitcoin transactions, in: *IEEE Conference on Dependable and Secure Computing*, 2017.

[39] S.-W. Zheng and L. Fan, Credit model based on p2p electronic cash system bitcoin, *Information Security and Communications Privacy* **3** (2012), 040.