8-2017

Fair deposits against double-spending for Bitcoin transactions

Xingjie YU
Singapore Management University

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

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

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

DOI: https://doi.org/10.1109/DESEC.2017.8073796

Citation

YU, Xingjie; THANG, Shiwen M.; LI, Yingjiu; and DENG, Robert H.. Fair deposits against double-spending for Bitcoin transactions. (2017). *2017 IEEE Conference on Dependable and Secure Computing: Taipei, Taiwan, August 7-10: Proceedings*. 44-51. Research Collection School Of Information Systems.
Available at: https://ink.library.smu.edu.sg/sis_research/3897

Fair Deposits against Double-Spending for Bitcoin Transactions

Xingjie Yu, Michael Thang Shiwen, Yingjiu Li, Robert Deng Huijie
School of Information Systems
Singapore Management University
Email: 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 blockchain opens up possibilities for double spending, i.e., a payer issues two separate transactions to two different payees transferring the same coins. To detect the double-spending and penalize the malicious payer, decentralized non-equivocation contracts have been proposed. 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 deposit 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 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. We start with proposing protocols of making a deposit for Bitcoin transactions. We then analyze the performance of deposits made for Bitcoin transactions and show how the fair deposits work efficiently in Bitcoin.**

I. Introduction

As a decentralized crypto-currency system, to eliminate the central bank, Bitcoin uses blockchain to take the role of a distributed ledger reflecting all transactions and ownerships. This enables every participant to keep a copy of transaction records which would classically be stored at central banks in traditional banking system. However, the distributed storage of multiple copies of the blockchain increases possibilities for double spending. A payer may perform two separate transactions with two different receivers 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. However, it still cannot entirely eliminate double-spending. Recent research by Rosenfeld et al. demonstrates that it is not possible to reduce the success probability of double spending to zero: an attacker with less than 50% of the total computational power is able to perform a double spend by brute force and a bit of luck [1].

978-1-5090-5569-2/17/$31.00 ©2017 IEEE

Although double spending cannot be completely prevented, we can detect it and penalize the double-spending payer. Ruffing et.al [2] proposed a non-equivocation contract based on accountable assertions which can penalize an equivocating party by the loss of his money. As double-spending in Bitcoin is a special case of equivocation, their non-equivocation contracts can be applied 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 predefined beneficiary or any other miners in the case of an equivocation. After the deposit is confirmed by Bitcoin network, the payer sends an accountable assertion to the payee along with the transaction information and to the beneficiary for storing. If the payer double spends, the beneficiary can extract the payer's secret key with two conflicting assertions, i.e., the payer asserts two different statements in the same context to two different recipients. Then the beneficiary can use the payer's secret key to transfer the funds in the deposit to his own Bitcoin address. If no equivocation occurs, the payer will regain full control of the deposit after the time-lock expires and his secret key remains uncompromised. Therefore, by setting aside a high-enough deposit with a third party, it is expected that the attacker would have no incentive to double spend his transaction.

However, such non-equivocation contracts are subjected to collusion attacks. In a collusion attack, an attacker, who makes a transaction with the recipient, can collude with the beneficiary to transfer the Bitcoin deposit back to the attacker when he equivocates, resulting in no penalties. Moreover, this deposit is unfair to the victim recipient. This is because whether the attacker colludes with the beneficiary or not, there is no compensation to the recipient since the deposit is made for the beneficiary only. In a business perspective, the recipient, who may be a service provider or a product seller, may lose valuable time and effort, and may result in a loss of profits. To defend against the collusion attack and guarantee the compensation to the victim recipient when double-spending happens, a new solution of making deposits is in demand and the value of a deposit should be appropriately determined.

The goal of this paper is to design new protocols for making time-locked deposits to not only defend against double spending in Bitcoin, but also prevent collusion attacks and guarantee the compensation to the victim payee's loss. We first provide a solution to make a deposit for one transaction. In our protocol, the payer needs to create a deposit for his transaction with the payee, so that the payer could be penalized by the loss of his deposit if he double spends. Our protocol uses the assertion scheme as proposed by Ruffing et al. [2]. In particular, the beneficiary can recover the payer's secret key if the payer double spends. However, to ensure that the payee's loss can be compensated if the payer double spends, in addition to a signature generated with the payer's secret key, a signature generated with the payee's secret key is required for the release of the compensation locked in the deposit. Meanwhile, the incentive for the beneficiary is also guaranteed in the deposit.

The rest of this paper is organized as follows. Section II introduces the preliminaries. Section III presents our threat model in Bitcoin network. Section IV presents the design of a fair time-locked deposit for one Bitcoin transaction. Section V implements the setup and usage of fair deposits in Bitcoin network and evaluates its usability and efficiency. Section VI describes the related work. Section VII discusses extensions of our design to deposits for multiple Bitcoin transactions, deposits without explicit beneficiaries, and non-equivocation contracts applicable to various distributed systems. Section VIII concludes this paper.

II. 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.

A. Non-equivocation Contracts

Ruffing et. al [2] 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 attacker 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 recipient(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 can extract A's secret key and use it to transfer the funds in the deposit to his 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 the attacker when he double spends, resulting in no penalties. On the other hand, even if the attacker has double spent the transaction and the beneficiary has behaved honestly, there is no compensation to the victim recipient since the coins locked in the deposit can only be redeemed by the beneficiary.

B. 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 UNIX time, or until a certain number of blocks have been mined. As discussed by Ruffing et. al [2], the payer who creates the time-locked deposit cannot transfer the coins in the deposit with his/her secret key before lock time T. Moreover, for deposits with explicit beneficiary, the lock time of the deposit should include a safe margin $T_{net}^{expl} + T_{conf}^{expl}$. The safe margin ensures that the closing transaction has already been broadcast to the Bitcoin network and confirmed by it before the deposit can be spent by the payer alone. For the broadcast, $T_{net}^{expl} = 10$ min is more than sufficient [3]. For the confirmations, as applied by Ruffing et. al [2], we expect the network to find 24 blocks in $T_{conf}^{expl} = 240$ min. Since their arrival is Poisson-distributed, the probability that fewer than six desired blocks have been found is $Pr[X \leq 5] < 2 - 18$ for $X \sim Pois(24)$. Throughout the paper, the lock time of a deposit is longer than the safe margin of 250 min.

C. Accountable Assertion Scheme

An accountable assertion is a cryptographic primitive introduced by Ruffing et. al [2]. The idea of this primitive is to bind *statements* to *contexts* in an accountable way: if the attacker equivocates, i.e., asserts two contradicting statements in the same *context*, then any observer can extract the attacker's Bitcoin secret key and, as a result, use it to force the loss of the attacker'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:

- $(apk, ask, auxsk) \leftarrow Gen(1^{\lambda})$: 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.

III. Threat Model

In our threat model, the payer can maliciously double-spend an input of a Bitcoin transaction, but he cannot control more than 50% of the network's/miners' computation power. Rosenfeld [4] has demonstrated that an attacker 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 the payer can control more than 50% computational power, he can double spend any transaction, including the deposit transaction. Even worse, the attacker who is in control of over 50% of the computation power of all miners combined has a chance of succeeding in rewriting the entire blockchain, which would affect the whole Bitcoin network. Hence, the inability to control more than 50% computation power is one of the fundamental security of Bitcoin. Our assumption complies with this fundamental security. As assumed by Ruffing et. al in [2], we also assume that the payer cannot break other fundamental security of Bitcoin, e.g., the payer cannot break the payee's private key.

Ruffing et. al have proven the security of accountable assertion algorithm in [2], and proposed protocols to defend against non-equivocations in distributed networks, such as double spending in Bitcoin. However, in their threat model, they did not consider the collusion attacks where the payer colludes with the beneficiary. In our threat model, we allow the payer to collude with the beneficiary. In particular, the payer can share his secret key with the beneficiary, and collude with the beneficiary when redeeming the deposit. Although we assume that the beneficiary can collude with the payer, he does not risk to lose his incentive defined in the deposit, which is outlined in Section IV.

IV. 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.

A. 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. Let Party A denote a user (the payer) who makes payment to Party B (the payee). Once A makes a transaction to B spending d coins on B's services (or products), A creates a deposit for this transaction with the beneficiary P. A should lock $d + \Delta$ coins in this deposit. In particular, d coins are used to compensate B's loss if A double spends and Δ coins are the incentive for P to detect A's double-spending.

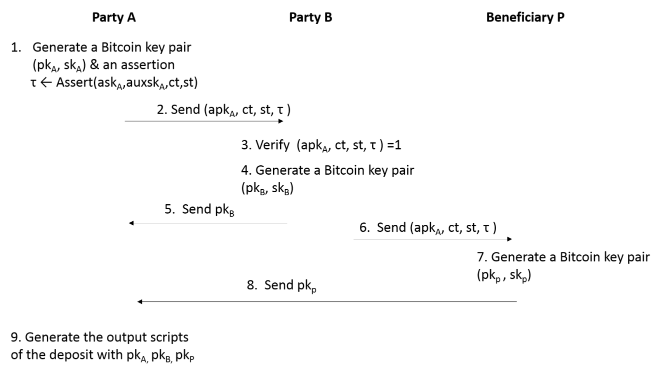

Party A
Party B
Beneficiary P
1. Generate a Bitcoin key pair (pk_A, sk_A) & an assertion $\tau \leftarrow$ Assert(ask_A,$auxsk_A$,ct,st)
2. Send (apk_A, ct, st, τ)
3. Verify (apk_A, ct, st, τ) =1
4. Generate a Bitcoin key pair (pk_B, sk_B)
5. Send pk_B
6. Send (apk_A, ct, st, τ)
7. Generate a Bitcoin 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 with single input and output.

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 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.

1) Deposit Setup: To create a deposit for the transaction in which A transfers d 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. Fig. 1 shows the message flow for setting up a deposit.

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 same 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 [2].

Next A uses the transaction number of the previous output

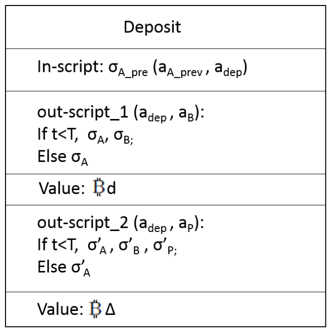

Deposit
In-script: σ_A_pre (a_A_prev , a_dep)
out-script_1 (a_dep , a_B):
If t<T, σ_A, σ_B;
Else σ_A
Value: ₿d
out-script_2 (a_dep , a_P):
If t<T, σ'_A , σ'_B , σ'_P;
Else σ'_A
Value: ₿Δ

Fig. 2. Deposit script for one transaction with single input and output.

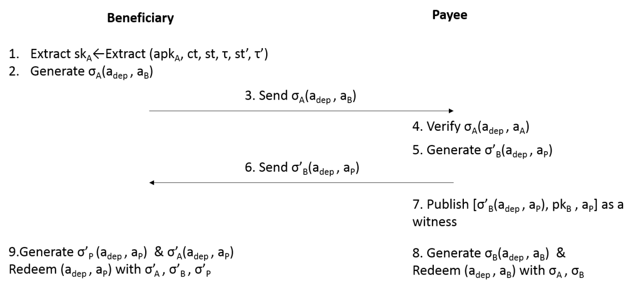

Beneficiary
Payee
1. Extract sk_A←Extract (apk_A, ct, st, τ, st', τ')
2. Generate σ_A(a_dep , a_B)
3. Send σ_A(a_dep , a_B)
4. Verify σ_A(a_dep , a_A)
5. Generate σ'_B(a_dep , a_P)
6. Send σ'_B(a_dep , a_P)
7. Publish [σ'_B(a_dep , a_P), pk_B , a_P] as a witness
9.Generate σ'_P (a_dep , a_P) & σ'_A(a_dep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

as ct and the random number received from B as st to generate an assertion $\tau \leftarrow$ Assert(ask_A,$auxsk_A$,ct,st). 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 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_B, sk_B) of his private address a_P and sends pk_B 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 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, where 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.

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

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 private address only if B has signed the transaction (a_{dep}, a_P) with his secret key sk_B. Therefore, if P wants to gets the d coins in the deposit, he has to generate $\sigma_A(a_{dep}$,$a_B)$ and send it to B first. The safety margins as discussed in Section 2.2 ensure that the transactions created by P and B will have already been confirmed and thus the deposit will have already been withdrawn when its expiry time is reached.

Our design also ensures that B can get his compensation of d coins and P can get Δ coins as his 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 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 future transactions. The possible negative influence on his 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 σ'_B (a_{dep}, a_P), we ensure that only P can transfer Δ coins that are locked in the deposit to his private address a_P by the additional requirement of σ'_P (a_{dep}, a_P) in the release condition Π_P. If σ'_P is not required in the release condition Π_P, A may use his secret key to generate the signature $\sigma'_A(a_{dep}, a_P)$ to transfer the coins to his 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 private address if it recovers A's secret key sk_A or colludes with A, which could result in that B get no compensation.

B. Deposit for Transactions with Multiple Inputs and Outputs

For a transaction with multiple inputs and outputs, deposits should be created for each payee to ensure that any payee who suffers due to double-spending can receive compensation. 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.

1) Deposit Setup: Let Party A denote the payer who makes a transaction with k payees, Party $B_i, i \in \{1, 2, ..., k\}$, denote the $i - th$ payee of the deposited transaction, and Party P_i denote the beneficiary of the deposit made to B_i. A should lock $d_{mult_i} + \Delta_{mult_i}$ coins in the deposit made to B_i with the expiry time T. Here, d_{mult_i} is the total value of coins transferred to B_i in the deposited transaction, and Δ_{mult_i} is the incentive for P_i to detect the A's misbehavior. In particular, Δ_{mult_i} is decided based on the negotiation between A and P_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 Party A's secret key sk_A and Party B_i'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, sk_A should be recoverable from the double-spending of any input. This ensures that B_i can get d_{mult_i} coins from the deposit. 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 $\tau_j, j \in \{1, 2, ..., n\}$ for n inputs respectively. To generate τ_j, ct_j is the transaction number of the previous output which the j-th 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 $(apk_A, ct_j, st_j, \tau_j)$, $j = (1, 2, ..., n)$, to B_i for verification. If all the records are verified as valid, B_i sends all these records to his respective beneficiary P_i.

2) Deposit Usage: After receiving the n records $(apk_A, ct_j, st_j, \tau_j)$, $j = (1, 2, ..., n)$, if P_i detects a different assertion under the same ct_j, he 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 the solution introduced above suggests that each B_i has his own 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 can generate σ_{Ai} on the transaction (a_{dep}, a_{B_i}) using sk_A for all B_i, $i = (1, 2, ..., k)$. 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.

The deposits for transactions with multiple inputs and multiple outputs provide the same security as the deposits for transactions with single input and output. The security analysis of the deposits for transactions with multiple inputs and outputs is given in a full version of this paper [5], due to the length limitation on the conference paper.

V. 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. 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.

A. Implementation

We implement the accountable assertion algorithm based on the codes published online [6] 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. [2] 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. [2].

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 IV, 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. Particularly, we use a 20-byte hexadecimal Bitcoin address as ct. 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. 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 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 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.

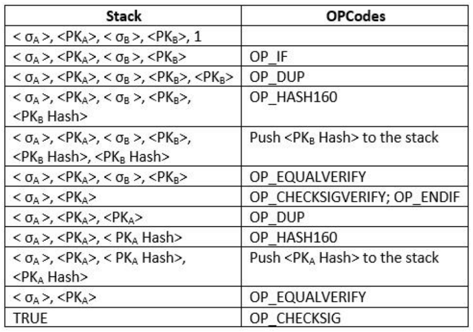
If the payer has not double spent, he can gain full access to his deposit using the following $scriptSig$:

- 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. Hence, the transactions that use our scripts can be supported by Bitcoin network without any modifications on current Bitcoin structures, since Bitcoin nodes which accept transactions using the IF_ELSE structure have already existed in Bitcoin network. The transactions that use our scripts can be broadcasted to such Bitcoin nodes, mined into blocks, and subsequently confirmed by the Bitcoin network.

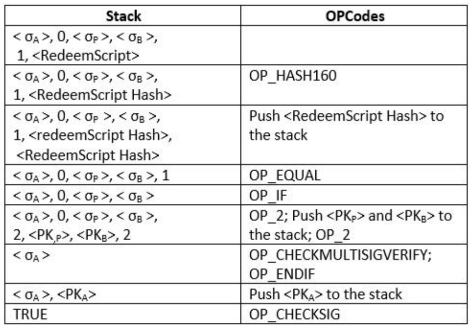
B. Validation and Evaluation

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



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.

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 Figure 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 the full version [5]. We also evaluate the stack during the scripts validation in the event that the payer acts honestly and the corresponding evaluation stacks over time can also be reached in the full version [5]. The results demonstrate that the transactions that redeem the deposit using our scripts are also valid when the payer acts honestly.

2) Performance Evaluation: We evaluate the overhead caused by the accountable assertion algorithm. For each round of the experiment, 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

TABLE I
ACCOUNTABLE ASSERTION PERFORMANCE EVALUATION.

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

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 I.

The experiments are perfomed on a 2.4GHz (Intel Core i5-4258U) machine with a DDR3-1600MHz RAM. Ruffing et al. [2] also evaluate the overhead of their design caused by assertion generation and assertion verification. Comparing with [2], 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 [2]. However, the computational overhead of our design is still millisecond-level, hence it is still manageable and acceptable.

VI. RELATED WORK

A. Non-equivocation Contracts

Non-equivocation contracts are a form of smart contract [8], [9], [10]. 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 [10], [11], [12], [13]. 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 [10], [14].

B. 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 [4]. 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 [15], 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 [16]. It is obvious that our scheme provide a new way to reward the miners with coins.

C. 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 [17], [18]. Fair secure computation with reputation systems was considered in [19]. 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 [20]. 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 [21] .

VII. Discussion

In Bitcoin network, double spending transactions happens with a low probability. Given that, if we create a deposit for every output of bitcoin transactions, it would result in locking a large number of coins in deposits. Especially, for the users who only have a relatively small number of coins, it would significantly limit their activities on Bitcoins network and obstruct their normal transactions. Moreover, much transaction fees would arise from the transactions of making deposits even for an honest user who has never double spent and has no intention of equivocating in the future. Therefore, to reduce the deposit value and transaction fees for a honest payer, we design a protocol allowing a payer to make just one deposit for multiple transactions. Our protocol ensures that the victim payee of the double-spending transaction get a fair enough compensation from the deposit. The details of this extension can be reached in the full version of this paper [5].

In some cases, the beneficiary of the deposit is a randomly selected miner rather than an explicit beneficiary. In such cases, we provide an extension of the deposit protocol without any explicit beneficiary. We also extend our deposit protocol to non-equivocation contracts. These extensions are also given in our full version [5]. Compared to the non-equivocation contracts proposed in [2], 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.

VIII. Conclusion

In this paper, we proposed 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 provided 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 also analysed the performance of our fair deposits and showed that it has an acceptable overhead.

References

[1] M. Rosenfeld, "Analysis of hashrate-based double spending," *arXiv preprint arXiv:1402.2009*, 2014.

[2] 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.

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

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

[5] "Fair deposits against double-spending for bitcoin transactions (full version)," [EB/OL], https://www.dropbox.com/s/3aeaffef79l11jz/Fair_deposit_Full%20Version.pdf?dl=0.

[6] T. Ruffing, A. Kate, and D. Schröder, "Implementation of accountable assertion scheme."

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

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

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

[10] "Providing a deposit." https://en.bitcoin.it/w/index.php?title=Contracts&oldid=50633#Example_1:_Providing_a_deposit, bitcoin Wiki.

[11] 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.

[12] I. Bentov and R. Kumaresan, "How to use bitcoin to design fair protocols," in *Advances in Cryptology–CRYPTO 2014*. Springer, 2014, pp. 421–439.

[13] 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.

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

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

[16] 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.

[17] 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.

[18] P. Golle and I. Mironov, "Uncheatable distributed computations," in *Topics in CryptologyCT-RSA 2001*. Springer, 2001, pp. 425–440.

[19] G. Asharov, Y. Lindell, and H. Zarosim, "Fair and efficient secure multi-party computation with reputation systems," in *Advances in Cryptology-ASIACRYPT 2013*. Springer, 2013, pp. 201–220.

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

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