Fair Two-Party Computations via the BitCoin Deposits

Marcin Andrychowicz[⋆], Stefan Dziembowski[⋆⋆], Daniel Malinowski[⋆⋆⋆] and Łukasz Mazurek[†]

University of Warsaw

Abstract. We show how the BitCoin currency system (with a small modification) can be used to obtain fairness in any two-party secure computation protocol in the following sense: if one party aborts the protocol after learning the output then the other party gets a financial compensation (in BitCoins). One possible application of such protocols is the fair contract signing: each party is forced to complete the protocol, or to pay to the other one a fine.

We also show how to link the output of this protocol to the BitCoin currency. More precisely: we show a method to design secure two-party protocols for functionalities that result in a "forced" financial transfer from one party the other.

Our protocols build upon the ideas of our recent paper "Secure Multiparty Computations on BitCoin" (Cryptology ePrint Archive, Report 2013/784). Compared to that paper, our results are more general, since our protocols allow to compute any function, while in the previous paper we concentrated only on some specific tasks (commitment schemes and lotteries). On the other hand, as opposed to "Secure Multiparty Computations on BitCoin" to obtain security we need to modify the BitCoin specification so that the transactions are "non-malleable" (we discuss this concept in more detail in the paper). We hope that for this reason our results can serve as a motivation for future improvements of BitCoin, and as a help for designers of new digital currency systems.

1 Introduction

In our recent paper [2] we put forward a new concept dubbed "secure multiparty computations (MPCs) on BitCoin". On a high level the idea of this concept is as follows. Recall that the MPCs [22,13] are protocols that allow a group of mutually distrusting parties to "emulate" a trusted functionality in a secure way. Examples of such functionalities include lotteries, auctions, voting schemes and many more. It is known since 1980s that for any efficiently-computable functionality there exists an efficient protocol that emulates it, assuming that the majority of the participants is honest and that certain computational problems are intractable. If there is no honest majority (in particular: if there are just two parties and one of them is cheating), then such protocols also exist, but in general they do not provide *fairness*, i.e. a dishonest party can prevent the other parties from learning their outputs, after she learned it herself [8,18].

[⋆] marcin.andrychowicz@crypto.edu.pl

[⋆⋆] stefan.dziembowski@crypto.edu.pl

[⋆⋆⋆] daniel.malinowski@crypto.edu.pl

[†] lukasz.mazurek@crypto.edu.pl

Despite of their great importance both to the theory and applications, the MPC protocols suffer from some inherent limitations. The first one is the above-mentioned lack on fairness when the majority of the participants is dishonest. The second is that the standard security definition of MPCs does not ensure that the parties provide the inputs to the computations in an honest way, and that they respect the outcome. For example, in most of the settings it is clearly impossible to guarantee in a cryptographic way that a bidder in an auction has enough money to pay his bid, or that the loosing party will accept the outcome of the voting procedure. BitCoin, due to its fully distributed nature, and the fact that the list of transactions is publicly known, gives an attractive opportunity to go beyond this barrier. In [2] we discuss this idea, and provide some examples of how it can be used. The main technical contribution of that paper is a protocol for a multiparty lottery with a very strong security property: each honest party can be sure that, once the game starts, it will be fair, and she will be payed the money in case she wins. This happens even if the other parties actively cheat, and in particular even if some (or all) of them abort the protocol prematurely. In order to achieve it we use a mechanism that financially penalizes a party that does not follow the protocol.

Our main tool is a special type of a "BitCoin-based timed commitment scheme", that has the following non-standard property: a committer has to pay a "deposit" during the commitment phase, that he gets it back only if he opens his commitment within some specific time. Although the main application of this commitment scheme is the lottery protocol, it can actually also be used to obtain fairness in protocols where the inputs and outputs do not concern BitCoin. One of the questions left open in [2] is to construct protocols for more general functionalities than the commitment scheme or the lottery.

1.1 Our contribution

In this paper we show that a small modification of the BitCoin specification would make it possible to construct protocols for a very general class of functionalities in a two-party settings. Roughly speaking (for more details see Section 3), for our protocols to work we need to assume that the transactions are "non-malleable" in the following sense: we assume that each transaction is identified by the hash of its simplified version (also called the "body" of a transaction in [2]), instead of the hash on the *complete* transaction (i.e. the body and the input scripts) as it is done currently in BitCoin. Assuming this modification, we show how to achieve fairness in any two-party protocol in the following sense. Before learning the output of the computation, each party has to pay some deposit. She is guaranteed to get this money back as long as she behaves honestly until the very end of the protocol, i.e. until the other party learns the output. If she misbehaves then her money is given to the other party.

In practice it will make sense to use this protocol if the potential gain from a premature termination is lower than the deposit that the party pays. As the potential applications of our protocols let us mention the *contract signing* problem, which has been extensively studied in cryptography since 1980s [12,6,10,21]. Informally, the challenge in this line of work is to design the protocols where two parties simultaneously sign a document M in a fair way, i.e. it should be impossible for one party, say Alice, to obtain Bob's signature on M without Bob obtaining Alice's signature on M (and vice-versa).

It was shown by Even and Yacobi [12] that this task is in general impossible to achieve, and since then there has been a substantial effort to overcome this impossibility result in various ways (e.g. by assuming an existence of a trusted third party). Since obviously a signing procedure can be modeled as a two-party functionality, hence one can use our protocol to achieve fairness. If the value of the contract is lower than the deposit payed by each party, then clearly the parties will have no incentive to cheat. Moreover, if one party, say Alice, cheats then Bob will earn Alice's deposit (plus he will get his own deposit back), which will compensate his loses resulting from the fact that Alice cheated during the contract signing protocol. Of course, our protocols can be used in several other applications that rely on a fair exchange of secrets, such as certified e-mail systems [24,3,1] or non-repudiation protocols [23].

We also show how to link the outputs of our protocols to the BitCoin money in the following sense (for more information see Sec. 6). The output of the emulated functionality can contain instructions of a form "Alice sends d ฿ to Bob" or "Bob sends d ฿ to Alice" (where "฿" is the BitCoin currency symbol). Our protocol will enforce that these transfers are indeed performed. Of course, this holds only if the parties conduct the protocol until the very end, but again, if one party decides to abort prematurely then her deposit will be payed to the other party. Hence, if this deposit is larger than d then it clearly makes no economic sense to abort. Of course, one example of a such a functionality is the lottery protocol. We would like to stress, however, that our result does not imply the result of [2], since the protocols of [2] work on the current version of BitCoin protocol (without any modification).

One can, of course, imagine several other applications of our protocols. For example, one can construct protocols for buying digital goods that can be specified by any poly-time computable functions $\pi : \{0,1\}^* \to \{\mathrm{true}, \mathrm{false}\}$. More precisely: imagine that Alice promises Bob that she will pay him 1 ฿ if he sends her a file $m \in \{0,1\}^*$ such that $\pi(m) = \mathrm{true}$, however she does not want to reveal this function neither to Bob nor to the public. Then, we can construct such a protocol that emulates the following functionality: the input of Alice is π and the input of Bob is m. If $\pi(m) = \mathrm{true}$ then the output is m and a "forced transfer of 1 ฿ from Alice to Bob", otherwise the output is $\perp$. Such π can be, e.g., a function that checks if m is a secret that concerns a certain person.[1]

On a technical level, our protocols are based on a new variant of a BitCoin-based timed commitment scheme that we call the "simultaneous commitment" and denote SCS. It can be viewed as an extension of the BitCoin-based commitment scheme from [2] described above. The main difference is that it forces both users to *simultaneously* commit to their secrets. In other words, the commitment of each party is valid (and she is forced to open it by some time t) only if the other party made her corresponding commitment at the same time.

[1] A real-life example of such situation is the recent case when the German tax authorities payed 4 million euro to an anonymous informant for a CD containing information about the German tax evaders with bank accounts in Switzerland [11].

1.2 Related work

As described above our paper builds upon the ideas from our previous paper [2], and hence most of the work relevant to that paper is also relevant to this one. Usage of BitCoin to achieve fairness in a case of a two-player lottery has been independently proposed by Back and Bentov in [4]. Similarly to this paper, their protocol works under the assumption that the transactions are non-malleable.

Improvements to BitCoin have already been suggested in an important work of Barber et al. [5] who study various security aspects of BitCoin and Miers et al. [19] who propose a BitCoin system with provable anonymity.

The idea to use some concepts from the MPC literature appeared already in Section 7.1 of [5] where the authors construct a secure "mixer", that allows two parties to securely "mix" their coins in order to obtain unlinkability of the transactions. They also construct commitment schemes with time-locks, however some important details are different, in particular, in the normal execution of the scheme the money is at the end transferred to the recipient. Also, the main motivation of this work is different: the goal of [5] is to fix an existing problem in BitCoin ("linkability"), while our goal is to use BitCoin to perform tasks that are hard (or impossible) to perform by other methods.

Commitment schemes and zero-knowledge proofs in the context of the BitCoin were already considered in [7], however, the construction and its applications are different — the main idea of [7] is to use the BitCoin system as a replacement of a trusted third party in time-stamping. The notion of "deposits" has already been used in BitCoin (see en.bitcoin.it/wiki/Contracts, Section "Example 1"[2]), but the application described there is different: the "deposit" is a method for a party with no reputation to prove that she is not a spambot by temporarily sacrificing some of her money.

The problem of the malleability of the transactions has been noticed before and described in BitCoin wiki (see en.bitcoin.it/wiki/Transaction_Malleability)

2 A description of BitCoin

An idea of BitCoin was introduced by Satoshi Nakamoto in [20]. We assume reader's familiarity with the basic principles of the BitCoin. In particular, we do not describe how the "BitCoin blockchain" works and how the money is created. Let us only briefly recall that the BitCoin currency system consists of *addresses* and *transactions* between them. An address is simply a public key pk.[3] Normally every such key has a corresponding private key sk known only to one user of the system. The pairs (sk, pk) can be easily generated offline. We will frequently denote key pairs using the capital letters (e.g. A). We will also use the following convention: if $A = (sk, pk)$ then let $\mathsf{sig}_A(m)$ denote a signature on a message m computed with sk and let $\mathsf{ver}_A(m, \sigma)$ denote the result (true or false) of the verification of a signature σ on message m with respect to the public

[2] Accessed on 24.11.2013.

[3] Technically an address is a *hash* of pk. In our informal description we decided to assume that it is simply pk. This is done only to keep the exposition as simple as possible, as it improves the readability of the transaction scripts later in the paper.

key pk. The private key is used for signing transactions, and the public key is used for verifying signatures.

BitCoin is a peer-to-peer network. When a user wants to pay somebody in BitCoins, he creates a transaction and broadcasts it to other nodes in the network. The nodes jointly maintain a public list of all transactions, called the blockchain. After a valid transactions is broadcast, it will appear on the blockchain after some time. Each transaction posted on the blockchain has a time stamp that refers to the moment when it appeared on the blockchain.

2.1 The BitCoin transactions

Currently in the BitCoin system each transaction T_y is identified by its hash $H(T_y)$. Our proposal, described in Section 3, is to change it. Let us however first describe the current version of BitCoin. Each transaction can have multiple inputs and outputs. Inputs of a transaction T_x are listed as triples $(y_1, a_1, \sigma_1), \dots, (y_n, a_n, \sigma_n)$, where each y_i is a hash of some previous transaction T_{y_i}, a_i is an index of the output of T_{y_i} (we say that T_x *redeems the a_i-th output of* T_y) and σ_i is called an *input-script* (this term will be explained in a moment). The outputs of a transaction are presented as a list of pairs $(v_1, \pi_1), \dots, (v_m, \pi_m)$, where each v_i specifies some amount of BitCoins (called the *value of the i-th output of* T_x) and π_i is an *output-script* (we explain this notion in the sequel). A transaction can also have a time-lock t, meaning that it is valid only if time t is reached. Hence, altogether a transaction in the most general form looks like this:

$$T_x = ((y_1, a_1, \sigma_1), \dots, (y_n, a_n, \sigma_n), (v_1, \pi_1), \dots, (v_m, \pi_m), t). \quad (1)$$

The *body of* T_x[4] is equal to T_x without the input-scripts, i.e.: $((y_1, a_1), \dots, (y_n, a_n), (v_1, \pi_1), \dots, (v_m, \pi_m), t)$, and denoted by $[T_x]$. One of the most useful properties of BitCoin is that the users have flexibility in defining the condition on how the transaction T_x can be redeemed. This is achieved by the input- and the output-scripts. One can think of an output-script as a description of a function whose output is Boolean. A transaction T_x from Eq. (1) is valid if for *every* $i = 1, \dots, n$ we have that $\pi'_i([T_x], \sigma_i)$[5] evaluates to true, where π'_i is the output-script corresponding to the a_i-th output of T_{y_i}. Another conditions that need to be satisfied are that the time t has already passed and $v_1 + \cdots + v_m \leq v'_1 + \cdots + v'_n$ where each v'_i is the value of the a_i-th output of T_{y_i}. The difference between the right-hand-side of this inequality and its left-hand-side is called the *transaction fee*. Of course, to be valid T_x can redeem only those outputs of transactions that has not yet been redeemed by some other transaction.

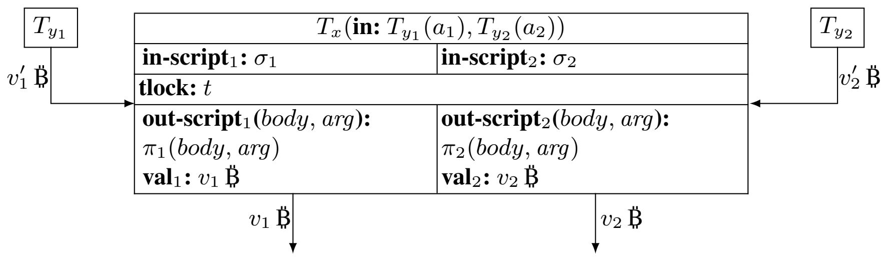
The scripts are written in the BitCoin scripting language, which is a stack based, not Turing-complete language (there are no loops in it). It provides basic arithmetical

[4] In the original BitCoin documentation this is called "simplified T_x"

[5] In reality $[T_x]$ is not directly passed as an argument to π'_i. The only way π'_i can access $[T_x]$ is to use an operation, which takes two arguments — a signature and a public key and checks whether the given signature is a valid signature under the given public key on the body of the transaction we are trying to validate, i.e. $[T_x]$. To make the exposition clearer, we assume in this paper, that $[T_x]$ is passed as an argument, but we use it only in the operation described above.

operations on numbers, operations on stack, if-then-else statements and some cryptographic functions like computing hash functions or verifying a signature. One example of a transaction is $T_x = ((y, a, \sigma), (v, \pi))$ where $\pi(body, \sigma') = \mathsf{ver}_A(body, \sigma')$ is a function that checks if the corresponding input-script is a valid signature on the body of the transaction we are validating with respect to some public key $A.pk$. However, much more general functions π are possible.

Following [2] we will present the transactions as boxes. The redeeming of transactions will be indicated with arrows (which will be labeled with the transaction values). For example a transaction $T_x = ((y_1, a_1, \sigma_1), (y_2, a_2, \sigma_2), (v_1, \pi_1), (v_2, \pi_2), t)$ will be represented as:



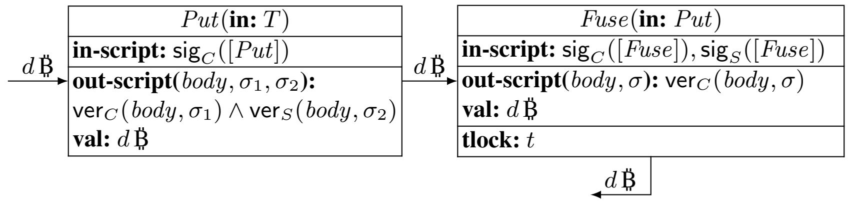
T_{y_1}
T_x(**in:** $T_{y_1}(a_1), T_{y_2}(a_2)$)
T_{y_2}
v_1' ฿
in-script$_1$**:** σ_1
in-script$_2$**:** σ_2
v_2' ฿
tlock: t
out-script$_1$(*body*, *arg*)**:**
π_1(*body*, *arg*)
val$_1$**:** v_1 ฿
out-script$_2$(*body*, *arg*)**:**
π_2(*body*, *arg*)
val$_2$**:** v_2 ฿
v_1 ฿
v_2 ฿

The transactions where the input script is a signature, and the output script is a verification algorithm are the most common type of transactions and are called *standard transactions*. The address against which the verification is done will be called a *recipient* of this transaction. Currently some miners accept only such transactions. However, there exist other ones that do accept the non-standard (also called *strange*) transactions, one example being a big mining pool[6] called *Eligius* (that mines a new block on average once per hour). We also believe that in the future accepting the general transaction will become a standard. This is important for our applications since our protocols rely heavily on the extended form of transactions.

2.2 Security Model

We use the security model defined in [2]. We assume that the parties are connected by an insecure channel and have access to the BitCoin chain. Let us discuss these two assumptions in detail. The only "trusted component" in the system is the BitCoin chain. We assume that the parties have access to a trusted third party denoted blockchain, whose contents is publicly available. One very important aspect that needs to be addressed are the security properties of the communication channel between the parties and the blockchain. We assume that the channel is authenticated, and in particular each party can access the current contents of the blockchain. We do not assume that this communication is private. This corresponds to a real-life situation when the users simply broadcast their transactions in the network when they want to post it on the blockchain. In other words, the adversary can read every transaction before it appears on the blockchain. We do not assume anything about the order in which transaction are

[6] Mining pools are coalitions of miners that perform their work jointly and share the profits.



Put(**in:** T)
in-script: $\mathsf{sig}_C([Put])$
out-script($body, \sigma_1, \sigma_2$)**:**
$\mathsf{ver}_C(body, \sigma_1) \wedge \mathsf{ver}_S(body, \sigma_2)$
val: d ฿
d ฿
$Fuse$(**in:** Put)
in-script: $\mathsf{sig}_C([Fuse]), \mathsf{sig}_S([Fuse])$
out-script($body, \sigma$)**:** $\mathsf{ver}_C(body, \sigma)$
val: d ฿
tlock: t
d ฿
d ฿

Fig. 1. The graph of transactions for a situation when a user locks d ฿. This is an exemplary situation when the problem of malleability arises. C and S denote the pairs of keys hold respectively by the client and the server. t is a moment of time, when the user can take his deposit back. T denotes an unredeemed transaction with value d ฿, which can be redeemed with key C.

included in the blockchain, the only assumption is that every valid transaction, which was sent to blockchain (that is broadcast in the network) will be eventually included in the blockchain and that there is known a maximum delay between broadcasting a transaction and including it in the blockchain.

In particular the following scenario could take place. The honest user broadcasts transaction T_x, which should redeem transaction T_y already included in the blockchain. The malicious adversary sees the transaction T_x before it is included in the blockchain and broadcasts another transaction $T_{x'}$, which redeems T_y. The transaction $T_{x'}$ is included in the blockchain first, what invalidates transaction T_x. It this scenario we assumed that an adversary is able to construct such transaction $T_{x'}$. However, in the current version of BitCoin, the transaction are "malleable", what means that it is always possible for an adversary to create a valid transaction $T_{x'}$, which is functionally identical to T_x (has the same *body*, but different input scripts) and has a different hash[7]. That is the reason why we need to assume that a minor modification of BitCoin protocol described in Sec. 3 took place. The above scenario may also be a concern in a situation, when an adversary for some reason is authorized to redeem T_y.

Similarly to [2] we assume that the parties have access to a perfect clock and that their internal computation takes no time. The communication between the parties also takes no time, unless the adversary delays it. These assumptions are made to keep the model as simple as possible, and the security of our protocols does not depend on these assumptions. For simplicity we also assume that the transaction fees are zero, but our model and security statements can be easily modified to take into account the non-zero fees.

3 BitCoin Improvement Proposal

3.1 Malleability

As mentioned earlier in the description of the security model (Sec. 2.2), one of the problems with constructing multi-party protocols using BitCoin is the "malleability"

[7] See Sec. 3 for more details.

of transactions. This problem has been noticed before by the BitCoin community[8] as it concerns several BitCoin protocols that use the advanced features of the scripting language.

Malleability of transactions means, that given a valid transaction T it is possible for everyone to construct a different valid transaction T', which is semantically identical to T, but has a different hash. The malleability of transactions comes from the fact, that a hash of a transaction is computed over the whole transaction including its input scripts. On the other hand, signatures are computed only over the body of the transaction, what means that they do not cover the input scripts[9]. Therefore, one can tweak an input script in a way that does not change its functionality (e.g. by adding *push* and *pop* operations[10]) and create a transaction, which is also correct (the signatures are still valid as the input scripts are not signed), and functionally equivalent to the original transaction[11]. Moreover, BitCoin signatures are malleable — having a valid signature on a message it is possible to constructing another valid signature on the same message.

To understand why malleability of transactions may be a problem consider a situation, when a client wants to prove to a server that he is not a spambot by locking (making unspendable for a particular amount of time) some amount of BitCoins[12]. To achieve this, the client should create a transaction such that he can not redeem it on his own. But he has to be sure, that he will eventually get his money back after some time. This could be resolved by using a transaction with a time-lock (see Fig. 1 for a graph of transactions) — the client first creates a transaction Put spending his money, which can be redeemed only by a transaction signed by him and the server (so they can agree to return the deposit to the client at any time). Then he sends the hash of this transaction to the server and the server returns a transaction $Fuse$ with a signature of the server on it — this transaction sends back the deposit to the client after some time. So now the client may broadcast the first transaction, and after some time he may use the $Fuse$ transaction to get back his deposit. This is exactly where the problem of malleability arises: if an adversary sees the transaction Put after it is broadcast, but before it is included in the BlockChain (as the transactions are broadcast in a peer-to-peer network), he can create and broadcast a transaction Put', which is semantically identical to Put, but has a different hash. Then, if Put' is included in the BlockChain first, the original Put becomes invalidated. As a result the $Fuse$ will not be correct (it contains a hash of Put, which never appeared in the BlockChain), so the client may lose his money.

A source of the malleability problem is that a hash of a transaction depends on its input scripts. In some situation this dependence is itself a problem, because we may not

[8] See en.bitcoin.it/wiki/Transaction_Malleability.

[9] The reason is that it is impossible to construct a signature, in such a way, that it is a part of the message being signed.

[10] In this paper we usually treat input scripts as arguments for the corresponding output scripts. In reality, however, they are scripts in BitCoin scripting language, which are supposed to push arguments for an output script on the stack.

[11] Of course a malicious user can change the input scripts significantly, making the transaction invalid. However this behavior (similarly as refusing to broadcast the transaction to the network) does not result in including the modified transaction in the BlockChain. Therefore the honest user can attempt to broadcast his transaction again via other nodes of the network

[12] To read more about such deposits see en.bitcoin.it/wiki/Contracts.

know the input scripts of the transaction T while signing a transaction redeeming T. In next section we present a possible solution for these problems. It requires a small modification of the BitCoin specification. We believe that this modification could be implemented in the future in BitCoin. We discuss why it does not decrease the security of BitCoin.

3.2 Our modification

In the current version of BitCoin protocol, each transaction contains a hash of the transaction it spends. That hash is computed over the *whole* transaction. We propose to compute those hashes over the transaction without its input scripts (i.e. over the *body* of the transaction)[13]. That means that the transaction would have the same hash value regardless of its input scripts.

Obviously with this modification, the malleability is not a problem. An adversary can still tweak the input script of an arbitrary transaction in the network and broadcast its modified version, but the hashes of both transactions — original and modified one — are identical, so it does not make any difference, which of them will be included in the BlockChain.

Additionally, with this modification it is possible to sign a chain of transactions even if we do not know the input scripts of some of them. The only thing, which is necessary to compute signatures are outputs (output scripts and values) and the hashes of the transactions redeemed by the first transaction in the chain. This may be useful in constructing more complex protocols.

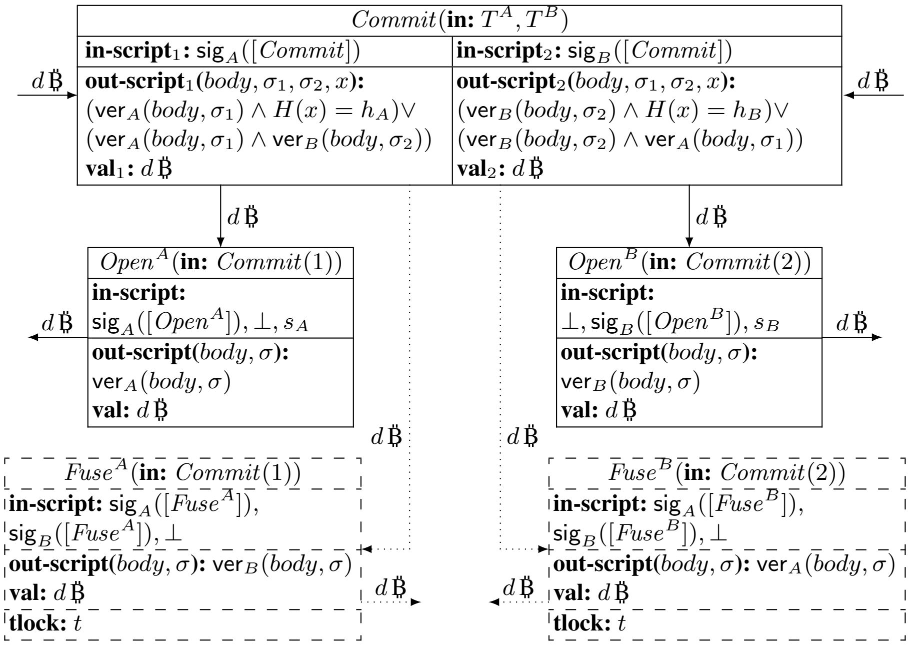
Now consider, what in fact is changed with this modification. The input scripts are used only to show that the transaction is authorized to redeem the other transactions. So two correct transactions which differ only in the input scripts are equivalent — they prove in two different ways that the BitCoin transfer is authorized. It is not possible that the block chain contains two such transactions. That is why the hash still uniquely identifies the redeemed transaction.[14]

4 Simultaneous BitCoin-based timed commitment scheme

In this section we present a modification of the BitCoin-based timed commitment scheme introduced in [2]. Recall that in the original commitment scheme (which was denoted by $\mathsf{CS}(\mathsf{C}, d, t, s)$) the committer C was first committing himself to a secret bit string s and then he was supposed to reveal the secret to the *recipient*. If he did not do it before the time t, then his deposit of d Ƀ was automatically transferred to the recipient.

[13] So they would be computed in the same way the hashes for transactions' signatures are currently being computed.

[14] The only exception are the so-called *generation* transactions, which create new BitCoins and can have arbitrary input scripts (the script is called "coinbase" in this case). However, it is not difficult to ensure that each such transaction has a different hash, by using a new pair of keys for each generation.



$Commit$(**in:** T^A, T^B)
in-script$_1$: $\mathsf{sig}_A([Commit])$
in-script$_2$: $\mathsf{sig}_B([Commit])$
out-script$_1$($body, \sigma_1, \sigma_2, x$): $(\mathsf{ver}_A(body, \sigma_1) \wedge H(x) = h_A) \vee (\mathsf{ver}_A(body, \sigma_1) \wedge \mathsf{ver}_B(body, \sigma_2))$
val$_1$: d ฿
out-script$_2$($body, \sigma_1, \sigma_2, x$): $(\mathsf{ver}_B(body, \sigma_2) \wedge H(x) = h_B) \vee (\mathsf{ver}_B(body, \sigma_2) \wedge \mathsf{ver}_A(body, \sigma_1))$
val$_2$: d ฿
d ฿
$Open^A$(**in:** $Commit(1)$)
in-script: $\mathsf{sig}_A([Open^A]), \perp, s_A$
out-script($body, \sigma$): $\mathsf{ver}_A(body, \sigma)$
val: d ฿
$Open^B$(**in:** $Commit(2)$)
in-script: $\perp, \mathsf{sig}_B([Open^B]), s_B$
out-script($body, \sigma$): $\mathsf{ver}_B(body, \sigma)$
val: d ฿
$Fuse^A$(**in:** $Commit(1)$)
in-script: $\mathsf{sig}_A([Fuse^A]), \mathsf{sig}_B([Fuse^A]), \perp$
out-script($body, \sigma$): $\mathsf{ver}_B(body, \sigma)$
val: d ฿
tlock: t
$Fuse^B$(**in:** $Commit(2)$)
in-script: $\mathsf{sig}_A([Fuse^B]), \mathsf{sig}_B([Fuse^B]), \perp$
out-script($body, \sigma$): $\mathsf{ver}_A(body, \sigma)$
val: d ฿
tlock: t

Pre-condition:

1. A holds the key pair A and B holds the key pair B.
2. A knows the secret s_A, B knows the secret s_B, both players know the hashes $h_A = H(s_A)$ and $h_B = H(s_B)$.
3. There are two unredeemed transactions T^A, T^B of value d ฿, which can be redeemed with the keys A and B respectively.

The $\mathsf{SCS.Commit}(\mathsf{A}, \mathsf{B}, d, t, s_A, s_B)$ **phase**

1. Both players compute the body of the transaction $Commit$ using T^A and T^B as inputs.
2. Both players compute the bodies of the transactions $Fuse^A$ and $Fuse^B$ using appropriate outputs ($Commit(1)$ and $Commit(2)$ respectively) of the $Commit$ transaction. Then, they sign $Fuse^A$ and $Fuse^B$ and exchange the signatures.
3. A signs the transaction $Commit$ and sends the signature to B.
4. B signs the transaction $Commit$ and broadcasts it.
5. Both parties wait until the transaction $Commit$ is included in the BlockChain.
6. If the transaction $Commit$ does not appear on the BlockChain until time $t - \mathsf{max}_{\mathsf{BB}}$, where $\mathsf{max}_{\mathsf{BB}}$ is maximal possible delay between broadcasting the transaction and including it in the BlockChain (what means that B did not perform Step. 4), then A immediately redeems the transaction T^A and quits the protocol. Analogously, if A did not send her signature to B until time $t - \mathsf{max}_{\mathsf{BB}}$, then B redeems the transaction T^B and quits the protocol.

The $\mathsf{SCS.Open}(\mathsf{A}, \mathsf{B}, d, t, s_A, s_B)$ **phase**

7. A and B broadcast the transactions $Open^A$ and $Open^B$ respectively, what reveals the secrets s_A and s_B.
8. If within time t the transaction $Open^A$ does not appear on the BlockChain, then B broadcasts the transaction $Fuse^A$ and gets d ฿. Similarly, if within time t the transaction $Open^B$ does not appear on the BlockChain, then A broadcasts the transaction $Fuse^B$ and gets d ฿.

Fig. 2. The SCS protocol. The scripts' arguments, which are omitted are denoted by $\perp$.

Suppose that there are two parties called Alice and Bob holding secret shares denoted respectively r_A and r_B. Moreover, they are both interested in learning the value $r = r_A \oplus r_B$, where $\oplus$ denotes xor. If both parties made a timed commitment to their secrets, they both would be sure to learn the result r or get a financial compensation. Notice however that they cannot just run two instantiations of the CS scheme, since this would give one party, say Alice, the possibility to abandon committing after Bob has already made a commitment. In that case Bob will be forced to reveal his secret share (or lose his deposit) and Alice would be the only one to learn r. It is pivotal, that these two commitments are made *simultaneously*, i.e. it is not possible that as a result of the protocol one of the parties is committed to her secret and the other one is not. Therefore, to avoid this possibility, we introduce the *simultaneous BitCoin-based timed commitment scheme*, which is a mutual commitment scheme executed between two parties.

The protocol is denoted by $\mathsf{SCS}(\mathsf{A}, \mathsf{B}, d, t, s_A, s_B)$, where A and B are the parties executing the protocol, d is the value of the deposits in Ƀ, t is the timestamp — the parties should open the commitments before that time, and s_A, s_B are the secrets. The protocol consists of two phases: the first one is the *commitment phase* denoted by $\mathsf{SCS.Commit}(\mathsf{A}, \mathsf{B}, d, t, s_A, s_B)$ and the second one is the *opening phase* $\mathsf{SCS.Open}(\mathsf{A}, \mathsf{B}, d, t, s_A, s_B)$ in which each party should open her commitment.

Our protocol uses a commitment scheme based on hashes. A party wanting to commit herself to the string s_A, sends $h_A := H(s_A)$ to the other party, where H is a hash function modeled as a random oracle. Opening the commitment is done by simply revealing s_A. This scheme is secure assuming that the secret has a high min-entropy, what means that the adversary is not able to guess it. If it is not the case, the party may use her secret concatenated with some random string instead. In SCS protocol we assume that both parties already know the hashes of *both* secrets (the reason will become clear in Sec. 5.1).

We construct the SCS protocol assuming the BitCoin modification from section 3 has taken place. The detailed description of the SCS protocol is presented on Fig. 2. The idea behind the protocol is as follows. First the parties use existing transactions T^A and T^B (which can be redeemed by respectively A and B) to construct the transaction $Commit$. The transaction $Commit$ has two outputs — one is used to commit A to s_A and the other one to commit B to s_B. The first output can be claimed by A with revealing her secret or after time t by B. The latter option is technically achieved by signing at the very beginning of the protocol a transaction $Fuse^A$, which redeems $Commit$, can be claimed only by B and has a time-lock t. The second output of $Commit$ is analogical.

The security definition of the SCS protocol is very similar to the security definition of the CS protocol described in [2]. We model the hash function H used in the protocol as a random oracle. We require that the commitment is hiding (before the opening phase the parties have no information about the opponent's secret) as long as the secrets have a high min-entropy and binding (the parties can open the commitment in only one way). We allow a negligible error probabilities in both hiding and biding. The protocol can be interrupted during the commitment phase — in this case the parties do not lose any BitCoins and do not learn the other secret. The only difference between the CS protocol and the SCS protocol is that if the SCS protocol is not interrupted during the commitment phase, then *both* parties are committed. This means that an honest party can be sure that her opponent either reveals the secret by the time t or transfers d ฿ to her. Moreover, it is guaranteed that the party which reveals a secret would get her deposit back. Again, we allow negligible probabilities that the above statements do not hold.

Lemma 1. *The* SCS *scheme from Fig. 2 is a simultaneous BitCoin-based commitment scheme assuming the modification from Sec. 3.*

The proof of the above lemma will appear in the full version of this paper.

5 Two-party computation

The concept of secure two-party computations has already been informally described in the introduction. For the lack of space we do not provide full security definitions of these protocols, and only briefly sketch the constructions. The reader may refer to [9,14] for more on this topic. A common paradigm [15] for constructing secure multiparty protocols is to: (1) create a protocol secure only against passive (also called "semi-honest") adversaries, i.e. adversaries, which honestly perform the protocol, and then (2) "compile" such a protocol to be secure against any type of adversarial behavior.

The problem that such a compiler needs to address is that a malicious party can send a different message than she is supposed to send according to the protocol. One can deal with this problem using the zero-knowledge protocols [17]. This is possible since in every protocol a message which should be sent by a party is determined by (a) the public inputs, (b) the party's private inputs, (c) the messages that she received earlier, and (d) the party's internal randomness. The idea is to attach to each message a zero-knowledge proof that this message was computed correctly. Since a message can depend on private inputs and the internal randomness of the sender (which are not known to the recipient), hence the players commit at the beginning of the protocol to their private inputs and the randomness and later use these commitments in the proof (they actually never open them). Moreover, we need to ensure that the bits used as internal randomness are indeed random, but it can be easily achieved by masking them with the bits chosen by the other party. More details can be found, e.g., in [14].

This compiler works as long as all the parties are interested in completing the protocol. However, the technique described above cannot be used to force a party to send a message if she loses interest in the execution. It is easy to see that in general, there is no "purely cryptographic" way to force a party to execute the protocol until the very end. This may have particularly bad consequences if one of the parties learns the output and, depending on its value either completes the protocol, or halts (preventing the other party from learning the output). This is precisely the problem of the lack of fairness described in the introduction.

In this paper we propose a new way to achieve fairness in two-party computation based on BitCoin deposits. The idea is that before starting the execution of the protocol both parties make a BitCoin deposit of an agreed amount d ฿. If the protocol terminates successfully, both parties gets their deposits back. However, if one of the parties interrupts the protocol after she learned the output, the other party takes both deposits — her own and the opponent's one, so she gains d ฿. We would like to stress that making such a deposit is completely safe — the party making it is guaranteed to get it back if she follows the protocol regardless of the other party's behavior.

5.1 Our FairComputation protocol

Our construction is based on the two-party computation protocol by Goldreich and Vainish [16]. We do not provide the details of this protocol here (for its full description the reader may consult, e.g., [9]). Let us just describe its most relevant part. The property which we take advantage of is that at the end of the protocol's execution the parties hold additive shares of the result of the computation, but none of the parties learned anything about the actual output. This means that the parties holds respectively bit strings r_A and r_B, such that the result of the computation is equal to $r_A \oplus r_B$. In the original protocol, the parties reconstruct the result by revealing their shares. More precisely, each party sends its share to the other party and makes a zero-knowledge proof that it is indeed its share of the result. Of course, one of the parties has to reveal her share first (or at least a part of it) and the other party can quit the protocol at this moment, leaving the honest party with no information about the output[15].

In FairComputation protocol, which we present in this section the parties reconstruct the result in a different and *fair* way. Fairness of that protocol means that at the end of its execution:

- both parties followed the protocol and they both know the result of the computation, or
- one of the parties interrupted the protocol at the beginning and none of the parties learned anything about the result, or
- only a malicious party learned the result, and she payed the other party an agreed amount of BitCoins.

The idea behind FairComputation protocol is as follows. Suppose that the parties are called Alice and Bob. At the very beginning Alice and Bob agree on a value of a deposit equal to d ฿. Then they execute the two-party protocol [16,9] together with the zero-knowledge proofs in order to make it secure against the active adversary. However, they *do not* reconstruct the result. Then, Alice sends a hash h_A of her share concatenated with some random string to Bob and makes a zero-knowledge proof that she indeed computed h_A in that way. Similarly, Bob sends h_B to Alice and makes an analogous proof. Later, the parties execute SCS protocol to simultaneously commit themselves to respectively h_A and h_B. When the commitment is done, the parties reveal their shares. If any of them does not reveal it, the honest party can claim the opponent's deposit. The description of the protocol is presented on Fig. 3.

Lemma 2. *The* FairComputation *protocol from Fig. 3 is a fair two party computation protocol assuming the modification from Sec. 3.*

The proof of the above lemma will appear in the extended version of this paper.

[15] Except of that, what she can learn from her inputs and from the function being computed.

Pre-condition:

1. A holds a key pair A and B holds the key pair B.
2. The parties agree on a function they want to jointly compute and on a value of deposits equal d ₿ each.

The computation phase

The parties execute the two-party protocol of Goldreich and Vainish [16] additionally secured against an active adversary with zero-knowledge proofs, but they do not reconstruct the secret. At the end of the execution A holds r_A and B holds r_B, such that the result of the computation in equal to $r_A \oplus r_B$.

The commitment phase

1. A computes her secret s_A as a concatenation of her share r_A and some random string q_A of length k, where k is a security parameter.
2. A sends $h_A := H(s_A)$ to B and makes a zero-knowledge proof to B that this value is indeed equal to $H(r_A||q_A)$ for some string q_A.
3. Similarly, B computes $s_B := r_B||q_B$ for some random string q_B of length k, sends $h_B := H(s_B)$ to A and makes an analogical proof.
4. The parties execute $\mathsf{SCS.Commit}(\mathsf{A}, \mathsf{B}, d, t, s_A, s_B)$ protocol for some moment of time t in the future.

The opening phase

5. The parties execute $\mathsf{SCS.Open}(\mathsf{A}, \mathsf{B}, d, t, s_A, s_B)$ protocol.
6. If A reveals s_A before time t, then B computes r_A as a prefix of s_A of an appropriate length and computes the result of the computation $r := r_A \oplus r_B$. Otherwise, B earns d ₿ from Fuse^A transaction (See Fig. 2).
7. Similarly, if B reveals s_B before time t, then A computes r_B as a prefix of s_B of an appropriate length and computes the result of the computation $r := r_A \oplus r_B$. Otherwise, A earns d ₿ from Fuse^B transaction (See Fig. 2).

Fig. 3. The FairComputation protocol.

6 Extensions

Payoffs depending on the result of the computation The FairComputation protocol can be easily extended to handle a situation when the result of the computation determines the winner, which will be given some reward (an agreed amount of BitCoin). To achieve this it is enough to add a third output with value equal to the value of the reward to the *Commit* transaction used in the execution of SCS.Commit in Step. 4 of FairComputation protocol. The output script would take as arguments both secrets s_A, s_B and a signature. It would check if both provided secrets are correct ($H(s_A) = h_A \wedge H(s_B) = h_B$), compute r_A and r_B as prefixes of respectively s_A and s_B, compute the actual result ($r := r_A \oplus r_B$), check which party is a winner and verify if the signature is the winner's signature on that transaction (this idea is very similar to the ones used in [4] and [2]).

The idea described above can be further extended to handle a situation, where the reward may be split arbitrarily among the parties depending on the result of the computation, e.g. the result is a fraction between 0 and 1, which determines how big part of the reward will be given to one of the parties (the other party gets the rest of the reward). Suppose that the reward is equal to $1\,\text{\textbaht}$. The parties have to add to *Commit* transaction, not one additional output, but a number of them — one with value $0.5\,\text{\textbaht}$, one with value $0.25\,\text{\textbaht}$, one with value $0.125\,\text{\textbaht}$ and so on [16]. Similarly as earlier, each output script expects both secrets and a signature. It computes the results of the computation, checks, which party should be given the appropriate part of the reward and verifies if the signature is that party's signature.

Non-equal deposits In the description of SCS and FairComputation protocols we assumed that the deposits of both parties are equal. However, the protocols may be straightforwardly modified to handle the situation, when one of the parties makes a bigger deposit than the other one. It is enough to change the values of the appropriate transactions created in these protocols.

Different underlying commitment scheme We have decided to use a commitment scheme based on hashes, because BitCoin scripting language has an instruction for computing SHA-256. It is possible to use an arbitrary commitment scheme instead, but checking if the commitment is opened properly (if the input value hashes to h_i in our case) has to be done in BitCoin script. Most of more sophisticated operations (e.g. multiplication, division, taking modulo of integers, taking a substring, xor) are currently disabled.[17] It is however still feasible to implement an arbitrary function with a limited input size in BitCoin scripting language. It can be achieved by putting the input bit by bit in the input script and computing the function through a binary circuit in the output script (the only necessary operation are accessing i-th value from the stack and binary operations), but such a construction would be in practice rather inefficient.

References

1. M. Abadi and N. Glew. Certified email with a light on-line trusted third party: Design and implementation. WWW '02.
2. M. Andrychowicz, S. Dziembowski, D. Malinowski, and Ł. Mazurek. Secure Multiparty Computations on BitCoin. Cryptology ePrint Archive, 2013. `http://eprint.iacr.org/2013/784`.
3. Giuseppe Ateniese and Cristina Nita-Rotaru. Stateless-recipient certified e-mail system based on verifiable encryption. In *Topics in Cryptology – CT-RSA 2002*.
4. Adam Back and Iddo Bentov. Note on fair coin toss via bitcoin, 2013. `http://www.cs.technion.ac.il/~idddo/cointossBitcoin.pdf`.
5. S. Barber, X. Boyen, E. Shi, and E. Uzun. Bitter to better - how to make Bitcoin a better currency. In *FC*, 2012.
6. Michael Ben-Or, Oded Goldreich, Silvio Micali, and Ronald L. Rivest. A fair protocol for signing contracts. *IEEE Transactions on Information Theory*, 36(1):40–46, 1990.
7. J. Clark and A. Essex. CommitCoin: Carbon dating commitments with Bitcoin - (short paper). In *FC*, 2012.

[16] The number of outputs created this way is limited and not greater than 30 as BitCoin is not infinitely divisible. The smallest amount of BitCoin is called "satoshi" and is equal to $10^{-8}\,\text{\textbaht}$.
[17] They are disabled out of the concern, that some clients may have bugs in their implementation.

8. R. Cleve. Limits on the security of coin flips when half the processors are faulty. STOC, 1986.
9. Ronald Cramer. Introduction to secure computation. In *Lectures on Data Security*, pages 16–62, 1998.
10. Ivan Damgård. Practical and provably secure release of a secret and exchange of signatures. In *EUROCRYPT '93*.
11. Der Spiegel International. Swiss Bank Data: German Tax Officials Launch Nationwide Raids, April 2013.
12. S. Even and Y. Yacobi. Relations among public key signature schemes., 1980. Technical Report 175, Computer Science Dept., Technion, Israel.
13. O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. STOC, 1987.
14. Oded Goldreich. *The Foundations of Cryptography - Volume 2, Basic Applications*. Cambridge University Press, 2004.
15. Oded Goldreich, Silvio Micali, and Avi Wigderson. Proofs that yield nothing but their validity and a methodology of cryptographic protocol design. In *FOCS '86*.
16. Oded Goldreich and Ronen Vainish. How to solve any protocol problem - an efficiency improvement. In *CRYPTO '87*.
17. S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM J. Comput.*, 18(1):186–208, February 1989.
18. S. D. Gordon, C. Hazay, J. Katz, and Y. Lindell. Complete fairness in secure two-party computation. *J. ACM*, 58(6), December 2011.
19. I. Miers, C. Garman, M. Green, and A. D. Rubin. Zerocoin: Anonymous distributed e-cash from bitcoin. *IEEE S&P*, 2012.
20. S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.
21. Birgit Pfitzmann, Matthias Schunter, and Michael Waidner. Optimal efficiency of optimistic contract signing. In *PODC '98*.
22. A. C.-C. Yao. How to generate and exchange secrets. In *FOCS*, 1986.
23. J. Zhou and D. Gollmann. A fair non-repudiation protocol. In *IEEE S&P*, 1996.
24. Jianying Zhou and Dieter Gollmann. Certified electronic mail. In *ESORICS '96*.