Fair Two-Party Computations via Bitcoin Deposits

Marcin Andrychowicz[1], Stefan Dziembowski[1,2], Daniel Malinowski[1], and Łukasz Mazurek[1(✉)]

[1] University of Warsaw, Warszawa, Poland
{marcin.andrychowicz,stefan.dziembowski,daniel.malinowski,
lukasz.mazurek}@crypto.edu.pl
[2] University of Rome La Sapienza, Roma, Italy

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 to 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).

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 [20,29] are protocols that allow a group of mutually distrusting parties to "emulate" a trusted third party 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

This work was supported by the WELCOME/2010-4/2 grant founded within the framework of the EU Innovative Economy (National Cohesion Strategy) Operational Programme.

R. Böhme et al. (Eds.): FC 2014 Workshops, LNCS 8438, pp. 105–121, 2014.
DOI: 10.1007/978-3-662-44774-1_8

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 [10,17].

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 losing 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 paid 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.

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 Sect. 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), 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 [6,12,14,18]. 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 [18] 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 paid 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 [1,4,31] or non-repudiation protocols [30].

We also show how to link the outputs of our protocols to the Bitcoin money in the following sense (for more information see Sect. 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 paid 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 $\pi : \{0,1\}^* \to \{\text{true}, \text{false}\}$. 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 $\pi(m) = \text{true}$, 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 $\pi(m) = \text{true}$ then the output is m and a "forced

transfer of 1 Ƀ 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.

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 create a secure and fair two-player lottery has been independently proposed by Back and Bentov in [5]. Similarly to [2], their protocol makes use of the time-locked transactions, but the purpose they are used for is slightly different. Their protocol uses time-locks to get the deposit back if the protocol is interrupted, while this paper and [2] use time-locks to make a financial compensation to an honest party, whenever the other party misbehaves.

Usage of timed-commitments to achieve fairness in MPC has been already proposed in a number of papers, e.g. [8,19,25], but this line of research uses a completely different approach from ours. It is based on a gradual release of information and if the protocol is interrupted prematurely than both parties can reconstruct the result with a huge computational effort. The fairness of two-party computation has been also studied by Gordon et al. [17], who showed that complete fairness can be achieved for some functions being computed, e.g. Boolean and/or, but not xor. In contrast, our construction works for an arbitrary function.

Improvements to Bitcoin have been suggested in an important work of Barber et al. [16] who study various security aspects of Bitcoin and Miers et al. [15] who propose a Bitcoin system with provable anonymity. The idea to use some concepts from the MPC literature appeared already in Sect. 7.1 of [16] 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 receiver. Also, the main motivation of this work is different: the goal of [16] 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 [9], however, the construction and its applications are

[1] A real-life example of such situation is the recent case when the German tax authorities paid 4 million euro to an anonymous informant for a CD containing information about the German tax evaders with bank accounts in Switzerland [13].

different—the main idea of [9] 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 [26], Example 1), 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 Bitcoin wiki "Contracts page" [26] contains several interesting multi-party protocols, and in some sense our work can be viewed as an effort to extend the set of possible types of contracts. We note that the main features that distinguishes our work from most of them is (1) we do not want to rely on any trusted third parties (like the "mediators") and (2) the focus of our protocols is to protect the input privacy.

The problem of the malleability of the transactions has been noticed before and described in [28]. Malleability is a problem for most of the protocols using time-locks (e.g. [5,27]) and Examples 1, 5, and 7 in [26], but is usually not even mentioned, probably because it is believed that it will be eliminated in the future versions of the Bitcoin protocol. In contrast, our lottery protocol from [2] is not susceptible to the malleability problem. We also note that in our subsequent work [3] we managed to solve the problem of constructing the simultaneous commitment schemes in the standard Bitcoin (without any modifications), at a cost of making the protocol more complicated. Nevertheless, we think that our modification proposal from this paper still makes sense, as it allows to construct simpler simultaneous commitment protocols, and may be useful also in other contexts.

2 A Description of Bitcoin

We assume reader's familiarity with the basic principles of the Bitcoin. 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 (technically an address is a *hash* of pk). 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 $\mathsf{sig}_A(m)$ denotes a signature on a message m computed with sk and $\mathsf{ver}_A(m, \sigma)$ denotes the result (true or false) of the verification of a signature σ on message m with respect to the public key pk.

Each Bitcoin transaction can have multiple inputs and outputs. Inputs of a transaction T_x are listed as triples $(y_1, a_1, \sigma_1), \ldots, (y_n, a_n, \sigma_n)$, where each y_i is a hash of some previous transaction T_{y_i} (our proposal, described in Sect. 3, is to change it, but for a moment let us stick to the current version of the system), 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_i}) and σ_i is called an *input-script* The outputs of a transaction are presented as a list of pairs $(v_1, \pi_1), \ldots, (v_m, \pi_m)$, where each v_i specifies some amount of coins (called the *value of the* i*-th output of* T_x) and π_i is an *output-script*. A transaction can also have a time-lock t, meaning that it is valid only if time t is reached. Hence, altogether transaction's most general form is:

$T_x = ((y_1, a_1, \sigma_1), \ldots, (y_n, a_n, \sigma_n), (v_1, \pi_1), \ldots, (v_m, \pi_m), t)$. The *body of* T_x[2] is equal to T_x without the input-scripts, i.e.: $((y_1, a_1), \ldots, (y_n, a_n), (v_1, \pi_1), \ldots, (v_m, \pi_m), t)$, 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 defined above is valid if for *every* $i = 1, \ldots, n$ we have that $\pi'_i([T_x], \sigma_i)$[3] 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 scripts are written in the Bitcoin scripting language.

We will present the transactions as boxes. The redeeming of transactions will be indicated with arrows (cf. e.g. Fig. 1). 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 *receiver* 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 called *Eligius*.

We use the security model defined in [2]. For the lack of space we only sketch it here. We assume that the parties are connected by an insecure channel and have access to the Bitcoin chain, which is the only "trusted component" in the system. We assume that each party can access the current contents of the block chain, and post messages on it. Let $\mathsf{max}_{\mathsf{BB}}$ be the is maximal possible delay between broadcasting the transaction and including it in the block chain. We do not assume that this communication is private. 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

One of the problems with constructing multi-party protocols using Bitcoin is the "malleability" of transactions. This problem has been noticed before by the Bitcoin community [28] as it concerns several Bitcoin protocols that use the advanced features of the scripting language. Essentially, the problem is that, given a valid transaction T, it is possible for everyone to construct a different valid transaction T', which is functionally equivalent 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

[2] In the original Bitcoin documentation this is called "simplified T_x".

[3] Technically in Bitcoin $[T_x]$ is not directly passed as an argument to π'_i. We adopt this convention to make the exposition clearer.

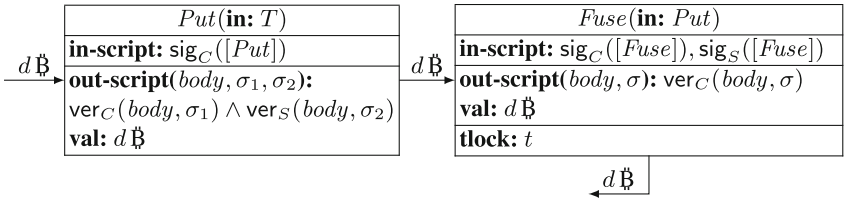


Fig. 1. The graph of transactions for a situation when a user locks d ₿. This is an exemplary situation when the problem of malleability arises. C and S denote the pairs of keys hold respectively by the client and the server. t is a moment of time, when the user can take his deposit back. T denotes an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

transaction, which means that they do not cover the input scripts[4]. Therefore, one can tweak an input script in a way that does not change its functionality (e.g. by adding *push* and *pop* operations[5]) 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, yet its hash is different.

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[6]. 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[7]—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 block chain (as the transactions are broadcast in a peer-to-peer network), he can create and broadcast a transaction

[4] 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.

[5] 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.

[6] To read more about such deposits see [26].

[7] The server signs a transactions *Fuse* without seeing the transaction *Put* and a malicious client could try to send a hash of an existing transaction instead of *Put*. Therefore, the server should use a fresh key every time to prevent itself from being tricked into signing a transaction spending some other transaction of its to the client.

Put', which is functionally equivalent to Put, but has a different hash. Then, if Put' is included in the block chain 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 block chain), 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 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.

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), so they would be computed in the same way the hashes for transactions' signatures are currently being computed. 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 block chain.

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.[8]

4 Simultaneous Bitcoin-Based Timed Commitment Scheme

In this section we present a modification of the Bitcoin-based timed commitment scheme introduced in [2]. To make the paper self-contained we first recall the

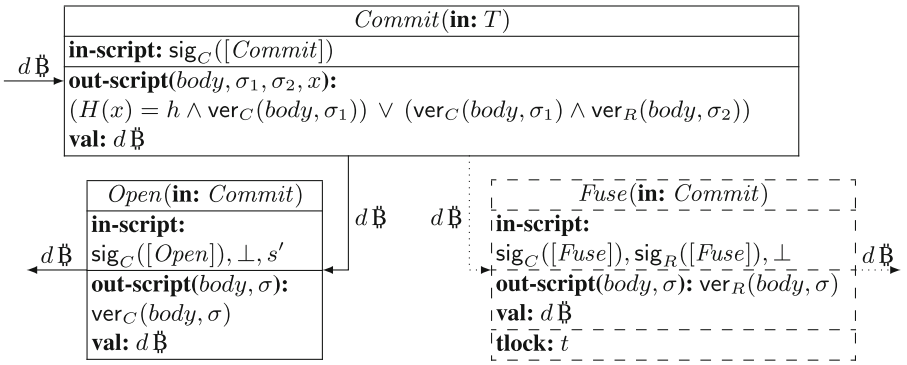
[8] The only exception are the so-called *generation* transactions, which create new bitcoins and can have arbitrary input scripts (the script is called "coinbase" in this case). However, it is not difficult to ensure that each such transaction has a different hash, by using a new pair of keys for each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

original timed-commitment scheme $\mathsf{CS}(\mathsf{C}, d, t, s)$ of [2], and then we describe our modified scheme.

Timed-Commitments of [2]. Recall that a (standard) commitment scheme is a protocol between two parties: a committer C and a receiver R. The protocol contains of two phases. In the first one, called the *commitment phase*, C *commits* to some secret string s by interacting with R. What is important is that after this interaction s should remain secret (this is called the *hiding* property of the scheme). Then comes the *opening phase* in which C *opens* the commitment by interacting again with R, which results in R learning s. What we require is that a cheating C cannot "change his mind", in other words, once the commitment phase is over, there exists at most one value s that R will accept. This property is called *binding*. A simple commitment scheme can be constructed as follows. Let H be a hash function. To commit to a string s (of some fixed length) the committer selects a random string ρ, computes $s' = (s||\rho)$ and sends $H(s')$ to the receiver. If H is modeled as a random oracle, and ρ is sufficiently large (say: linear in the security parameter), then obviously $H(s')$ does not reveal any significant information about s (hence the commitment is hiding). To open the commitment, C sends s' to R. The binding property of this commitment scheme follows from the collision-resistance of H.

Several other commitment schemes have been constructed over the last 2 decades. One inherent problem with all of them is related to the fairness issue in the two-party computation protocols (see Sect. 1). Namely, there is no way to force C to open the commitment. This problem has negative consequences for several applications. Consider, e.g., a simple protocol in which two parties (call them again C and R) want to "flip a coin", i.e., to select a bit $b \leftarrow \{0,1\}$ uniformly at random. A simple protocol of Blum [7] for this problem works as follows: (1) C commits to some random bit $c \leftarrow \{0,1\}$, (2) R selects a random bit $r \leftarrow \{0,1\}$ and sends it to C, (3) C opens his commitment, and the output of the protocol is computed as $b = c \oplus r$. This protocol is obviously secure, informally because the hiding property of the commitment scheme guarantees that R does not know c when he chooses r, and the binding property prevents C from changing c after he learned r. Unfortunately, there is no way to force R to complete Step (3) and to open the commitment. Hence, C he can make the protocol "crash" without producing the output, depending on what the output is.

As a remedy to this problem [2] propose to use Bitcoin in the following way. During the commitment phase the committer has to put aside a "deposit". Assume its value is d ฿, and it comes from an unredeemed standard transaction T, whose receiver is C. The committer gets his money back once he opens the commitment. If he does not open the commitment within some time t then the money can be claimed by the receiver. This is implemented using the Bitcoin scripts and time-locks on top of the hash-based commitment scheme described above. Let C and R be the respective key-pairs of C and R. The transactions used in this implementation are as follows (the scripts' arguments, which are omitted are denoted by $\perp$):



To commit to a secret s the committer first computes $s' = (s||\rho)$ (where ρ is a random string of some fixed length), and sets $h := H(s')$. He then creates the *Commit* transaction and posts it on the block chain. The role of this step is to publish h and to deposit the money. The committer also creates the body of the *Fuse* transaction with time lock set to some time t in the future, and sends it to R together with his signature on it. Hence, the only thing that is missing to obtain the complete *Fuse* transaction is the receiver's signature on the body. This, however, R can compute himself. Hence at the end of the commitment phase R holds a *Fuse* transaction. The purpose of *Fuse* is to allow the receiver to claim the money, if R did not open the commitment within time t.

In the opening phase the committer posts *Open* on the block chain. This has two consequences. Firstly, this reveals s' (and hence s), which is part of the input script. Secondly, it allows the committer to get his money back. Thanks to the way in which the scripts are created, this is actually the only way for him to get his money. If he does not do it by the time t, then R posts *Fuse* on the block chain and gets the committer's deposit.

It is easy to see how this timed-commitment scheme solves the problem of fairness in the coin-flipping protocol described above: if C did not open the commitment scheme on time then he is "punished" financially for this, and R gets a compensation. Unfortunately, this commitment scheme does not solve the fairness problem in general. This is because for the general two-party computation protocols we need something stronger. More precisely, the problem is that this commitment scheme forces the committer to reveal his secret (or to pay a fine), no matter how the other party behaves. To see why it is a problem, imagine two parties, called Alice and Bob holding secrets denoted respectively s_A and s_B. Suppose that the protocol instructs both of them to commit to their secrets and then to reveal them (in fact this is exactly the situation that we have in our two-party scheme in Sect. 5). If they just run two instantiations of the CS scheme, then one party, say Alice, can interrupt the protocol where she is the committer, after Bob has already made a commitment. In that case Bob will be forced to reveal his secret share or lose his deposit. Hence, it is important that both commitment schemes are executed *simultaneously*, i.e. it is not possible that as a result of the protocol one of the parties is committed to her secret and

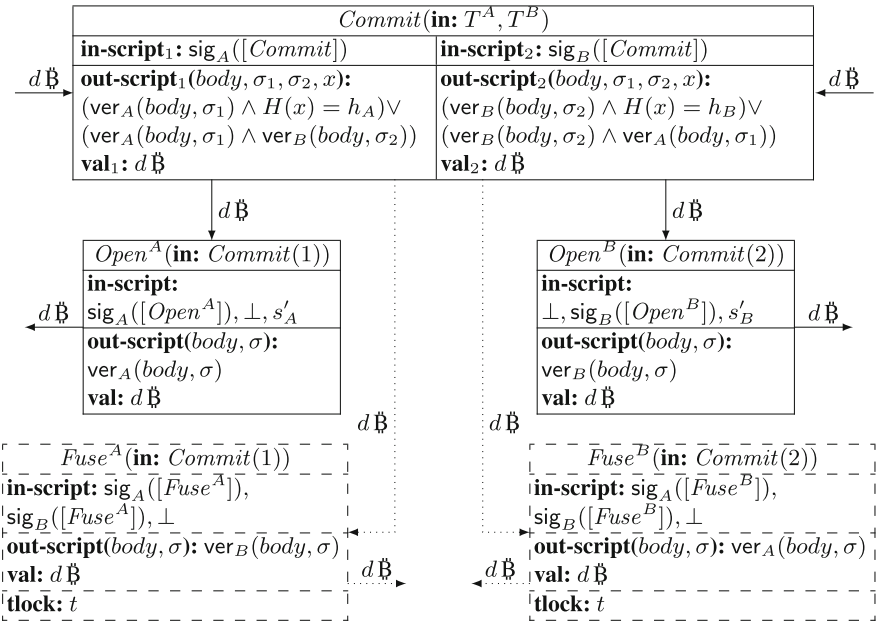
the other one is not. A construction of such a commitment scheme is one of our two main contributions and is presented below.

Simultaneous Bitcoin-Based Timed Commitment Scheme. The protocol is denoted by $\mathsf{SCS}(\mathsf{A}, \mathsf{B}, d, t)$, 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. We assume that A and B are the respective key pairs of A and B and the block chain contains unredeemed transactions T^A and T^B, both of a value d, whose receivers are A and B respectively. The protocol is depicted on Fig. 2. The commitment phase is denoted by $\mathsf{SCS.Commit}(\mathsf{A}, \mathsf{B}, d, t)$ and the opening phase is denoted by $\mathsf{SCS.Open}(\mathsf{A}, \mathsf{B}, d, t)$. Let α be the security parameter.

The security definition of the SCS protocol is very similar to the security definition of the CS protocol described above. We model the hash function H used in the protocol as a random oracle. We require that the commitment is hiding and binding. We allow a negligible (in α) 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 party's 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 Ƀ 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.

We construct the SCS protocol assuming the Bitcoin modification from Sect. 3. The detailed description of the SCS protocol is presented on Fig. 2. In SCS protocol we assume that both parties already know the hashes h_A and h_B of *both* secrets concatenated with some random strings ρ_A and ρ_B (resp.). More precisely: $h_A := H(s_A||\rho_A)$ and $h_B := H(s_B||\rho_B)$, where $\rho_A \leftarrow \{0,1\}^\alpha$ and $\rho_B \leftarrow \{0,1\}^\alpha$. The reason for this will become clear in Sect. 5. The idea behind the protocol is as follows. First the parties use the existing transactions T^A and T^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 analogous. The proof of the following lemma appears in the extended version of this paper.

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



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 $h_B = H(s'_B)$, where $s'_A := (s_A||\rho_A)$, $s'_B := (s_B||\rho_B)$, and $\rho_A, \rho_B \leftarrow \{0,1\}^\alpha$ are random strings known only to A and B respectively.
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 $\mathsf{SCS.Commit}(\mathsf{A}, \mathsf{B}, d, t)$ **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 block chain.
6. If the transaction $Commit$ does not appear on the block chain until time $t - 2\mathsf{max}_{\mathsf{BB}}$, where $\mathsf{max}_{\mathsf{BB}}$ is a maximal possible delay between broadcasting the transaction and including it in the block chain (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 $t - 2\mathsf{max}_{\mathsf{BB}}$, then B redeems the transaction T^B and quits the protocol.

The $\mathsf{SCS.Open}(\mathsf{A}, \mathsf{B}, d, t)$ **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 block chain, then B broadcasts the transaction $Fuse^A$ and gets d₿. Similarly, if within time t the transaction $Open^B$ does not appear on the block chain, then A broadcasts the transaction $Fuse^B$ and gets d₿.

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

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 [11,21] for more on this topic. A common paradigm [22] 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 [24]. 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 receiver), 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 [21].

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 Ƀ. If the protocol terminates successfully, then both parties get 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 Ƀ. 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.

Our construction is based on the two-party computation protocol by Goldreich and Vainish [23]. We do not provide the details of this protocol here (for its full description the reader may consult, e.g. [11]). 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 s_A and s_B, such that the result of the computation is equal to $s_A \oplus s_B$. 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[9].

In FairComp protocol, which we present in this section the parties reconstruct the result in a different and *fair* way. Fairness of that protocol means that one of the following things happened: either (1) at the end of its execution both parties followed the protocol and they both know the result of the computation, or (2) one of the parties interrupted the protocol at the beginning and none of the parties learned anything about the result, or (3) only a malicious party learned the result, and she paid the other party an agreed amount of bitcoins.

The idea behind FairComp 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 Ƀ. Then they execute the two-party protocol [11,23] 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 its share, the honest party can claim the opponent's deposit. The description of the protocol is presented on Fig. 3.

We now have the following lemma whose proof appears in the extended version of this paper.

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

6 Extensions

The result from the previous section can be extended in various ways. It is for example relatively easy to see that the deposits in the SCS and FairComp do not need to be equal for both parties. Another generalization is that (in theory, and very inefficiently) one can use an arbitrary commitment scheme, not necessarily the one based on hashes (the details of this will be provided in the extended version of this paper).

Probably the most interesting extension is to make the payoffs in the FairComp protocol depend on the result of the computation. More precisely, the FairComp

[9] 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 [23] 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 s_A and B holds s_B, such that the result of the computation in equal to $s_A \oplus s_B$.

The commitment phase

1. A computes her secret s'_A as a concatenation of her share s_A and some random string ρ_A of length α, where α 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 $H(s_A||\rho_A)$ for some string ρ_A.
3. Similarly, B computes $s'_B := s_B||\rho_B$ for some random string ρ_B of length α, sends $h_B := H(s'_B)$ to A and makes an analogous proof.
4. The parties execute SCS.Commit(A, B, d, t) protocol for some moment of time t in the future.

The opening phase

5. The parties execute SCS.Open(A, B, d, t) protocol.
6. If A reveals s'_A before time t, then B computes s_A as a prefix of s'_A of an appropriate length and computes the result of the computation $s := s_A \oplus s_B$. Otherwise, B earns d ₿ from $Fuse^A$ transaction (See Fig. 2).
7. Similarly, if B reveals s'_B before time t, then A computes s_B as a prefix of s'_B of an appropriate length and computes the result of the computation $s := s_A \oplus s_B$. Otherwise, A earns d ₿ from $Fuse^B$ transaction (See Fig. 2).

Fig. 3. The FairComp protocol.

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 bitcoins). To achieve this it is enough to add a third output with the value equal to the value of the reward to the *Commit* transaction used in the execution of SCS.Commit in Step. 4 of FairComp 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 s_A and s_B as prefixes of respectively s'_A and s'_B, compute the actual result ($s := s_A \oplus s_B$), 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 [2,5]).

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 ₿. The parties have to add to *Commit* transaction, not one additional output, but

a number of them—one with value 0.5 Ƀ, one with value 0.25 Ƀ, one with value 0.125 Ƀ and so on[10]. 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.

Acknowledgments. We would like to thank the anonymous reviewers for their valuable comments.

References

1. Abadi, M., Glew, N.: Certified email with a light on-line trusted third party: design and implementation. In: WWW '02
2. Andrychowicz, M., Dziembowski, S., Malinowski, D., Mazurek, Ł.: Secure multiparty computations on Bitcoin. Cryptology ePrint Archive (2013). http://eprint.iacr.org/2013/784
3. Andrychowicz, M., Dziembowski, S., Malinowski, D., Mazurek, L.: How to deal with malleability of Bitcoin transactions. CoRR, abs/1312.3230 (2013)
4. Ateniese, G., Nita-Rotaru, C.: Stateless-recipient certified e-mail system based on verifiable encryption. In: Preneel, B. (ed.) CT-RSA 2002. LNCS, vol. 2271, pp. 182–199. Springer, Heidelberg (2002)
5. Back, A., Bentov, I.: Note on fair coin toss via Bitcoin (2013). http://www.cs.technion.ac.il/~idddo/cointossBitcoin.pdf
6. Ben-Or, M., Goldreich, O., Micali, S., Rivest, R.L.: A fair protocol for signing contracts. IEEE Trans. Inf. Theor. **36**(1), 40–46 (1990)
7. Blum, M.: Coin flipping by telephone. In: CRYPTO 1981 (1981)
8. Boneh, D., Naor, M.: Timed commitments. In: Bellare, M. (ed.) CRYPTO 2000. LNCS, vol. 1880, pp. 236–254. Springer, Heidelberg (2000)
9. Clark, J., Essex, A.: CommitCoin: carbon dating commitments with Bitcoin. In: Keromytis, A.D. (ed.) FC 2012. LNCS, vol. 7397, pp. 390–398. Springer, Heidelberg (2012)
10. Cleve, R.: Limits on the security of coin flips when half the processors are faulty. In: STOC '86
11. Cramer, Ronald: Introduction to secure computation. In: Damgård, Ivan Bjerre (ed.) EEF School 1998. LNCS, vol. 1561, p. 16. Springer, Heidelberg (1999)
12. Damgård, I.B.: Practical and provably secure release of a secret and exchange of signatures. In: Helleseth, T. (ed.) EUROCRYPT 1993. LNCS, vol. 765, pp. 200–217. Springer, Heidelberg (1994)
13. Der Spiegel International. Swiss Bank Data: German Tax Officials Launch Nationwide Raids, April 2013
14. Pfitzmann, B., et al.: Optimal efficiency of optimistic contract signing. In: PODC '98
15. Miers, I., et al.: Zerocoin: anonymous distributed e-cash from Bitcoin. IEEE S&P (2012)

[10] The number of outputs created this way is limited and not greater than 30 as a bitcoin is not infinitely divisible. The smallest amount of bitcoins is called "satoshi" and is equal to 10^{-8} Ƀ.

16. Barber, S., Boyen, X., Shi, E., Uzun, E.: Bitter to better—how to make bitcoin a better currency. In: Keromytis, A.D. (ed.) FC 2012. LNCS, vol. 7397, pp. 399–414. Springer, Heidelberg (2012)
17. Gordon, S., et al.: Complete fairness in secure two-party computation. J. ACM **58**(6), 1–37 (2011)
18. Even, S., Yacobi, Y.: Relations among public key signature schemes. Technical report 175, Computer Science Department, Technion, Israel (1980)
19. Garay, J.A., Jakobsson, M.: Timed release of standard digital signatures. In: Blaze, M. (ed.) FC 2002. LNCS, vol. 2357, pp. 168–182. Springer, Heidelberg (2003)
20. Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game. In: STOC 1987 (1987)
21. Goldreich, O.: The Foundations of Cryptography: Basic Applications, vol. 2. Cambridge University Press, Cambridge (2004)
22. Goldreich, O., Micali, S., Wigderson, A: Proofs that yield nothing but their validity and a methodology of cryptographic protocol design. In: FOCS '86
23. Goldreich, O., Vainish, R.: How to solve any protocol problem - an efficiency improvement. In: CRYPTO '87
24. Goldwasser, S., Micali, S., Rackoff, C.: The knowledge complexity of interactive proof systems. SIAM J. Comput. **18**(1), 186–208 (1989)
25. Pinkas, B.: Fair secure two-party computation. In: Biham, E. (ed.) EUROCRYPT 2003. LNCS, vol. 2656, pp. 87–105. Springer, Heidelberg (2003)
26. Bitcoin wiki: Contracts. http://en.bitcoin.it/wiki/Contracts. Accessed 24 Nov 2013
27. Bitcoin wiki: Dominant Assurance Contracts. http://en.bitcoin.it/wiki/Dominant_Assurance_Contracts. Accessed 19 Jan 2014
28. Bitcoin wiki: Transaction malleability. https://en.bitcoin.it/wiki/Transaction_Malleability. Accessed 20 Jan 2014
29. Yao, A.C.-C.: How to generate and exchange secrets. In: FOCS 1986 (1986)
30. Zhou, J., Gollmann, D.: A fair non-repudiation protocol. In: IEEE S&P (1996)
31. Zhou, J., Gollmann, D.: Certified electronic mail. In: Martella, G., Kurth, H., Montolivo, E., Bertino, E. (eds.) ESORICS 1996. LNCS, vol. 1146, pp. 160–171. Springer, Heidelberg (1996)