Secure Multiparty Computations on Bitcoin

Marcin Andrychowicz, Stefan Dziembowski*, Daniel Malinowski, Łukasz Mazurek
University of Warsaw, Poland
{marcin.andrychowicz, stefan.dziembowski, daniel.malinowski, lukasz.mazurek}@crypto.edu.pl
*on leave from *Sapienza* University of Rome

Abstract

Bitcoin is a decentralized digital currency, introduced in 2008, that has recently gained noticeable popularity. Its main features are: (a) it lacks a central authority that controls the transactions, (b) the list of transactions is publicly available, and (c) its syntax allows more advanced transactions than simply transferring the money. The goal of this paper is to show how these properties of Bitcoin can be used in the area of secure multiparty computation protocols (MPCs).

Firstly, we show that the Bitcoin system provides an attractive way to construct a version of "timed commitments", where the committer has to reveal his secret within a certain time frame, or to pay a fine. This, in turn, can be used to obtain fairness in some multiparty protocols. Secondly, we introduce a concept of multiparty protocols that work "directly on Bitcoin". Recall that the standard definition of the MPCs guarantees only that the protocol "emulates the trusted third party". Hence ensuring that the inputs are correct, and the outcome is respected is beyond the scope of the definition. Our observation is that the Bitcoin system can be used to go beyond the standard "emulation-based" definition, by constructing protocols that link their inputs and the outputs with the real Bitcoin transactions.

As an instantiation of this idea we construct protocols for secure multiparty lotteries using the Bitcoin currency, without relying on a trusted authority (one of these protocols uses the Bitcoin-based timed commitments mentioned above). Our protocols guarantee fairness for the honest parties no matter how the loser behaves. For example: if one party interrupts the protocol then her money is transferred to the honest participants. Our protocols are practical (to demonstrate it we performed their transactions in the actual Bitcoin system), and can be used in real life as a replacement for the online gambling sites. We think that this paradigm can have also other applications. We discuss some of them.

Keywords

bitcoin; multiparty; lottery;

I. Introduction

Secure multiparty computation (MPC) protocols, originating from the seminal works of Yao [41] and Goldreich et al. [29], allow a group of mutually distrusting parties to compute a joint function f on their private inputs. Typically, the security of such protocols is defined with respect to the *ideal model* where f is computed by a trusted party T_f. More precisely: it is required that during the execution of a protocol the parties cannot learn more information about the inputs of the other participants than they would learn if f was computed by T_f who: (a) receives the inputs from the parties, (b) computes f, and (c) sends the output back to the parties. Moreover, even if some parties misbehave and do not follow the protocol, they should not be able to influence the output of the honest parties more than they could in the ideal model by modifying their own inputs.

As an illustration of the practical meaning of such security definition consider the case when there are only two participants, called Alice and Bob, and the function that they compute is a conjunction $f_\wedge(a,b) = a \wedge b$, where $a, b \in \{0,1\}$ are Boolean variables denoting the inputs of Alice and Bob, respectively. This is sometimes called the *marriage proposal problem*, since one can interpret the input of each party as a declaration if she/he wants to marry the other one. More precisely: suppose $a = 1$ if and only if Alice wants to marry Bob, and $b = 1$ if and only if Bob wants to marry Alice. In this case $f_\wedge(a,b) = 1$ if and only if *both* parties want to marry each other, and hence, if, e.g., $b = 0$ then Bob has no information about Alice's input. Therefore the privacy of Alice is protected.

One can also consider randomized functions f, the simplest example being the *coin tossing problem* [9] where the computed function $f_{\mathrm{rnd}} : \{\perp\} \times \{\perp\} \to \{0,1\}$ takes no inputs, and outputs a uniformly random bit. Yet another generalization are the so-called *reactive functionalities* where the trusted party T maintains a state and the parties can interact with T in several rounds. One example of such a functionality is the *mental poker* [39] where T simulates a card game, i.e. she first deals a deck of cards and then ensures that the players play the game according to the rules.

It was shown in [29] that for any efficiently-computable function f (or, more general, any reactive functionality) there exists an efficient protocol that securely computes it, assuming the existence of the trapdoor-permutations. If the minority of the parties is malicious (i.e. does not follow the protocol) then the protocol always terminates, and the output is known to each honest participant. If not, then the malicious parties can terminate the protocol after learning the output, preventing the honest parties from learning it. It turns out [20] that in general this problem, called the lack of *fairness*, is unavoidable, although there has been some effort to overcome this impossibility result by relaxing the security requirements [30], [15], [6], [35]. Note that in case of the two-player protocols it makes no sense to assume that the majority of the players is honest, as this would simply

mean that none of the players is malicious. Hence, the two-party protocols in general do not provide complete fairness (unless the security definition is weakened).

Since the introduction of the MPCs there has been a significant effort to make these protocols efficient [32], [7], [22] and sometimes even to use them in the real-life applications such as, e.g., the online auctions [10]. On the other hand, perhaps surprisingly, the MPCs have not been used in many other areas where seemingly they would fit perfectly. One prominent example is the internet gambling: it may be intriguing that currently gambling over the internet is done almost entirely with the help of the web-sites that play the roles of the "trusted parties", instead of using the coin flipping or the mental poker protocols. This situation is clearly unsatisfactory from the security point of view, especially since in the past there were cases when the operators of these sites abused their privileged position for their own financial gain (see e.g. [36]). Hence, it may look like the multiparty techniques that eliminate the need for a trusted party would be a perfect replacement for the traditional gambling sites (an additional benefit would be a reduced cost of gambling, since the gambling sites typically charge fees for their service).

In our opinion there are at least two main reasons why the MPCs are not used for online gambling. The first reason is that the multiparty protocols do not provide fairness in case there is no honest majority among the participants. Consider for example a simple two-party lottery based on the coin-tossing protocol: the parties first compute a random bit b, if $b = 0$ then Alice pays \$1 to Bob, if $b = 1$ then Bob pays \$1 to Alice, and if the protocol did not terminate correctly then the parties do not pay any money to each other. In this case a malicious party, say Alice, could prevent Bob from learning the output if it is equal to 0, making 1 the only possible output of a protocol. Since this easily generalizes to the multiparty case, it is clear that the gambling protocol would work only if the majority is honest, which is not a realistic assumption in the fully distributed internet environment (there are many reasons for this, one of them being the *sybil* attacks [23] where one malicious party creates and controls several "fake" identities, easily obtaining the "majority" among the participants).

The second reason is even more fundamental, as it comes directly from the inherent limitations of the MPC security definition, namely: such protocols do not provide security beyond the trusted-party emulation. This drawback of the MPCs is rarely mentioned in the literature as it seems obvious that in most of the real-life applications cryptography cannot be "responsible" for controlling that the users provide the "real" input to the protocol and that they respect the output. Consider for example the marriage proposal problem: it is clear that even in the ideal model there is no technological way to ensure that the users honestly provide their input to the trusted party, i.e. nothing prevents one party, say Bob, to lie about his feelings, and to set $b = 1$ in order to learn Alice's input a. Similarly, forcing both parties to respect the outcome of the protocol and indeed marry cannot be guaranteed in a cryptographic way. This problem is especially important in the gambling applications: even in the simplest "two-party lottery" example described above, there exists no cryptographic method to force the loser to transfer the money to the winner.

One pragmatic solution to this problem, both in the digital and the non-digital world is to use the concept of "reputation": a party caught on cheating (i.e. providing the wrong input or not respecting the outcome of the game) damages her reputation and next time may have trouble finding another party willing to gamble with her. Reputation systems have been constructed and analyzed in several papers (see, e.g. [37] for an overview), however they seem too cumbersome to use in many applications, one reason being that it is unclear how to define the reputation of the new users in the scenarios when the users are allowed to pick new names whenever they want [26].

Another option is to exploit the fact that the financial transactions are done electronically, and hence one could try to "incorporate" the final transaction (transferring \$1 from the loser to the winner) into the protocol, in such a way that the parties learn who won the game only when the transaction has already been performed. It is unfortunately not obvious how to do it within the framework of the existing electronic cash systems. Obviously, since the parties do not trust each other, we cannot accept solutions where the winning party learns e.g. the credit card number, or the account password of the loser. One possible solution would be to design a multiparty protocol that simulates, in a secure way, a simultaneous access to all the online accounts of the participants and executes a wire transfers in their name.[1] Even if theoretically possible, this solution is clearly very hard to implement in real life, especially since the protocol would need to be adapted to several banks used by the players (and would need to be updated whenever they change). The same problems occur obviously also if above we replace the "bank" with some other financial service (like PayPal). One could consider using Chaum's Ecash [17], or one of its variants [18], [16]. Unfortunately, none of these systems got widely adopted in real-life. Moreover, they are also bank-dependent, meaning that even if they get popular, one would face a challenge of designing a protocol that simulates the interaction of a real user with a bank, and make it work for several different banks.

We therefore turn our attention to Bitcoin, which is a decentralized digital currency introduced in 2008 by Satoshi Nakamoto[2] [34]. Bitcoin has recently gained a noticeable popularity (its current market capitalization is over \$5 billion) mostly due to its distributed nature and the lack of a central authority that controls the transactions. Because of that it is infeasible for anyone to take control over the system, create large amounts of coins (to generate inflation), or shut it down. The money is transferred directly between two parties — they do not have to trust anyone else and transaction fees are zero or very small. Another advantage is pseudonymity[3] — the users are only identified by their public keys that can be easily created, and hence it is hard

[1]Note that this would require, in particular, "simulating" the web-browser and the SSL sessions, since each individual user should not learn the contents of the communication between the "protocol" and his bank, as otherwise he could interrupt the communication whenever he realizes that the "protocol" ordered a wire transfer from his account. Moreover, one would need to assume that the transactions cannot be cancelled once they were ordered.

[2]This name is widely believed to be a pseudonym.

[3]A very interesting modification of Bitcoin that provides real cryptographic anonymity has been recently proposed in [33].

to link the real person with the virtual party spending the money. However, since all the transactions and the connections between them are publicly known there are several ways to extract some information about Bitcoin users from the block chain, see e.g. [38].

In Section II we describe the main design principles of Bitcoin, focusing only on the most relevant parts of this system. A more detailed description can be found in Nakamoto's original paper [34], the Bitcoin wiki webpage en.bitcoin.it (sections particularly relevant to our work are: "Transactions" and "Contracts"), or other papers on Bitcoin [33], [19], [5], [38]. In the sequel "฿" denotes the Bitcoin currency symbol.

A. Our contribution

We study how to do "MPCs on Bitcoin". First of all, we show that the Bitcoin system provides an attractive way to construct a version of "timed commitments" [11], [27], where the committer has to reveal his secret within a certain time frame, or to pay a fine. This, in turn, can be used to obtain fairness in certain multiparty protocols. Hence it can be viewed as an "application of Bitcoin to the MPCs".

What is probably more interesting is our second idea, which in some inverts the previous one by showing an "application of the MPCs to Bitcoin", namely we introduce a concept of multiparty protocols that work directly on Bitcoin. As explained above, the standard definition of the MPCs guarantees only that the protocol "emulates the trusted third party". Hence ensuring that the inputs are correct, and the outcome is respected is beyond the scope of the definition. Our observation is that the Bitcoin system can be used to go beyond the standard "emulation-based" definition, by constructing protocols that link the inputs and the outputs with the real Bitcoin transactions. This is possible since the Bitcoin lacks a central authority, the list of transactions is public, and its syntax allows more advanced transactions than simply transferring the money.

As an instantiation of this idea we construct protocols for secure multiparty lotteries using the Bitcoin currency, without relying on a trusted authority. By "lottery" we mean a protocol in which a group of parties initially invests some money, and at the end one of them, chosen randomly, gets all the invested money (called the *pot*). Our protocols can work in purely peer-to-peer environment, and can be executed between players that are anonymous and do not trust each other. Our constructions come with a very strong security guarantee: no matter how the dishonest parties behave, the honest parties will never get cheated. More precisely, each honest party can be sure that, once the game starts, it will always terminate and will be fair.

Our two main constructions are as follows. The first protocol (Section IV) can be executed between any number of parties. Its security is obtained via the so-called *deposits*: each user is required to initially put aside a certain amount of money, which will be paid back to her once she completes the protocol honestly. Otherwise the deposit is given to the other parties and "compensates" them the fact that the game terminated prematurely. This protocol uses the timed commitment scheme described above. A certain drawback of this protocol is that the deposits need to be relatively large, especially if the protocol is executed among larger groups of players. More precisely to achieve security the deposit of each player needs to be $N(N-1)$ times the size of the bet (observe that for the two-party case it simply means that the deposit is twice the size of the bet).

We also describe (in Section V) a protocol that does not require the use of deposits at all. This comes at a price: the protocol works only for two parties, and its security relies on an additional assumption (see Section V for more details).

The only cost that the participants need to pay in our protocols are the Bitcoin transaction fees. The typical Bitcoin transactions are currently free. However, the participants of our protocols need to make a small number of non-standard transactions (so-called "strange transactions", see Section II), for which there is usually some small fee (currently around $0.00005฿ \approx \$0.03$). To keep the exposition simple we initially present our results assuming that the fees are zero, and later, in Section VI, argue how to extend the definitions and security statements to take into account also the non-zero fees. For the sake of simplicity we also assume that the bets in the lotteries are equal to $1฿$. It should be straightforward to see how to generalize our protocols to other values of the bets.

Our constructions are based on the coin-tossing protocol of Blum [9]. We managed to adapt this protocol to our model, without the need to modify the current Bitcoin system. We do not use any generic methods like the MPC or zero-knowledge compilers, and hence the protocols are very efficient. The only cryptographic primitives that we use are the commitment schemes, implemented using the hash functions (which are standard Bitcoin primitives). Our protocols rely strongly on the advanced features of the Bitcoin (in particular: the so-called "transaction scripts", and "time-locks"). Because of the lack of space we only sketch the formal security definitions. The security proofs will appear in an extended version of this paper. We executed our transactions on the real Bitcoin. We provide a description of these transactions and a reference to them in the Bitcoin chain.

B. Applications and future work

Although, as argued in Section I-C below, it may actually make economic sense to use our protocols in practice, we view gambling mostly as a motivating example for introducing a concept that can be called "MPCs on Bitcoin", and which will hopefully have other applications. One (rather theoretical) example of such application is the "millionaires problem" where Alice and Bob want to establish who is richer.[4] It is easy to see that Alice and Bob can (inefficiently) determine who has more coins

[4]The formal definition is as follows: let $a, b \in \mathbf{N}$ denote the amount of coins that Alice and Bob respectively own. In this case the parties compute the function $f_{\text{mill}} : \mathbf{N} \times \mathbf{N} \to \{A, B\}$ defined as: $f_{\text{mill}}(a, b) = A$ if and only if $a \geq b$ and $f_{\text{mill}}(a, b) = B$ otherwise.

by applying the generic MPC and zero-knowledge techniques. This is possible since the only inputs that are needed are (a) the contents of the Bitcoin ledger (more precisely: its subset consisting of the non-redeemed transactions), which is public, and (b) Alice's and Bob's private keys used as their respective private inputs (see Section II for the definitions of these terms). Obviously, using this protocol makes sense only if, for some reason, each party is interested in proving that she is the richer one. This is because every participant can easily pretend to be poorer than she really is and "hide" his money by transferring it to some other address (that he also controls). Since we do not think that this protocol is particularly relevant to practical applications, we do not describe it in detail here. Let us only observe that, interestingly, this protocol is in some sense dual to the coin-tossing protocol, as it uses the Bitcoin system to verify the correctness of the inputs, instead guaranteeing that the outcome is respected (as it is the case with the coin-tossing)[5].

We think that analyzing what functionalities can be computed this way (taking into account the problem of the participants "pretending to be poorer than they really are") may be an interesting research direction. Other possible future research directions are: constructing protocols secure against "malleability attacks" and "eavesdropping attacks" (see Sec. V for more details) that do not require the deposits, providing a more formal framework to analyze the deposit-based technique (this can probably be done using the tools from the "rational cryptography" literature [31], [1], [28]).

C. Economic analysis

Besides of being conceptually interesting, we think that our protocols can have direct practical applications in the online gambling, which is a significant market: it is estimated that there are currently 1,700 gambling sites worldwide handling bets worth over $4 billion per year [25]. Some of these sites are using Bitcoin. The largest of them, *SatoshiDice*, has been recently sold for over $12 million [14]. All of the popular sites charge a fee for their service, called the *house edge* (on top of the Bitcoin transaction fees). Currently, the honesty of these sites can be verified only ex post facto: they commit to their randomness before the game starts and later prove to the public that they did not cheat. Hence, nothing prevents them from cheating and then disappearing from the market (using the MPC terminology: such protocols provide security only against the "covert adversaries" [40]). Of course, this means that the users need to continually monitor the behavior of the gambling sites in order to identify the honest ones. This system, called the "mathematically provable fairness" is described in a recent article [13], where it is advised to look on a particular page, called *Mem's Bitcoin Gambling List*, to check the gambling sites' reputation. This simple reputation system can of course be attacked in various ways. Moreover, one can expect that the sites with more established reputation will have a higher house edge, and indeed the SatoshiDice site charges more than the other, less well-known, sites. Currently SatoshiDice house edge is around 2% [13].

Compared to the gambling sites, our protocols have the following advantage. First of all, the security guarantee is stronger, as it does not depend on the honesty of any trusted party. Secondly, in our protocols there is obviously no "house edge". On a negative side, the Bitcoin transaction fees can be slightly larger in our case than in the case of the gambling sites (since we have more transactions, and some of them are "strange"). At the moment of writing this paper, using our solution is cheaper than using SatoshiDice for bets larger than, say, $5, but of course whether our protocols become really widely used in practice depends on several factors that are hard to predict, like the value of the fees for the "strange transactions".

We also note that, although our initial motivation was the peer-to-peer lottery, it can actually make a lot of sense for the online gambling services to use our solutions, especially the two-party protocol. Of course the business model of such services makes sense only if there is non-zero house edge. This is not a problem since our protocols can be easily used in lotteries where the expected payoff is positive for one party (in this case: the gambling service) and it is negative for the other one (the client). Such "provably guaranteed fairness" can be actually a good selling line for some of these services.

D. Previous, concurrent and subsequent work

Some of the related work has been already described in the earlier sections. Previous papers on Bitcoin analyze the Bitcoin transaction graph [38], or suggest improvements of the current version of the system. This includes important work of Barber et al. [5] who study various security aspects of Bitcoin and Miers et al. [33] who propose a Bitcoin system with provable anonymity. Our paper does not belong to this category, and in particular our solutions are fully compatible with the current version of Bitcoin (except of the ,,malleability" and ,,eavesdropping" problem concerning the last protocol, Section V).

Usage of Bitcoin to create a secure and fair two-player lottery has been independently proposed by Adam Back and Iddo Bentov in [4]. Similarly to our solution, 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 uses time-locks to make a financial compensation to an honest party, whenever the other party misbehaves. Additionally, protocol from [4] is not resilient to the malleability attacks, while our main schemes (Section III and IV) are.

Another work relevant to ours is 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, which are similar to these from [4]. Also, the main motivation of this work is different: the goal of [5] is to fix an

[5]The reader may be tempted to think that a similar protocol could be used with the eCash [17]. This is not the case, as in eCash there is no method of proving in zero-knowledge that the money has not been spent (since the list of transactions is not public).

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.

Commitments in the context of the Bitcoin were also considered in [19], however, the construction and its applications are different — the main idea of [19] 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"[6]), 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.

In the subsequent work [2], [3] we show how to extend the ideas from this paper in order to construct a fair two-party protocol for any functionality. Similar ideas were developed independently by Iddo Bentov and Ranjit Kumaresan [8].

E. Acknowledgments

We would like to thank Iddo Bentov and Ranjit Kumaresan for fruitful discussions and for pointing out an error in a previous version of our lottery (see footnote 18 on page 12).

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.

II. A short description of Bitcoin

Bitcoin works as a peer-to-peer network in which the participants jointly emulate the central server that controls the correctness of transactions. In this sense it is similar to the concept of the multiparty computation protocols. Recall that, as described above, a fundamental problem with the traditional MPCs is that they cannot provide fairness if there is no honest majority among the participants, which is particularly difficult to guarantee in the peer-to-peer networks where the sybil attacks are possible. The Bitcoin system overcomes this problem in the following way: the honest majority is defined in terms of the "majority of computing power". In other words: in order to break the system, the adversary needs to control machines whose total computing power is comparable with the combined computing power of all the other participants of the protocol. Hence, e.g., the sybil attack does not work, as creating a lot of fake identities in the network does not help the adversary. In a moment we will explain how this is implemented, but let us first discuss the functionality of the trusted party that is emulated by the users.

One of the main problems with the digital currency is the potential double spending: if coins are just strings of bits then the owner of a coin can spend it multiple times. Clearly this risk could be avoided if the users had access to a trusted ledger with the list of all the transactions. In this case a transaction would be considered valid only if it is posted on the board. For example suppose the transactions are of a form: "user X transfers to user Y the money that he got in some previous transaction T_y", signed by the user X. In this case each user can verify if money from transaction T_y has not been already spent by X. The functionality of the trusted party emulated by the Bitcoin network does precisely this: it maintains a full list of transactions that happened in the system. The format of the Bitcoin transactions is in fact more complex than in the example above. Since it is of a special interest for us, we describe it in more detail in Section II-A.

The Bitcoin ledger is implemented using the concept of the Proofs of Work (PoWs) [24] in the following clever way. The users maintain a chain of *blocks*. The first block B_0, called the *genesis block*, was generated by the designers of the system in January 2009. Each new block B_i contains a list T_i of new transactions, the cryptographic hash of the previous block $H(B_{i-1})$, and some random salt R. The key point is that not every R works for given T_i and $H(B_{i-1})$. In fact, the system is designed in such a way that it is moderately hard to find a valid R. Technically it is done be requiring that the binary representation of the hash of $(T_i||H(B_{i-1})||R)$ starts with a certain number m of zeros (the procedure of extending the chain is called *mining*, and the machines performing it are called *miners*). The hardness of finding the right R depends of course on m, and this parameter is periodically adjusted to the current computing power of the participants in such a way that the extension happens an average each 10 minutes. The system contains an incentive to work on finding the new blocks. We will not go into the details of this, but let us only say that one of the side-effects of this incentive system is the creation of new coins[7].

The idea of the block chain is that the longest chain C is accepted as the proper one. If some transaction is contained in a block B_i and there are several new blocks on top of it, then it is infeasible for an adversary with less than a half of the total computating power of the Bitcoin network to revert it — he would have to mine a new chain C' bifurcating from C at block B_{i-1} (or earlier), and C' would have to be longer than C. The difficulty of that grows exponentially with number of new blocks on top of B_i. In practice the transactions need 10 to 20 minutes (i.e. 1-2 new blocks) for reasonably strong confirmation and 60 minutes (6 blocks) for almost absolute certainty that they are irreversible.

To sum up, when a user wants to pay somebody in bitcoins, he creates a transaction and broadcasts it to other nodes in the network. They validate this transaction, send it further and add it to the block they are mining. When some node solves the mining problem, it broadcasts its block to the network. Nodes obtain a new block, validate transactions in it and its hash and accept it by mining on top of it. Presence of the transaction in the block is a confirmation of this transaction, but some users may choose to wait for several blocks on top of it to get more assurance.

[6] Accessed on 13.11.2013.

[7] The number of coins that are created in the system is however limited, and therefore Bitcoin is expected to have no inflation.

A. The Bitcoin transactions

The Bitcoin currency system consists of *addresses* and *transactions* between them. An address is simply a public key pk.[8] Normally every such a key has a corresponding private key sk known only to one user. The private key is used for signing the transactions, and the public key is used for verifying the signatures. Each user of the system needs to know at least one private key of some address, but this is simple to achieve, since the pairs (sk, pk) can be easily generated offline. We will frequently denote key pairs using the capital letters (e.g. A), and refer to the private key and the public key of A by: $A.sk$ and $A.pk$, respectively (hence: $A = (A.sk, A.pk)$). We will also use the following convention: if $A = (A.sk, A.pk)$ then let $\mathsf{sig}_A(m)$ denote a signature on a message m computed with $A.sk$ and let $\mathsf{ver}_A(m, \sigma)$ denote the result (true or false) of the verification of the signature σ on the message m with respect to the public key $A.pk$.

1) Simplified version: We first describe a simplified version of the system and then show how to extend it to obtain the description of the real Bitcoin. We do not describe how the coins are created as it is not relevant to this paper. Let $A = (A.sk, A.pk)$ be a key pair. In our simplified view a transaction describing the fact that an amount v (called the *value* of a transaction) is transferred from an address $A.pk$ to an address $B.pk$ has the following form

$$T_x = (y, B.pk, v, \mathsf{sig}_A(y, B.pk, v)),$$

where y is an index of a previous transaction T_y. We say that $B.pk$ is the recipient of T_x, and that the transaction T_y is an *input* of the transaction T_x, or that it is *redeemed* by this transaction (or redeemed by the address $B.pk$). More precisely, the meaning of T_x is that the amount v of money transferred to $A.pk$ in transaction T_y is transferred further to $B.pk$. The transaction is valid only if (1) $A.pk$ was a recipient of the transaction T_y, (2) the value of T_y was at least v (the difference between v and the value of T_y is called the *transaction fee*), (3) the transaction T_y has not been redeemed earlier, and (4) the signature of A is correct. Clearly all of these conditions can be verified publicly.

The first important generalization of this simplified system is that a transaction can have several "inputs" meaning that it can accumulate money from several past transactions $T_{y_1}, \ldots, T_{y_\ell}$. Let $A_1, \ldots, A_\ell$ be the respective key pairs of the recipients of those transactions. Then a multiple-input transaction has the following form:

$$\begin{aligned} T_x = (y_1, \ldots, y_\ell, B.pk, v, \mathsf{sig}_{A_1}(y_1, B.pk, v), \ldots, \\ \mathsf{sig}_{A_\ell}(y_\ell, B.pk, v)), \end{aligned}$$

and the result of it is that $B.pk$ gets the amount v, provided it is at most equal to the sum of the values of transactions $T_{y_1}, \ldots, T_{y_\ell}$. This happens only if *none* of these transactions has been redeemed before, and *all* the signatures are valid. Moreover, each transaction can have a *lock-time* t that tells at what time the transaction becomes final (t can refer either to a block index or to the real physical time). In this case we have:

$$\begin{aligned} T_x = (y_1, \ldots, y_\ell, B.pk, v, t, \mathsf{sig}_{A_1}(y_1, B.pk, v, t), \ldots, \\ \mathsf{sig}_{A_\ell}(y_\ell, B.pk, v, t)). \end{aligned}$$

Such a transaction becomes valid only if time t is reached and if none of the transactions $T_{y_1}, \ldots, T_{y_\ell}$ has been redeemed by that time (otherwise it is discarded). Each transaction can also have several outputs, which is a way to divide money between several users or get a change. We ignore this fact in our description since we will not use it in our protocols.

2) A more detailed version: The real Bitcoin system is significantly more sophisticated than what is described above. First of all, there are some syntactic differences, the most important for us being that each transaction T_x is identified not by its index, but by its hash $H(T_x)$. Hence, from now on we will assume that $x = H(T_x)$.

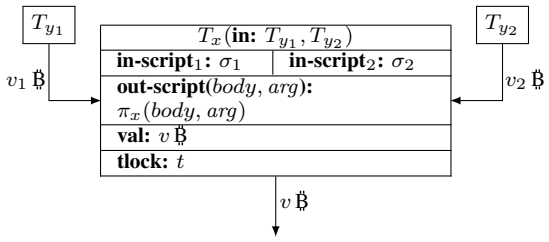
The main difference is, however, that in the real Bitcoin the users have much more flexibility in defining the condition on how the transaction T_x can be redeemed. Consider for a moment the simplest transactions where there is just one input and no time-locks. Recall that in the simplified system described above, in order to redeem a transaction the recipient $A.pk$ had to produce another transaction T_x signed with his private key $A.sk$. In the real Bitcoin this is generalized as follows: each transaction T_y comes with a description of a function (*output-script*) π_y whose output is Boolean. The transaction T_x redeeming the transaction T_y is valid if π_y evaluates to true on input T_x. Of course, one example of π_y is a function that treats T_x as a pair (a message m_x, a signature σ_x), and checks if σ_x is a valid signature on m_x with respect to the public key $A.pk$. However, much more general functions π_y are possible. Going further into details, a transaction looks as follows: $T_x = (y, \pi_x, v, \sigma_x)$, where $[T_x] = (y, \pi_x, v)$ is called the *body*[9] of T_x and σ_x is a "witness" that is used to make the script π_y evaluate to true on T_x (in the simplest case σ_x is a signature on $[T_x]$). 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 operations on numbers, operations on stack, if-then-else statements and some cryptographic functions like calculating hash function or verifying a signature. The generalization to the multiple-input transactions with time-locks is straightforward: a transaction has a form:

$$T_x = (y_1, \ldots, y_\ell, \pi_x, v, t, \sigma_1, \ldots, \sigma_\ell),$$

[8] 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.

[9] In the original Bitcoin documentation this is called "simplified T_x". We have chosen to rename it to "body" since we find the original terminology slightly misleading.

where the body $[T_x]$ is equal to $(y_1, \ldots, y_\ell, \pi_x, v, t)$, and it is valid if (1) time t is reached, (2) *every* $\pi_i([T_x], \sigma_i)$ evaluates to true, where each π_i is the output script of the transaction T_{y_i}, and (3) none of these transactions has been redeemed before. We will present the transactions as boxes. The redeeming of transactions will be indicated with arrows (the arrows will be labelled with the transaction values). For example a transaction $T_x = (y_1, y_2, \pi_x, v, t, \sigma_1, \sigma_2)$ will be represented as:



The transactions where the input script is a signature, and the output script is a verification algorithm are the most common type of transactions. We will call them *standard transactions*, and the address against which the verification is done will be called the *recipient* of a 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[10] called *Eligius* (that mines a new block on average once per hour). We also believe that in the future accepting the general transactions will become standard, maybe at a cost of a slightly increased fee. This is important for our applications since our protocols rely heavily on the extended form of transactions.

B. Security Model

To reason formally about the security we need to describe the attack model that corresponds to the current Bitcoin system. 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. First, recall that our protocol should allow any pair of users on the internet to engage in a protocol. Hence, we cannot assume that there is any secure connection between the parties (as this would require that they can verify their identity, which obviously is impossible in general), and therefore any type of a man-in-the middle attack is possible.

The only "trusted component" in the system is the Bitcoin chain. For the sake of simplicity in our model we will ignore the implementation details of it, and simply assume that the parties have access to a trusted third party denoted Ledger, 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 Ledger. Firstly, it is completely reasonable to assume that the parties can verify Ledger's authenticity. In other words: each party can access the current contents of the Ledger. In particular, the users can post transactions on the Ledger. After a transaction is posted it appears on the Ledger (provided it is valid), however it may happen not immediately, and some delay is possible. There is an upper bound $\max_{\mathsf{Ledger}}$ on this delay. This corresponds to an assumption that sooner or later every transaction will appear in some Bitcoin block. We use this assumption very mildly and e.g. $\max_{\mathsf{Ledger}} = 1$ day is also ok for us (the only price for this is that in such case we have to allow the adversary to delay the termination of the protocol for time $O(\max_{\mathsf{Ledger}})$). Each transaction posted on the Ledger has a time stamp that refers to the moment when it appeared on the Ledger.

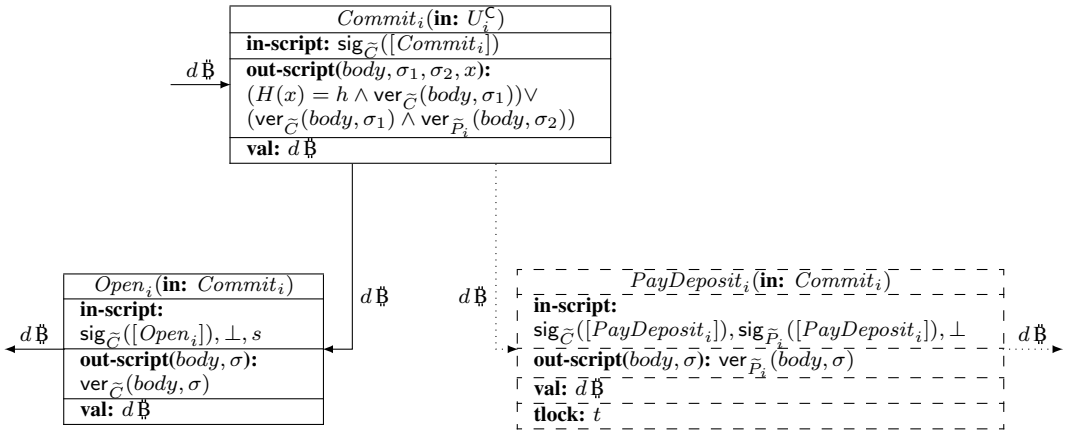
What is a bit less obvious is how to define privacy of the communication between the parties and the Ledger, especially the question of the privacy of the writing procedure. More precisely, the problem is that it is completely unreasonable to assume that a transaction is secret until it appears on the Ledger (since the transactions are broadcast between the nodes of the network). Hence we do not assume it. This actually poses an additional challenge in designing the protocols because of the problem of the *malleability*[11] of the transactions. Let us explain it now. Recall that the transactions are referred to by their hashes. Suppose a party P creates a transaction T and, before posting it on the Ledger, obtains from some other party P' a transaction T' that redeems T (e.g.: T' may be time-locked and serve P to redeem T if P' misbehaves). Obviously T' needs to contain (in the signed body) a hash $H(T)$ of T. However, if now P posts T then an adversary (allied with malicious P') is able to produce another transaction $\hat{T}$ whose semantics is the same as T, but whose hash is different (this can be done, e.g., by adding some dummy instructions to the input scripts of T). The adversary can now post $\hat{T}$ on the Ledger and, if he is lucky, $\hat{T}$ will appear on the Ledger instead of T! In this case T' will be invalid, so P may lose the money. It is possible that in the future versions of the Bitcoin system this issue will be addressed and the transactions will not be malleable. In Section V we propose a scheme that is secure under the assumption that the communication between the parties and the Ledger is private. We would like to stress that our main schemes (Section III and IV) do *not* not assume non-malleability of the transactions, and are secure even if the adversary obtains full information on the transactions before they appear on the Ledger.

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

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

We do not need to assume any privacy of the reading procedure, i.e. each party accesses pattern to Ledger can be publicly known. 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. In particular we assume that the network is asynchronous and our protocols are also secure if the communication takes some small amount of time. For simplicity we also assume that the transaction fees are zero. The extension to non-zero transaction fees is discussed in Section. VI.

III. BITCOIN-BASED TIMED COMMITMENT SCHEME



Pre-condition:

1) The key pair of C is $\widetilde{C}$ and the key pair of each P_i is $\widetilde{P}_i$.
2) The Ledger contains n unredeemed transactions $U_1^{\mathsf{C}}, \ldots, U_n^{\mathsf{C}}$, which can be redeemed with key $\widetilde{C}$, each having value d₿.

The $\mathsf{CS.Commit}(\mathsf{C}, d, t, s)$ **phase**

3) The Committer C computes $h = H(s)$. He sends to the Ledger the transactions $Commit_1, \ldots, Commit_n$. This obviously means that he reveals h, as it is a part of each $Commit_i$.
4) If within time $\max_{\mathsf{Ledger}}$ some of the $Commit_i$ transactions does not appear on the Ledger, or if they look incorrect (e.g. they differ in the h value) then the parties abort.
5) The Committer C creates the bodies of the transactions $PayDeposit_1, \ldots, PayDeposit_n$, signs them and for all i sends the signed body $[PayDeposit_i]$ to P_i. If an appropriate transaction does not arrive to P_i, then he halts.

The $\mathsf{CS.Open}(\mathsf{C}, d, t, s)$ **phase**

6) The Committer C sends to the Ledger the transactions $Open_1, \ldots, Open_n$, what reveals the secret s.
7) If within time t the transaction $Open_i$ does not appear on the Ledger then P_i signs and sends the transaction $PayDeposit_i$ to the Ledger and earns d₿.

Fig. 1. The CS protocol. The scripts' arguments, which are omitted are denoted by $\perp$.

We start with constructing a Bitcoin-based timed-commitment scheme [11], [27]. Recall that a commitment scheme [9], [12] consists of two phases: the *commitment phase* $Commit$ and the *opening phase* $Open$. Typically a commitment scheme is executed between two parties: a committer C and a recipient. To be more general we will assume that there are n recipients, denoted $\mathsf{P}_1, \ldots, \mathsf{P}_n$. The committer starts the protocol with some secret value x. This value will become known to every recipient after the opening phase is executed. Informally, we require that, if the committer is honest, then before the opening phase started, the adversary has no information about x (this property is called "hiding"). On the other hand, every honest recipient can be sure that, no matter how a malicious sender behaves, the commitment can be open in exactly one way, i.e. it is impossible for the committer to "change his mind" and open with some $x' \neq x$. This property is called "binding". Although incredibly useful in many applications, the standard commitment schemes suffer from the following problem: there is no way to force the committer to reveal his secret x, and, in particular, if he aborts before the $Open$ phase starts then x remains secret.

Bitcoin offers an attractive way to deal with this problem. Namely: using the Bitcoin system one can force the committer to back his commitment with some money, called the *deposit*, that will be given to the other parties if he refuses to open the commitment within some time t.

We now sketch the definition of a *Bitcoin-based commitment scheme*. First, assume that before the protocol starts the Ledger contains n unredeemed standard transactions $U_1^{\mathsf{C}}, \ldots, U_n^{\mathsf{C}}$ that can be redeemed with a key known only to C, each having value d₿ (for some parameter d). In fact, in real life it would be enough to have just one transaction, that would later be "split" inside of the protocol. This would, however, force us to use the multiple-output transactions which we want to avoid, in order not to additionally complicate the description of the system.

The protocol is denoted $\mathsf{CS}(\mathsf{C}, d, t, s)$ and it consists of two phases: the *commitment* phase, denoted $\mathsf{CS.Commit}(\mathsf{C}, d, t, s)$ (where s contains the message to which C commits and some randomness) and the *opening* phase $\mathsf{CS.Open}(\mathsf{C}, d, t, s)$. The honest

committer always opens his commitment by time t. In this case he gets back his money, i.e. the Ledger consists of standard transactions that can be redeemed with a key known only to him, whose total value[12] is $(d \cdot n)$ Ƀ.

The security definition in the standard commitment scheme: assuming that the committer is honest, the adversary does not learn any significant information about x before the opening phase, and each honest party can be sure that there is at most one value x that the committer can open in the opening phase. Each recipient can also abort the commitment phase (which happens if he discovers that the Committer is cheating, or if the adversary disturbs the communication). However, there is one additional security guarantee: if the committer did not open the commitment by time t then every other party earns d Ƀ. More precisely: for every honest P_i the Ledger contains a transaction, whose value is d Ƀ, that can be redeemed with a key known only to P_i.

Let us also comment on the formal aspects. To satisfy the page limit, we do not provide the full formal model, however, from the discussion above it should be clear how such a model can be defined. We allow negligible error probabilities both in binding and in hiding. Also, the last security property (concerning the deposits) has to hold only with overwhelming probability. As these notions are asymptotic, this requires using a security parameter, denoted by k. Of course, in reality the parameter k is partially fixed by the Bitcoin specification (e.g. we cannot modify the length of the outputs of the hash functions).

A. The implementation

Our implementation can be based on any standard commitment scheme as long as it is *hash-based*, by which we mean that it has the following structure. Let H be a hash function. During the commitment phase the committer sends to the recipient some value denoted h (which essentially constitutes his "commitment" to x), and in the opening phase the committer sends to the recipient a value s, such that $H(s) = h$. If $H(s) \neq h$ then the recipient does not accept the opening. Otherwise he computes x from s (there exists an algorithm that allows him to do it efficiently). One example of such a commitment scheme is as follows. Suppose $x \in \{0,1\}^*$. In the commitment phase C computes $s := (x||r)$, where r is chosen uniformly at random from $\{0,1\}^k$, and sends to every recipient $h = H(s)$. In the opening phase the committer sends to every recipient s, the recipient checks if indeed $h = H(s)$, and recovers x by stripping-off the last k bits from s. The binding property of this commitment follows from the collision-resistance of the hash function H, since to be able to open the commitment in two different ways a malicious sender would need to find collisions in H. For the hiding property we need to assume that H is a random oracle. We think that this is satisfactory since anyway the security of the Bitcoin PoWs relies on the random oracle assumption. Clearly, if H is a random oracle then no adversary can obtain any information about x if he does not learn s (which an honest C keeps private until the opening phase).

The basic idea of our protocol is as follows. The committer will talk independently to each recipient P_i. For each of them he will create in the commitment phase a transaction $Commit_i$ with value d that normally will be redeemed by him in the opening phase with a transaction $Open_i$. The transaction $Commit_i$ will be constructed in such a way that the $Open_i$ transaction has to automatically open the commitment. Technically it will be done by constructing the output script of $Commit_i$ in such a way that the redeeming transaction has to provide s (which will therefore become publicly known as all transactions are publicly visible). Of course, this means that the money of the committer is "frozen" until he reveals s. However, to set a limit on the waiting time of the recipient, we also require the committer to send to $\mathcal{P}_i$ a transaction $PayDeposit_i$ that can redeem $Commit_i$ if time t passes. Of course, P_i, after receiving $PayDeposit_i$ needs to check if it is correct. The commitment scheme and the transactions are depicted on Figure 1 (page 8). We now state the following lemma, whose proof will appear in the extended version of this paper.

Lemma 1: The CS scheme on Figure 1 is a Bitcoin-based commitment scheme.

IV. The lottery protocol

As discussed in the introduction, as an example of an application of the "MPCs on Bitcoin" concept we construct a protocol for a lottery executed among a group of parties $\mathsf{P}_1, \ldots, \mathsf{P}_N$. We say that a protocol is a *fair lottery protocol* if it is *correct* and *secure*. To define correctness assume all the parties are following the protocol and the communication channels between them are secure (i.e. it reliably transmits the messages between the parties without delay).

We assume that before the protocol starts, the Ledger contains unredeemed standard transactions $T^1, \ldots, T^N$ known to all the parties, all of value 1 Ƀ and each T^i can be redeemed with a key known only to P_i. Moreover, since we will use the commitment scheme from Section III, the parties need to have money to pay the "deposits". This money will come from transactions $\{U^i_j\}$, where $i, j \in \{1, \ldots, N\}$ and $i \neq j$, such that each U^i_j can be redeemed only by P_i and has value d Ƀ (for some parameter d whose value will be determined later). We assume that these transactions are on the Ledger before the protocol starts. The protocol has to terminate in time $O(\max_{\mathsf{Ledger}})$ and at the moment of termination, the Ledger has to contain a standard transaction with value N Ƀ which can be redeemed with a key known only to P_w, were w is chosen uniformly at random from the set $\{1, \ldots, N\}$. The Ledger also contains transactions for paying back the deposits, i.e. we require that for each P_i there is an additional transaction (that can be redeemed only by him) whose value is $(N-1)d$ Ƀ. Of course, in the case of the non-zero fees these values will be slightly smaller, but to keep things simple we assume here that these fees are zero.

[12] In case of non-zero transaction fees this value can be decreased by these fees. This remark applies also to the amounts d redeemed by the recipients.

To define security, look at the execution of the protocol from the point of view of one party, say P_1 (the case of the other parties is symmetric). Assume P_1 is honest and hence, in particular, the Ledger contains the transactions $T^1, U_2^1, \ldots, U_N^1$, whose recipient is P_1 and whose value is: $1\,\ddot{\mathbb{B}}$ in case of T^1 and $d\,\ddot{\mathbb{B}}$ in case of the U_j^1's. Obviously, P_1 has no guarantee that the protocol will terminate successfully, as the other party can, e.g., leave the protocol before it is completed. What is important is that P_1 should be sure that she will not lose money because of this termination (in particular: the other parties should not be allowed to terminate the protocol after he learned that P_1 won). This is formalized as follows: we define the *payoff of* P_1 in the execution of the protocol to be equal to the difference between the money that P_1 invested and the money that he won. More formally, the payoff of P_1 is equal to $X_1 - ((N-1)\cdot d + 1)\,\ddot{\mathbb{B}}$, where X_1 is defined as the total sum of the values of transactions from the execution of the protocol (including $T^1, U_2^1, \ldots, U_N^1$) that P_1 (and only him) can redeem when the protocol terminates. (The payoff of any other participant P_i is defined symmetrically.)

Ideally we would like to require that the expected payoff of each honest player cannot be negative[13]. However, since the security of our protocol relies on non-perfect cryptographic primitives, such as commitment schemes, we have to take into account a negligible probability of the adversary breaking them. Hence, we require only that these values are "at least negligible"[14] in some security parameter k (that is used in the crypto primitives). Formally, we say that the protocol is *secure* if for any strategy of the adversary, that controls the network and corrupts the other parties, (1) the execution of the protocol terminates in time $O(\max_{\mathsf{Ledger}})$, and (2) the expected payoff of each honest party is at least negligible. The expected values are taken over all the randomness in the experiment (i.e. both the internal randomness of the parties and the adversary). We also note that, of course, a dishonest participant can always interrupt in a very early stage. This is not a problem if the transaction fees are zero. In case of the non-zero transaction fees this may cause the other parties to lose a small amount of money. This problem is addressed in Section VI.

A. The protocol

Our protocol is built on top of the classical coin-tossing protocol of Blum [9] that is based on cryptographic commitments. The Blum's scheme adapted to N parties is very simple — each party P_i commits herself to an element $b_i \in Z_N$. Then, the parties open their commitments and the winner is P_w where $w = (b_1 + \cdots + b_N \bmod N) + 1$. As described in the introduction, this protocol does not directly work for our applications, and we need to adapt it to Bitcoin. In particular, in our solution creating and opening the commitments are done by the transactions' scripts using *double* SHA-256 hashing[15]. Due to the technical limitations of Bitcoin scripting language in its current form[16], instead of random numbers b_i, the parties commit themselves to strings s_i sampled with uniformly random length from $\mathcal{S}_k^N := \{0,1\}^{8k} \cup \ldots \cup \{0,1\}^{8(k+N-1)}$, i.e. the set of strings of length $k, \ldots, (k+N-1)$ bytes[17], where k is the security parameter. The winner is determined by the *winner choosing function* $f(s_1, \ldots, s_N)$ and in our protocol $f(s_1, \ldots, s_N) = \mathsf{P}_\ell$ if $\sum_{i=1}^N |s_i| \equiv (\ell - 1) \bmod N$, where $s_1, \ldots, s_N$ are the secret strings chosen from $\mathcal{S}_k^N$ and $|s_i|$ is a length of string s_i in bytes. Honest users first randomly choose length (in bytes) of their strings from the set $\{k, \ldots, k+N-1\}$ and then generate a random string of the appropriate length. It is easy to see that as long as one of the parties draws her string's length uniformly, then the output of $f(s_1, \ldots, s_N)$ is also uniformly random.

1) First attempt: For simplicity let us start with the case of $N = 2$ parties, called Alice A and Bob B. Their key pairs are A and B (resp.) and their unredeemed transactions placed on the Ledger before the protocol starts are denoted $T^{\mathsf{A}}, U^{\mathsf{A}}$ and $T^{\mathsf{B}}, U^{\mathsf{B}}$. We start with presenting a naive and insecure construction of the protocol, and then show how it can be modified to obtain a secure scheme. The protocol starts with Alice and Bob creating their new pairs of keys $\widetilde{A} = (\widetilde{A}.sk, \widetilde{A}.pk)$ and $\widetilde{B} = (\widetilde{B}.sk, \widetilde{B}.pk)$, respectively. These keys will be used during the protocol. It is actually quite natural to create new keys for this purpose, especially since many Bitcoin manuals recommend creating a fresh key pair for every transaction. Anyway, there is a good reason to do it in our protocol, e.g. to avoid interference with different sessions of the same protocol. Both parties announce their public keys to each other. Alice and Bob also draw at random their secret strings s_{A} and s_{B} (respectively) from the set $\mathcal{S}_k^2$ and they exchange the hashes $h_{\mathsf{A}} = H(s_{\mathsf{A}})$ and $h_{\mathsf{B}} = H(s_{\mathsf{B}})$. Moreover Alice sends to the Ledger the following transaction:

$PutMoney_1^{\mathsf{A}}$(**in:** T^{A})
in-script: $\mathsf{sig}_{\mathsf{A}}([PutMoney_1^A])$
out-script$(body, \sigma)$**:** $\mathsf{ver}_{\widetilde{A}}(body, \sigma)$
val: $1\,\ddot{\mathbb{B}}$

[13] In principle it can be actually positive if the adversary plays against his own financial interest.

[14] Formally: a function $\alpha : \mathbf{N} \to \mathbf{R}$ is *at least negligible* if there exists a function $\beta : \mathbf{N} \to \mathbf{R}$ such that for every i we have $\alpha(i) \geq \beta(i)$ and β is negligible, i.e. its absolute value is asymptotically smaller than the inverse of any polynomial.

[15] Notice that use of *single* SHA-256 would be insecure here, because it is constructed using Merkle–Damgard transformation and therefore it is susceptible to the *length extension attack* [21]. It this attack an adversary which knows $H(x)$ can compute a value $H(x||y)$ for some string y controlled by him without the knowledge of the original value x. It could allow to completely compromise the lottery protocol, because the winner choosing function (described later) highly depends on the lengths of the secrets.

[16] Most of the more advanced instructions (e.g. concatenation, accessing particular bits in a string or arithmetic on big integers) have been disabled out of concern that the clients may have bugs in their implementation. Therefore, computing length (in bytes), hashing and testing equality are the only operations available for strings.

[17] The transactions in the protocol will always check if their inputs are from $\mathcal{S}_k^N$ (whenever they are supposed to be from this set). If not, they are considered invalid, and the transaction is not evaluated.

Bob also sends to the Ledger a transaction $PutMoney_1^{\mathsf{B}}$ defined symmetrically (recall that T^{A} and T^{B} are standard transactions that can be redeemed by Alice and Bob respectively). If at any point later a party $P \in \{\mathsf{A}, \mathsf{B}\}$ realizes that the other party is cheating, then the first thing P will do is to "take the money and run", i.e. to post a transaction that redeems $PutMoney_1^P$. We will call it "halting the execution". This can clearly be done as long as $PutMoney_1^P$ has not been redeemed by some other transaction. In the next step one of the parties constructs a transaction $Compute_1$ defined as follows:

$Compute_1$(**in:** $PutMoney_1^{\mathsf{A}}, PutMoney_1^{\mathsf{B}}$)	
in-script$_1$**:** $\mathsf{sig}_{\widetilde{A}}([Compute_1])$	**in-script**$_2$**:** $\mathsf{sig}_{\widetilde{B}}([Compute_1])$
out-script$(body, \sigma_1, \sigma_2, \hat{s}_{\mathsf{A}}, \hat{s}_{\mathsf{B}})$**:** $(\hat{s}_{\mathsf{A}}, \hat{s}_{\mathsf{B}} \in \mathcal{S}_k^2 \wedge H(\hat{s}_{\mathsf{A}}) = h_{\mathsf{A}} \wedge H(\hat{s}_{\mathsf{B}}) = h_{\mathsf{B}}$ $\wedge\ f(\hat{s}_{\mathsf{A}}, \hat{s}_{\mathsf{B}}) = \mathsf{A} \wedge \mathsf{ver}_{\widetilde{A}}(body, \sigma_1)) \quad \vee$ $(\hat{s}_{\mathsf{A}}, \hat{s}_{\mathsf{B}} \in \mathcal{S}_k^2 \wedge H(\hat{s}_{\mathsf{A}}) = h_{\mathsf{A}} \wedge H(\hat{s}_{\mathsf{B}}) = h_{\mathsf{B}}$ $\wedge\ f(\hat{s}_{\mathsf{A}}, \hat{s}_{\mathsf{B}}) = \mathsf{B} \wedge \mathsf{ver}_{\widetilde{B}}(body, \sigma_2))$	
val: $2\,\text{฿}$	

Note that the body of $Compute_1$ can be computed from the publicly-available information. Hence this construction can be implemented as follows: first one of the players, say, Bob computes $[Compute_1]$, and sends his signature $\mathsf{sig}_{\widetilde{B}}([Compute_1])$ on it to Alice. Alice adds her signature $\mathsf{sig}_{\widetilde{A}}([Compute_1])$ and posts the entire transaction $Compute_1$ to the Ledger.

The output script of $Compute_1$ is an alternative of two conditions. Since they are symmetric (with respect to A and B) let us only look at the first condition (call it γ). To make it evaluate to true on $body$ one needs to provide as "witnesses" $(\sigma_1, \hat{s}_{\mathsf{A}}, \hat{s}_{\mathsf{B}})$ where $\hat{s}_{\mathsf{A}}$ and $\hat{s}_{\mathsf{B}}$ are the pre-images of h_{A} and h_{B} (with respect to H) from $\mathcal{S}_k^2$. Clearly the collision-resistance of H implies that $\hat{s}_{\mathsf{A}}$ and $\hat{s}_{\mathsf{B}}$ have to be equal to s_{A} and s_{B} (resp.). Hence γ can be satisfied only if the winner choosing function f evaluates to A on input $(s_{\mathsf{A}}, s_{\mathsf{B}})$. Since only Alice knows the private key of $\widetilde{A}$, only she can later provide a signature σ_1 that would make the last part of γ (i.e.: "$\mathsf{ver}_{\widetilde{A}}(body, \sigma_1)$") evaluate to true.

Clearly before $Compute_1$ appears on the Ledger each party P can "change her mind" and redeem her initial transaction $PutMoney_1^P$, which would make the transaction $Compute_1$ invalid. As we said before, it is ok for us if one party interrupts the coin-tossing procedure as long as she had to decide about doing it *before* she learned that she lost. Hence, Alice and Bob wait until $Compute_1$ appears on the Ledger before they proceed to the step in which the winner is determined. This final step is simple: Alice and Bob just broadcast s_{A} and s_{B}, respectively. Now: if $f(s_{\mathsf{A}}, s_{\mathsf{B}}) = \mathsf{A}$ then Alice can redeem the transaction $Compute_1$ in a transaction $ClaimMoney_1^{\mathsf{A}}$ constructed as:

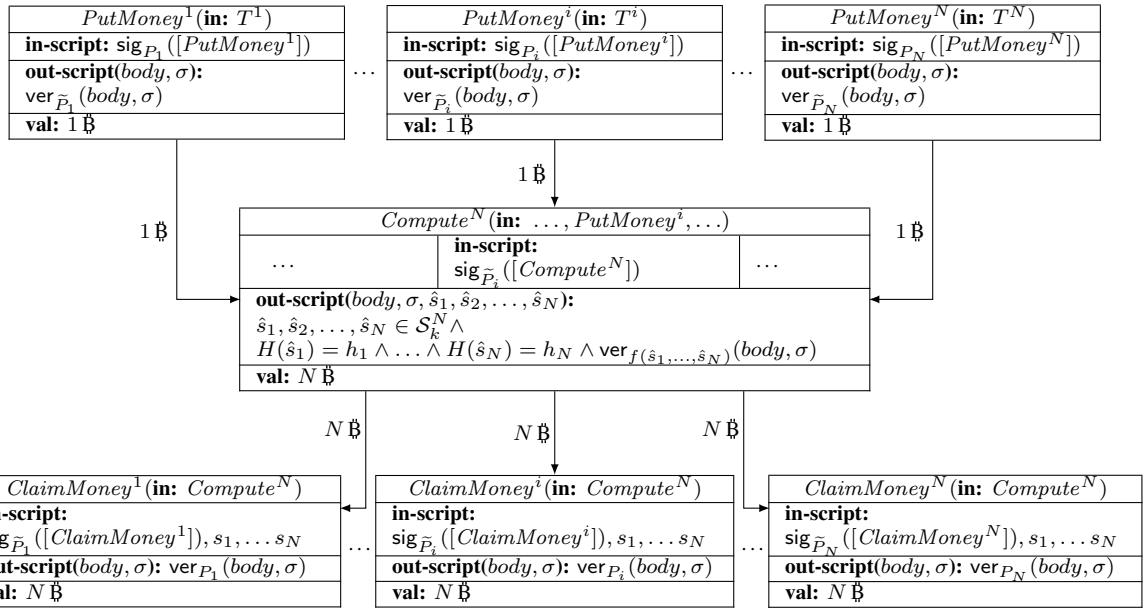
$ClaimMoney_1^{\mathsf{A}}$(**in:** $Compute_1$)
in-script: $\mathsf{sig}_{\widetilde{A}}([ClaimMoney_1^{\mathsf{A}}]), \bot, s_{\mathsf{A}}, s_{\mathsf{B}}$
out-script$(body, \sigma)$**:** $\mathsf{ver}_{\mathsf{A}}(body, \sigma)$
val: $2\,\text{฿}$

On the other hand Bob cannot redeem $Compute_1$, as the condition $f(s_{\mathsf{A}}, s_{\mathsf{B}}) = \mathsf{B}$ evaluates to false. Symmetrically: if $f(s_{\mathsf{A}}, s_{\mathsf{B}}) = \mathsf{B}$ then only Bob can redeem $Compute_1$ by an analogous transaction $ClaimMoney_1^{\mathsf{B}}$.

This protocol is obviously correct. It may also look secure, as it is essentially identical to Blum's protocol described at the beginning of this section (with the hash functions used as the commitment schemes). Unfortunately, it suffers from the following problem: there is no way to guarantee that the parties always send s_{A} and s_{B}. In particular: one party, say, Bob, can refuse to send s_{B} *after* he learned that he lost (i.e. that $f(s_{\mathsf{A}}, s_{\mathsf{B}}) = \mathsf{A}$). As his money is already "gone" (his transaction $PutMoney_1^{\mathsf{B}}$ has already been redeemed in transaction $Compute_1$) he would do it just because of sheer nastiness. Unfortunately in a purely peer-to-peer environment, with no concept of a "reputation", such behavior can happen, and there is no way to punish it.

This is exactly why we need to use the Bitcoin-based commitment scheme from Section III. Later, in Section V we also present another technique for dealing with this problem, which avoids using the deposits. Unfortunately, it suffers from certain shortcomings. First of all, it works only for two parties. Secondly, and more importantly, to achieve full security it needs an assumption that an adversary cannot see transactions until they appear on the Ledger.

2) The secure version of the scheme: The general idea behind the secure MultiPlayersLottery protocol is that each party first commits to her inputs using the $\mathsf{CS}(\mathsf{C}, d, t, s)$ commitment scheme, instead of the standard commitment scheme (the parameters d and t will be determined later). Recall that the CS commitment scheme can be opened by sending a value s, and this opening is verified by checking that s hashes to a value h sent by the committer in the commitment phase. So, Alice executes the CS protocol acting as the committer and Bob as a recipient (note that there is only one recipient and hence $n = 1$). Let s_{A} and h_{A} be the variables s and h created this way. Symmetrically: Bob executes the CS protocol acting as the committer, and Alice being the recipient, and the corresponding variables are s_{B} and h_{B}. Once both commitment phases are executed successfully (recall that this includes receiving by each party the signed $PayDeposit$ transaction), the parties proceed to the next steps, which are exactly as before: first, each of them posts his transaction $PutMoney$ on the Ledger. Once all these transactions appear on the Ledger they create the $Compute^2$ transaction (in the same way as before), and once it appears on the Ledger they open the commitments. The only difference is obviously that, since they used the CS commitment scheme, they can now "punish" the



Pre-condition:

1) For each i, player P_i holds a pair of keys $(P_i.sk, P_i.pk)$.
2) For each i, the Ledger contains a standard transaction T^i that has value 1 ₿ and whose recipient is P_i. The Ledger contains also the transactions $\{U_j^i\}$, for $i, j \in \{1, \ldots, N\}$ and $i \neq j$, such that each U_j^i can be redeemed by P^i and has value $d = N$ ₿.

Initialization phase:

3) For each i, player P_i generates a pair of keys $(\widetilde{P}_i.sk, \widetilde{P}_i.pk)$ and sends his public key $\widetilde{P}_i.pk$ to all other players.
4) For each i, player P_i chooses his secret s_i from $\mathcal{S}_k^N$ (with uniformly random length).

Deposits phase:

5) Let t be the current time. For each i, the commitment phase $\mathsf{CS.Commit}(\mathsf{P}_i, d, t + 4 \cdot \max_{\mathsf{Ledger}}, s_i)$ is executed using the transactions $\{U_j^i\}$ as inputs.
6) If any two commitments of different players are equal (i.e. $h_i = h_j$ for $i \neq j$) then the players abort the protocol[18].

Execution phase:

7) For each i, player P_i puts the transaction $PutMoney^i$ to the Ledger. The players halt if any of those transactions did not appear on the Ledger before time $t + 2 \cdot \max_{\mathsf{Ledger}}$.
8) For each $i \geq 2$, player P_i computes his signature on the transaction $Compute^N$ and sends it to the player P_1.
9) Player P_1 puts all received signatures (and his own) into inputs of transaction $Compute^N$ and puts it to the Ledger. If $Compute^N$ did not appear on the Ledger in time $t + 3 \cdot \max_{\mathsf{Ledger}}$, then the players halt.
10) For each i, the player P_i puts his $Open$ transactions on the Ledger what reveals his secret and sends back to him the deposits he made during the executions of CS protocol from Step. 5. If some player did not reveal his secret in time $t + 4 \cdot \max_{\mathsf{Ledger}}$, then other players send the appropriate $PayDeposit$ transactions from that player CS protocols to the Ledger to get N ₿.
11) The player, that is the winner (i.e. $P_{f(s_1,\ldots,s_N)}$), gets the pot by sending the transaction $ClaimMoney^{f(s_1,\ldots,s_N)}$ to the Ledger.

Fig. 2. The MultiPlayersLottery protocol.

other party if she did not open her commitment by executing $PayDeposit$ after the time t passes, and claim her deposit. On the other hand: each honest party is always guaranteed to get her deposit back, hence she does not risk anything investing this money at the beginning of the protocol.

It is straightforward how to extend this protocol for any number of players. The more detailed description is presented on Figure 2 (page 12).

We also need to comment about the choice of the parameters t and d. First, it easy to see that the maximum time in which the honest parties will complete the protocol is at most $4 \cdot \max_{\mathsf{Ledger}}$ after time t' — the time when the protocol started. Hence we can safely set $t := t' + 4 \cdot \max_{\mathsf{Ledger}}$.

The parameter d should be chosen in such a way that it will fully compensate to each party the fact that a player aborted. Let us now calculate the payoff of some fixed player P_1, say, assuming the worst-case scenario, which is as that (a) the protocol is always aborted when P_1 is about to win, and (b) there is only one "aborting party" (so P_i is paid only one deposit). Hence his expected payoff is $-\frac{N-1}{N}$ ₿ (this corresponds to the case when he lost) plus $\frac{d-1}{N}$ ₿ (the case when the protocol was aborted). Therefore to make the expected value equal to 0 we need to set $d = N$ ₿. This implies that the total amount of money invested

[18]We would like to thank Ranjit Kumaresan and Iddo Bentov for pointing out this step. It protects from the *copy attack*: e.g. in case of two players lottery P_1 waits until P_2 commits with his hash h_2 and then commits with the same hash. During the opening phase P_1 again waits until P_2 reveals his secret s_2 and reveals the same secret. By doing this he always wins since $f(s_2, s_2) = \mathsf{P}_1$.

in deposit by each player has to be equal to $N(N-1)$ ₿. In real-life this would be ok probably for small groups $N = 2, 3$, but not for the larger ones.

We now have the following lemma, whose proof will appear in the extended version of this paper.

Lemma 2: The MultiPlayersLottery protocol from Figure 2 is a fair lottery protocol for $d = N$ ₿ and $t = t' + 4 \cdot \max_{\mathsf{Ledger}}$, where t' is the starting time of the protocol.

V. Two-party lottery secure in a stronger model

In this section we show a construction of a two-party lottery which avoids using the deposits, and hence may be useful for applications where the parties are not willing to invest extra money in the execution of the protocol. The drawback of the protocol presented in this section is that it works only for two parties. Moreover, to achieve full security it needs an assumption that the channel between the parties and the Ledger is private, what means that the adversary cannot see the transactions sent by the honest user, before they appear on the Ledger. In reality Bitcoin transactions are broadcast via a peer-to-peer network, so it is relatively easy to eavesdrop the transactions waiting to be posted on the Ledger. Another problem related to eavesdropping is malleability of transaction (already described in Section II-B). Recall, that the problem is that an adversary can modify ("maul") the transaction T eavesdropped in the network in such a way that the modified transaction is semantically equivalent to the original one, but it has a different hash. Then, the adversary can send the modified version of T to the Ledger and if he is lucky it will be posted on the Ledger and invalidate the original transaction T (its input will be already redeemed). Hence, e.g., the transactions that were created to redeem T will not be able to do it (as the hash of the transaction is different). In order to be secure for the protocol presented in this section, we need to assume that such attacks are impossible. Technically, we do it by assuming that the channel from each party to the Ledger is private. The protocols secure in this model will be called secure *under private channel assumption*.

To explain our protocol let us go to the point in Section IV where it turned out that we need the Bitcoin commitment schemes. Recall that we observed that the protocol from Section IV-A2 is not secure against a "nasty behavior" of the party that, after realizing that she lost, simply quits the protocol.

3) An alternative (and slightly flawed) idea for a fix: Suppose for a moment we are only interested in security against the "nasty Bob". Our method is to force him to reveal s_{B} simultaneously with $Compute_1$ being posted on the Ledger, by requiring that s_{B} is a part of $Compute_1$. More concretely this is done as follows. Recall that in our initial protocol we said that $Compute_1$ is created and posted on the Ledger by "one of the parties". This was ok since the protocol was completely symmetric for A and B. In our new solution we break this symmetry by modifying the $Compute_1$ transaction (this new version will be denoted $Compute_2$) and designing the protocol in such a way that $Compute_2$ will be always posted on the Ledger by B. First of all, however, we redefine the $PutMoney_1^{\mathsf{B}}$ transaction that B posts on the Ledger at the beginning of the protocol. The modified transaction is denoted $PutMoney_2^{\mathsf{B}}$.

$PutMoney_2^{\mathsf{B}}$(**in:** T^{B})
in-script: $\mathsf{sig}_{\mathsf{B}}([PutMoney_2^{\mathsf{B}}])$
out-script$(body, \sigma, \hat{s})$**:** $\mathsf{ver}_{\widetilde{B}}(body, \sigma) \ \wedge \ \hat{s} \in \mathcal{S}_k^2 \ \wedge \ H(\hat{s}) = h_{\mathsf{B}}$
val: 1 ₿

The only difference compared to $PutMoney_1^{\mathsf{B}}$ is the addition of the "$\wedge\, \hat{s} \in \mathcal{S}_k^2 \wedge (H(\hat{s}) = h_{\mathsf{B}})$" part. This trick forces Bob to reveal the pre-image of h_B (which has to be equal to s_{B}) whenever he redeems $PutMoney_2^{\mathsf{B}}$.

The transaction $PutMoney_1^{\mathsf{A}}$ remains unchanged, i.e.: $PutMoney_2^{\mathsf{A}} := PutMoney_1^{\mathsf{A}}$. Clearly players can still redeem their transactions later in case they discover that the other player is cheating. Transaction $Compute_2$ is the same as $Compute_1$, except that s_{B} is added to the input script for the second input transaction:

$Compute_2$(**in:** $PutMoney_2^{\mathsf{A}}, PutMoney_2^{\mathsf{B}}$)	
in-script$_1$**:** $\mathsf{sig}_{\widetilde{A}}([Compute_2])$	**in-script**$_2$**:** $\mathsf{sig}_{\widetilde{B}}([Compute_2]), s_{\mathsf{B}}$
out-script$(body, \sigma_1, \sigma_2, \hat{s}_{\mathsf{A}}, \hat{s}_{\mathsf{B}})$**:** $(\hat{s}_{\mathsf{A}}, \hat{s}_{\mathsf{B}} \in \mathcal{S}_k^2 \wedge H(\hat{s}_{\mathsf{A}}) = h_{\mathsf{A}} \wedge H(\hat{s}_{\mathsf{B}}) = h_{\mathsf{B}}$ $\wedge\ f(\hat{s}_{\mathsf{A}}, \hat{s}_{\mathsf{B}}) = \mathsf{A} \wedge \mathsf{ver}_{\widetilde{A}}(body, \sigma_1))$ $\vee$ $(\hat{s}_{\mathsf{A}}, \hat{s}_{\mathsf{B}} \in \mathcal{S}_k^2 \wedge H(\hat{s}_{\mathsf{A}}) = h_{\mathsf{A}} \wedge H(\hat{s}_{\mathsf{B}}) = h_{\mathsf{B}}$ $\wedge\ f(\hat{s}_{\mathsf{A}}, \hat{s}_{\mathsf{B}}) = \mathsf{B} \wedge \mathsf{ver}_{\widetilde{B}}(body, \sigma_2))$	
val: 2 ₿	

The parties post their $PutMoney_2$ transactions on the Ledger and construct transaction $Compute_2$ in the following way. First, observe that both parties can easily construct the body of $Compute_2$ themselves, as all the information needed for this is: the transactions $PutMoney_2^{\mathsf{A}}$ and $PutMoney_2^{\mathsf{B}}$ and the hashes of s_{A} and s_{B}, which are all publicly available. Hence the only thing that needs to be computed are the input scripts. This computation is done as follows: first Alice computes her input script $\mathsf{sig}_{\widetilde{A}}([Compute_2])$ and sends it to Bob. Then Bob adds his input script $(\mathsf{sig}_{\widetilde{B}}([Compute_2]), s_{\mathsf{B}})$, and posts $Compute_2$ on the Ledger.

The $ClaimMoney_2^P$ procedures (for $P \in \{\mathsf{A}, \mathsf{B}\}$) remain unchanged (except, of course that their input is $Compute_2$ instead of $Compute_1$). Let us now analyze the security of this protocol from the point of view of both parties. First, observe that Alice does not risk anything by sending $\mathsf{sig}_{\widetilde{A}}([Compute_2])$ to Bob. This is because it consists of a signature on the entire body of the transaction, and hence it is useless as long as Bob did not add his input script[19]. But, if Bob added a correct input script and posted $Compute_2$ on the Ledger then he automatically had to reveal s_{B}. Hence, from the point of view of Alice the problem of "nasty Bob" is solved.

Unfortunately, from the point of view of Bob the situation looks much worse, as he still has no guarantee that Alice will post s_{A} once she learned that she lost. This is why one more modification of the protocol is needed.

4) The secure version of the scheme: To fix the problem described above we extend our protocol by adding a special transaction that we denote $Fuse$ and that will be used by Bob to redeem $Compute$ if Alice did not send s_{A} within some specific time, say, $2 \cdot \max_{\mathsf{Ledger}}$. To achieve this we will use the time-lock mechanism described in the introduction. This requires modifying once again the $Compute_2$ transaction so it can be redeemed by $Fuse$. All in all, the transactions are now defined as follows:

$Compute$(**in:** $PutMoney^{\mathsf{A}}, PutMoney^{\mathsf{B}}$)	
in-script$_1$**:** $\mathsf{sig}_{\widetilde{A}}([Compute])$	**in-script**$_2$**:** $\mathsf{sig}_{\widetilde{B}}([Compute]), s_{\mathsf{B}}$
out-script$(body, \sigma_1, \sigma_2, \hat{s}_{\mathsf{A}}, \hat{s}_{\mathsf{B}})$**:** $(\hat{s}_{\mathsf{A}}, \hat{s}_{\mathsf{B}} \in \mathcal{S}_k^2 \wedge H(\hat{s}_{\mathsf{A}}) = h_{\mathsf{A}} \wedge H(\hat{s}_{\mathsf{B}}) = h_{\mathsf{B}}$ $\wedge f(\hat{s}_{\mathsf{A}}, \hat{s}_{\mathsf{B}}) = \mathsf{A} \wedge \mathsf{ver}_{\widetilde{A}}(body, \sigma_1)) \quad \vee$ $(\hat{s}_{\mathsf{A}}, \hat{s}_{\mathsf{B}} \in \mathcal{S}_k^2 \wedge H(\hat{s}_{\mathsf{A}}) = h_{\mathsf{A}} \wedge H(\hat{s}_{\mathsf{B}}) = h_{\mathsf{B}}$ $\wedge f(\hat{s}_{\mathsf{A}}, \hat{s}_{\mathsf{B}}) = \mathsf{B} \wedge \mathsf{ver}_{\widetilde{B}}(body, \sigma_2)) \quad \vee$ $(\mathsf{ver}_{\widetilde{A}}(body, \sigma_1) \wedge \mathsf{ver}_{\widetilde{B}}(body, \sigma_2))$	
val: $2\ddot{\mathrm{B}}$	

$Fuse$(**in:** $Compute$)
in-script: $\mathsf{sig}_{\widetilde{A}}([Fuse]), \mathsf{sig}_{\widetilde{B}}([Fuse]), \bot, \bot$
out-script$(body, \sigma)$**:** $\mathsf{ver}_{\mathsf{B}}(body, \sigma)$
val: $2\ddot{\mathrm{B}}$
tlock: $t + 2 \cdot \max_{\mathsf{Ledger}}$

(t above refers roughly to the time when $Fuse$ is created, we will define it more concretely in a moment).

The transactions $ClaimMoney^A$ and $ClaimMoney^B$ are almost the same as the transactions $ClaimMoney_2^A$ and $ClaimMoney_2^B$ (except that they redeem $Compute$ transaction instead of $Compute_2$). It is clear that $Compute$ can be generated jointly by Alice and Bob in the same way as before (the only new part of $Compute$ is the last line "$\mathsf{ver}_{\widetilde{A}}(body, \sigma_1) \wedge \mathsf{ver}_{\widetilde{B}}(body, \sigma_2)$" that can be easily computed by both parties from the public information).

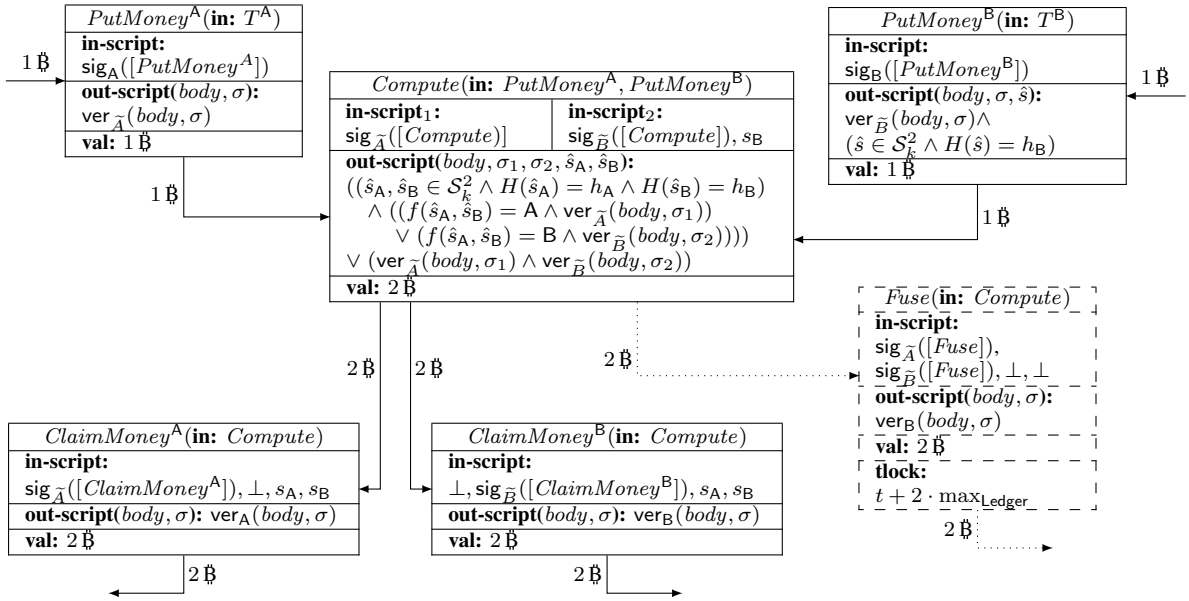
What remains is to describe the construction of the $Fuse$ transaction. Clearly, Bob can create the entire $Fuse$ by himself, except of the signature $\mathsf{sig}_{\widetilde{A}}([Fuse])$ in the input script, which has to be computed by Alice, as only she knows her private key. To do this Alice needs to know the body of $Fuse$. It is easy to see that she knows all of it, except of the input transaction $Compute$. Moreover, Bob cannot simply send $Compute$ to Alice, since $Compute$ includes the information about his secret s_{B} which Alice should not learn at this point.

We solve this problem by exploiting the details of the Bitcoin implementation, namely the fact that the transactions are referenced by their hashes. Hence, to create the body of $Fuse$ Alice only needs to know the *hash* $h_{Compute}$ of $Compute$. Therefore our protocol will contain the following sub-procedure (executed directly after Bob constructs $Compute$, but before he posts it on the Ledger): (1) Bob sends $h_{Compute} = H(Compute)$ to Alice, (2) Alice computes $[Fuse]$, signs it, and sends the signature $\mathsf{sig}_{\widetilde{A}}([Fuse])$ to Bob, (3) Bob verifies Alice's signature and halts if it is incorrect. Time t that is used in the time-lock in $Fuse$ will refer to time when Alice executed Step (2) above. This system guarantees that Bob can always claim his $2\ddot{\mathrm{B}}$ in time $t + 2 \cdot \max_{\mathsf{Ledger}}$ even if Alice did not execute the last step. Observe that of course Alice should halt her execution if she does not see $Compute$ on the Ledger within time $t + \max_{\mathsf{Ledger}}$, as otherwise Bob could simply post $Compute$ much later (after time $t + 2 \cdot \max_{\mathsf{Ledger}}$, say) and immediately use $Fuse$ to claim the reward.

There are some issues in this procedure that need to be addressed. Firstly, the reader may be worried that $H(Compute)$ reveals some information on $Compute$. In practice (and in theory if H is a random oracle) this happens only if the set of possible inputs to H is small and known to the adversary. In our case the adversary is the dishonest Alice, and it can be easily seen that from her point of view the set of possible $Compute$ transactions is huge, one reason for this being that $Compute$ includes s_{B}, which is secret and uniform.

Unfortunately the fact that Alice does not know the complete transaction $Compute$, but only its hash, poses a risk to her. This is because a dishonest Bob can, instead of sending $H(Compute)$, send a hash of some other transaction T in order to obtain the information that can be used to redeem some other transaction used within the protocol, or even outside this session of the protocol. This is actually one of the reasons why we assumed that the keys used by the users in our procedure are

[19]Recall that the body of a transaction includes also the information about its input transactions, and moreover, a transaction becomes valid only if *all* the input transactions can be redeemed



Pre-condition:

1) Alice holds a pair of keys $A = (sk_A, pk_A)$ and Bob holds a pair of keys $B = (sk_B, pk_B)$.
2) The Ledger contains standard transactions T^{A} and T^{B} that have value 1₿ each and whose recipients are pk_A and pk_B, respectively.

Initialization phase:

3) Alice and Bob generate their key pairs $\widetilde{A} = (\widetilde{A}.sk, \widetilde{A}.pk)$ and $\widetilde{B} = (\widetilde{B}.sk, \widetilde{B}.pk)$ (respectively) and exchange the public keys $\widetilde{A}.pk$ and $\widetilde{B}.pk$.
4) The players choose their secret strings s_{A} and s_{B}.
5) Alice computes a hash of her secret $h_{\mathsf{A}} := H(s_{\mathsf{A}})$ and sends it to Bob.
6) Bob computes a hash of his secret $h_{\mathsf{B}} := H(s_{\mathsf{B}})$ and sends it to Alice.
7) If the commitments are equal (i.e. $h_{\mathsf{A}} = h_{\mathsf{B}}$) then the players abort the protocol.
8) Each $P \in \{\mathsf{A}, \mathsf{B}\}$ computes $PutMoney^P$ and posts it on the Ledger. The players proceed to the next step only once both of these transactions appear on the Ledger.

Computation phase:

9) The players construct the $Compute$ transaction as follows:
   a) Alice computes the body of the transaction $Compute$ together with the signature $\mathsf{sig}_{\widetilde{A}}([Compute])$ and sends $\mathsf{sig}_{\widetilde{A}}([Compute])$ to Bob.
   b) Bob verifies Alice's signature and halts if it is incorrect. Otherwise he computes the whole transaction $Compute$ by adding a signature $\mathsf{sig}_{\widetilde{B}}([Compute])$ to the message received in the previous step.
10) The players construct the $Fuse$ transaction as follows:
   a) Bob sends $h_{Compute} = H(Compute)$ to Alice,
   b) Alice computes $[Fuse]$, signs it, and sends the signature $\mathsf{sig}_{\widetilde{A}}([Fuse])$ to Bob (let t denote the time when it happened),
   c) Bob verifies Alice's signature and halts if it is incorrect.

Execution phase:

11) Bob sends $Compute$ to the Ledger. Note that this reveals s_{B}. If $Compute$ did not appear on the Ledger in time $t + \max_{\mathsf{Ledger}}$ then Alice halts.
12) Alice sends s_{A} to Bob. If Bob did not receive it in time $t + 2 \cdot \max_{\mathsf{Ledger}}$ then he sends the $Fuse$ transaction to the Ledger.
13) If $f(s_{\mathsf{A}}, s_{\mathsf{B}}) = \mathsf{A}$ then Alice sends $ClaimMoney^{\mathsf{A}}$ to the Ledger, otherwise Bob sends $ClaimMoney^{\mathsf{B}}$ to the Ledger.

Fig. 3. The TwoPlayersLottery protocol

fresh and will not be used later: in this way we can precisely know, which transactions can be redeemed if one obtains Alice's signature on $[Fuse]$ constructed with false $h_{Compute}$.[20]. It is easy to see that the only transaction other than $Compute$, that could be potentially redeemed using Alice's signature is $PutMoney^{\mathsf{A}}$. This transaction cannot be redeemed by "$Fuse$ with false $h_{Compute}$", for several reasons, one of them being that the value of $PutMoney^{\mathsf{A}}$ is 1₿, which is less than the value of $Fuse$ (equal to 2₿).

In this way we constructed the TwoPlayersLottery protocol. Its complete description is presented on Figure 3 (page 15). We now have the following lemma (the proof will appear in the extended version of this paper).

Lemma 3: The TwoPlayersLottery protocol from Figure 3 is a secure lottery protocol under the private channel assumption.

Notice that without the private channel assumption there are two possible attacks, which could harm Bob. One is that Alice could see $Compute$, which contains s_{B} before it is posted on the Ledger. In case she lost she could react with sending to the

[20] As a more concrete example what could go wrong without this assumption consider the following scenario. Assume there is a not-redeemed transaction $Compute^*$ on the Ledger whose recipient is Alice and that also can be redeemed by a transaction with an input script $(\mathsf{sig}_{\widetilde{A}}([T]), \mathsf{sig}_{\widetilde{B}}([T]), \bot, \bot)$ (this can happen, e.g., if two coin-tossing protocol are executed in parallel between Alice and Bob). Then a dishonest Bob can send to Alice $H(Compute^*)$ instead of $H(Compute)$, and redeem $Compute^*$ in time $t + 2 \cdot \max_{\mathsf{Ledger}}$

Ledger another transaction T, which redeems $PutMoney^{\mathsf{A}}$. If she was lucky and the transaction T was posted on the Ledger before $Compute$, then $Compute$ would become invalidated (one of its inputs would be already redeemed) and Bob would not earn any money. The other possible attack concerns the malleability problem: for the security to hold Bob needs to be sure that the $Fuse$ transaction will redeem the transaction $Compute$. Unfortunately, $Fuse$ has to be created strictly before $Compute$ appears on the Ledger. If an adversary intercepts $Compute$ before it happened (or: if a miner is malicious) then he can post a "mauled" $Compute$ transaction on the Ledger that behaves exactly as the original one, except that it has a different hash. Hence $Fuse$ would becomes useless.

VI. Non-zero transaction fees

We now address the problem of the transaction fees, which was ignored in the description above. On a technical level there is no problem with incorporating the fees into our protocol: the transactions can simply include a small fee that has to be agreed upon between the parties before the protocol starts. The expected payoff of the parties will be in this case slightly negative (since the fees need to be subtracted from the outcome). It is straightforward how to modify the security definition to take this into account. One problem that the reader may notice is the issue of the "nasty" behavior of the parties. For example, a malicious Alice can initiate the protocol with Bob just to trigger him to post $PutMoney^{\mathsf{B}}$ on the Ledger. If Alice later aborts then Bob obviously gets his money back, except of the transaction fee. Of course, this does not change his expected payoff, but it still may be against his interests, as he loses some money on a game that from the beginning was planned (by Alice) never to start.

We now describe a partial pragmatic remedy for this problem. The basic idea is to modify the protocol by changing the instructions what to do when the other parties misbehave. Recall, that in our protocols the parties are instructed to simply redeem all their transactions if they notice a suspicious behavior of the other party. Now, instead of doing this, they could keep these transactions on the Ledger and reuse them in some other sessions of the protocol. Of course, this has to be done with care. For example the timed commitment schemes have to be redeemed within a certain time frame). One also has to be careful to avoid problems with reusing the keys, described above, cf. Footnote 20. To argue formally about the security of this solution one would need to introduce a multi-player mathematical model capturing the fact that several sessions can be executed with shared secrets. This is beyond the scope of this paper, so we just stay on this informal level.

VII. Implementation

As a "proof of concept" we have implemented and executed the presented protocols. The transactions were created using *bitcoinj* Java library, as normal Bitcoin clients do not allow user to create (nor broadcast) non-standard transactions, and sent directly to *Eligius* mining pool, which is currently the only one accepting non-standard transactions[21].

Below we present links to some of these transactions on the blockchain.info website. To save space, all links are relative to the url blockchain.info/tx-index/ (hence, they are only indices of transactions used by the blockchain.info site).

Commitment scheme CS*:* Links to all transactions in a correct execution of the commitment scheme with one recipient are as follows: $Commit$: 97079150; $Open$: 97094781.

Here is an example of an execution for two recipients, which finished with $PayDeposit$ transactions broadcast: $Commit$: 96947667; $PayDeposit^1$: 96982401; and $PayDeposit^2$: 96982398.

*Three-party lottery (*MultiPlayersLottery*):* We have performed a correct execution of the three-party lottery protocol, where each player bets 0.0012฿. First, the players perform standard transactions with output value 0.0012฿. The transactions are as follows: $PutMoney^A$: 96946847; $PutMoney^B$: 96946887; and $PutMoney^C$: 96947563. Then the players exchange the hashes h_A, h_B and h_C, and sign and broadcast the $Compute$ transaction (96964833). After the revealing of their secrets s_A, s_B and s_C (by opening the commitments), the winner (in this case, player C) broadcasts the $ClaimMoney^C$ transaction (96966124) to get the pot.

*Two-party lottery without deposits (*TwoPlayersLottery*):* Below we present links to all transactions in a correct execution of the protocol won by Alice: $PutMoney^A$: 96424665; $PutMoney^B$: 96436412; $Compute$: 96436416; $ClaimMoney^A$: 96436417; In this execution, the players bet 0.04 ฿ each and the transaction fees were set to 0.0001 ฿ for each transaction.

We also performed an execution which finished with the $Fuse$ transaction: $PutMoney^A$: 97094615; $PutMoney^B$: 97094780; $Compute$: 97099280; $Fuse$: 97105484.

As an example of a raw transaction we present the $PutMoney^B$ transaction from the first of the TwoPlayersLottery protocol executions described above (96436412) in more details. Here is its dump (with some fields omitted):

```
{ "lock_time":0,
  "in":[{ "prev_out":  {"hash":"a14...096", "n":0},
```

[21] Unfortunately, it seems that currently Eligius is not accepting any transaction T_x redeeming T_y if it earlier received some other transaction T'_x redeeming T_y, even if T'_x is time-locked in the future. We would like to stress that this is just an implementation decision made by the Eligius administrators and it does not follow from the Bitcoin specification. We hope that this feature of Eligius will be removed in the future (or, that the other mining pools will start accepting the non-standard transactions).

```
  "scriptSig":"304...a01 039...443" }],
"out":[{"value":"0.03990000",
       "scriptPubKey":"
       OP_SIZE 32 34 OP_WITHIN OP_VERIFY
       OP_SHA256 f53...226 OP_EQUALVERIFY
       020...e33 OP_CHECKSIG" }]}
```

The meaning of the above is as follows. `"lock_time":0` means, that the transaction does not have a time lock. `"hash":"a14...096"` denotes a hash of the transaction, which is being redeemed in $PutMoney^B$ and `"n":0` denotes, which output of that transaction is being redeemed. The input script `"scriptSig":"304...a01 039...443"` consists of the signature (`039...443`) on $PutMoney^B$ under the key $B.pk$ and the public key $B.pk$ itself (`304...a01`) (in standard transaction's output there is only pk's hash and pk has to be included in the corresponding input).

The output script expects to get as input an appropriate signature and Bob's secret string. The script consists of three parts: the first part `OP_SIZE 32 34 OP_WITHIN OP_VERIFY` checks whether the second argument has an appropriate length; the second part `OP_SHA256 f53...226 OP_EQUALVERIFY` checks if its hash is equal to h_B i.e. `f53...226`); and the last part `020...e33 OP_CHECKSIG` checks if the first argument is an appropriate signature under key $\widetilde{B}.pk$ (i.e. `020...e33`).

REFERENCES

[1] Ittai Abraham, Danny Dolev, Rica Gonen, and Joseph Y. Halpern. Distributed computing meets game theory: robust mechanisms for rational secret sharing and multiparty computation. In Eric Ruppert and Dahlia Malkhi, editors, *25th ACM Symposium Annual on Principles of Distributed Computing*, pages 53–62. ACM Press, July 2006.

[2] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Łukasz Mazurek. Fair two-party computations via bitcoin deposits. Cryptology ePrint Archive, Report 2013/837, accepted to the First Workshop on Bitcoin Research 2014 (in Assocation with Financial Crypto), 2013. http://eprint.iacr.org/.

[3] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Łukasz Mazurek. How to deal with malleability of bitcoin transactions. *CoRR*, abs/1312.3230, 2013.

[4] Adam Back and Iddo Bentov. Note on fair coin toss via bitcoin, 2013. http://www.cs.technion.ac.il/~idddo/cointossBitcoin.pdf.

[5] Simon Barber, Xavier Boyen, Elaine Shi, and Ersin Uzun. Bitter to better - how to make Bitcoin a better currency. In Angelos D. Keromytis, editor, *FC 2012: 16th International Conference on Financial Cryptography and Data Security*, volume 7397 of *Lecture Notes in Computer Science*, pages 399–414. Springer, February / March 2012.

[6] Amos Beimel, Yehuda Lindell, Eran Omri, and Ilan Orlov. $1/p$-Secure multiparty computation without honest majority and the best of both worlds. In Phillip Rogaway, editor, *Advances in Cryptology – CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 277–296. Springer, August 2011.

[7] Assaf Ben-David, Noam Nisan, and Benny Pinkas. FairplayMP: a system for secure multi-party computation. In Peng Ning, Paul F. Syverson, and Somesh Jha, editors, *ACM CCS 08: 15th Conference on Computer and Communications Security*, pages 257–266. ACM Press, October 2008.

[8] Iddo Bentov and Ranjit Kumaresan. How to use bitcoin to design fair protocols. Cryptology ePrint Archive, Report 2014/129, 2014. http://eprint.iacr.org/.

[9] Manuel Blum. Coin flipping by telephone. In Allen Gersho, editor, *Advances in Cryptology – CRYPTO'81*, volume ECE Report 82-04, pages 11–15. U.C. Santa Barbara, Dept. of Elec. and Computer Eng., 1981.

[10] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael I. Schwartzbach, and Tomas Toft. Secure multiparty computation goes live. In Roger Dingledine and Philippe Golle, editors, *FC 2009: 13th International Conference on Financial Cryptography and Data Security*, volume 5628 of *Lecture Notes in Computer Science*, pages 325–343. Springer, February 2009.

[11] Dan Boneh and Moni Naor. Timed commitments. In Mihir Bellare, editor, *Advances in Cryptology – CRYPTO 2000*, volume 1880 of *Lecture Notes in Computer Science*, pages 236–254. Springer, August 2000.

[12] Gilles Brassard, David Chaum, and Claude Crepeau. Minimum disclosure proofs of knowledge. *Journal of Computer and System Sciences*, 37(2):156 – 189, 1988.

[13] Vitalik Buterin. The bitcoin gambling diaspora, Aug 2013. Bitcoin Magazine.

[14] Vitalik Buterin. Satoshidice sold for $12.4 million, Jul 2013. Bitcoin Magazine.

[15] Christian Cachin and Jan Camenisch. Optimistic fair secure computation. In Mihir Bellare, editor, *Advances in Cryptology – CRYPTO 2000*, volume 1880 of *Lecture Notes in Computer Science*, pages 93–111. Springer, August 2000.

[16] Jan Camenisch, Susan Hohenberger, and Anna Lysyanskaya. Compact e-cash. In Ronald Cramer, editor, *Advances in Cryptology – EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 302–321. Springer, May 2005.

[17] David Chaum. Blind signature system. In David Chaum, editor, *Advances in Cryptology – CRYPTO'83*, page 153. Plenum Press, New York, USA, 1983.

[18] David Chaum, Amos Fiat, and Moni Naor. Untraceable electronic cash. In Shafi Goldwasser, editor, *Advances in Cryptology – CRYPTO'88*, volume 403 of *Lecture Notes in Computer Science*, pages 319–327. Springer, August 1988.

[19] Jeremy Clark and Aleksander Essex. CommitCoin: Carbon dating commitments with Bitcoin - (short paper). In Angelos D. Keromytis, editor, *FC 2012: 16th International Conference on Financial Cryptography and Data Security*, volume 7397 of *Lecture Notes in Computer Science*, pages 390–398. Springer, February / March 2012.

[20] Richard Cleve. Limits on the security of coin flips when half the processors are faulty. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, STOC '86, pages 364–369, New York, NY, USA, 1986. ACM.

[21] Jean-Sebastien Coron, Yevgeniy Dodis, Cecile Malinaud, and Prashant Puniya. Merkle-Damgård revisited: How to construct a hash function. In Victor Shoup, editor, *Advances in Cryptology - CRYPTO 2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 430–448. Springer Berlin Heidelberg, 2005.

[22] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In *ESORICS 2013: 18th European Symposium on Research in Computer Security*, Lecture Notes in Computer Science, pages 1–18. Springer, 2013.

[23] John R. Douceur. The sybil attack. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 251–260, London, UK, UK, 2002. Springer-Verlag.

[24] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In Ernest F. Brickell, editor, *Advances in Cryptology – CRYPTO'92*, volume 740 of *Lecture Notes in Computer Science*, pages 139–147. Springer, August 1992.

[25] The Economist. Online gambling: Know when to fold, 2013.

[26] Eric J. Friedman and Paul Resnick. The social cost of cheap pseudonyms. *Journal of Economics and Management Strategy*, 10:173–199, 2000.

[27] Juan A. Garay and Markus Jakobsson. Timed release of standard digital signatures. In Matt Blaze, editor, *FC 2002: 6th International Conference on Financial Cryptography*, volume 2357 of *Lecture Notes in Computer Science*, pages 168–182. Springer, March 2002.

[28] Juan A. Garay, Jonathan Katz, Ueli Maurer, Björn Tackmann, and Vassilis Zikas. Rational protocol design: Cryptography against incentive-driven adversaries. In *54th Annual Symposium on Foundations of Computer Science*, pages 648–657. IEEE Computer Society Press, 2013.

[29] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *19th Annual ACM Symposium on Theory of Computing*, pages 218–229. ACM Press, May 1987.

[30] S. Dov Gordon and Jonathan Katz. Partial fairness in secure two-party computation. In Henri Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science*, pages 157–176. Springer, May 2010.

[31] Joseph Y. Halpern and Vanessa Teague. Rational secret sharing and multiparty computation: Extended abstract. In László Babai, editor, *36th Annual ACM Symposium on Theory of Computing*, pages 623–632. ACM Press, June 2004.

[32] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay - a secure two-party computation system. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 20–20, Berkeley, CA, USA, 2004. USENIX Association.

[33] Ian Miers, Christina Garman, Matthew Green, and Aviel D. Rubin. Zerocoin: Anonymous distributed e-cash from bitcoin. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 397–411, Washington, DC, USA, 2013. IEEE Computer Society.

[34] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[35] Benny Pinkas. Fair secure two-party computation. In Eli Biham, editor, *Advances in Cryptology – EUROCRYPT 2003*, volume 2656 of *Lecture Notes in Computer Science*, pages 87–105. Springer, May 2003.

[36] The Washington Post. Cheating scandals raise new questions about honesty, security of internet gambling, November 30, 2008.

[37] Paul Resnick, Ko Kuwabara, Richard Zeckhauser, and Eric Friedman. Reputation systems. *Commun. ACM*, 43(12):45–48, December 2000.

[38] Dorit Ron and Adi Shamir. Quantitative analysis of the full bitcoin transaction graph. In *FC 2013: 17th International Conference on Financial Cryptography and Data Security*, Lecture Notes in Computer Science, pages 6–24. Springer, 2013.

[39] Adi Shamir, Ron Rivest, and Leonard Adleman. Mental poker, April 1979. Technical Report LCS/TR-125, Massachusetts Institute of Technology.

[40] Luis von Ahn, Nicholas J. Hopper, and John Langford. Covert two-party computation. In Harold N. Gabow and Ronald Fagin, editors, *37th Annual ACM Symposium on Theory of Computing*, pages 513–522. ACM Press, May 2005.

[41] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th Annual Symposium on Foundations of Computer Science*, pages 162–167. IEEE Computer Society Press, October 1986.