DOI:10.1145/2896386

Secure Multiparty Computations on Bitcoin

By Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Łukasz Mazurek

Abstract

Is it possible to design an online protocol for playing a lottery, in a completely decentralized way, that is, without relying on a trusted third party? Or can one construct a fully decentralized protocol for selling secret information, so that neither the seller nor the buyer can cheat in it? Until recently, it seemed that every online protocol that has financial consequences for the participants needs to rely on some sort of a trusted server that ensures that the money is transferred between them. In this work, we propose to use Bitcoin (a digital currency, introduced in 2008) to design such fully decentralized protocols that are secure even if no trusted third party is available. As an instantiation of this idea, we construct protocols for secure multiparty lotteries using the Bitcoin currency, without relying on a trusted authority. 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 in principle could be used in real life as a replacement for the online gambling sites.

1. INTRODUCTION

One of the most attractive features of the Internet is its decentralization: the TCP/IP protocol itself, and several other protocols running on top of it do not rely on a single server, and often can be executed between parties that do not need to trust each other, or even do not need to know each other's true identity. Examples of such protocols include: the SMTP and the HTTP protocols, the peer-to-peer content distributions platforms, messaging systems, and many others. A natural question to ask is how far can the "decentralization" of the digital world go? In other words, what are the real-life applications which one can implement on the Internet without the need of a trusted third party? Until recently, one notable example of a task that seemed to always require some sort of a "trusted server" was the online financial transactions (that had to rely on a bank or a credit card company). This situation changed radically in 2009 when the first fully decentralized digital currency, called Bitcoin, was deployed by Nakamoto.[17, a] The huge success of Bitcoin (its current market capitalization is around $5 billion) is due precisely to its distributed nature and the lack of a central authority that controls Bitcoin transactions. We describe Bitcoin in more detail in Section 2.

The fact that Bitcoin money transfers can be done without a trusted server raises another intriguing question, namely, can we "decentralize" the financial system even further, that is, can we implement some more advanced financial instruments in a distributed manner? The Bitcoin specification partly answers this question, by providing the so-called "nonstandard transactions." We describe this feature in more detail in Section 2, but for a moment, let us only say that Bitcoin allows the parties to specify more complex conditions about when the money can be spent. This, in turn, permits them to create the so-called "Bitcoin contracts," which are forms of agreements whose execution is later enforced by the Bitcoin system itself (without the need of a trusted third party). Examples of such contracts include rapidly adjusted micropayments, assurance contracts, and dispute mediation (see https://en.bitcoin.it/wiki/Contracts for more on this).

Probably, one of the most advanced types of multiparty protocols that can be performed digitally are the cryptographic "secure multiparty computation (MPC)" protocols, originating from the seminal works of Yao[20] and Goldreich et al.[14] Informally, such protocols allow a group of mutually distrusting parties to compute a joint function f on their private inputs. For example, for two parties, Alice and Bob, Alice has an input x, Bob has an input y, and they both want to learn $f(x, y)$, but without Alice learning y or Bob learning x. In this paper, we initiate the study of using Bitcoin to perform MPC protocols.

The coin-tossing protocol. A very simple example of such a protocol is the *coin-tossing problem*,[6] executed between two parties, Alice and Bob, who want to jointly compute a bit b that is equally likely to be 0 or 1. In other words, they want to compute a randomized function $f_{\text{rnd}} : \{\perp\} \times \{\perp\} \to \{0, 1\}$ that takes no inputs and outputs as a uniformly random bit. This protocol can be implemented using an idea similar to the rock-paper-scissors game: Alice sends a bit b_A to Bob, and simultaneously Bob sends a bit b_B to Alice. The output b is computed

A longer version of this paper appeared on the IEEE Symposium on Security and Privacy 2014. An extended version of it is also available on the Cryptology Eprint Archive eprint.iacr.org/2013/784. 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. Łukasz Mazurek is a recipient of the Google Europe Fellowship in Security, and this research is supported in part by this Google Fellowship.

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

as $b := b_A \oplus b_B$ (where "$\oplus$" denotes the *x* or function). Clearly if at least one of the bits b_A and b_B is uniformly random, then b is also uniformly random, and hence each party can be sure that the game is fair, as long as she behaves honestly (i.e., chooses her bit uniformly). When one tries to implement this protocol over the Internet, then of course, the main challenge is to ensure that Alice and Bob send their bits simultaneously. This is because if one party, say Alice, can choose her bit b_A after she learns b_B, then she can make b equal to any value b' she wants by choosing $b_A := b' \oplus b_B$.

The solution proposed in Blum[6] is to use a tool called a *cryptographic commitment scheme*. Informally, such a scheme is a two-party protocol executed between a *committer* and a *receiver*. At the beginning, the committer knows some value s that is secret to the receiver. The parties first perform the *commitment phase* (*Commit*). After this phase is executed, the receiver still does not know s (this property is called *hiding*). Later, the parties execute the opening phase (*Open*) during which the receiver learns s. The key property of a commitment scheme is that the committer cannot "change his mind" after the commitment phase. More precisely, after the first phase is executed, there exists precisely one value s that can be opened in the second phase. This property is called *binding*. In some sense, the commitment phase is analogous to sending a message s in a locked box, and the opening phase can be thought of as sending the key to the box. Clearly after the box is sent, the committer cannot change its contents, but before getting the key, the receiver does not know what is inside the box.

There exist several secure methods of constructing such commitments. In this paper, we use the ones that are based on the cryptographic hash functions (see Section 3).

It is now easy to see how a commitment scheme can be used to solve the coin-tossing problem: instead of sending her bit b_A directly to Bob, Alice just commits to it (i.e., Alice and Bob execute the commitment scheme with Alice acting as the committer, Bob acting as the receiver, and b_A being the secret). Symmetrically, Bob commits to his bit b_B. After this commitment phase is over, the parties execute the opening phase and learn each other's bits. Then the output is computed as $b = b_A \oplus b_B$. The security of the commitment scheme guarantees that no party can choose her bit depending on the bit of the other party, and hence this procedure produces a uniformly random bit.

Boolean operations. The coin-tossing example above is a particularly simple case of a multiparty protocol since the parties that execute it do not take any inputs. To explain what we mean by a protocol where the parties do take inputs, consider the case when the function that Alice and Bob compute is the 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 for example, $b = 0$, then Bob after learning the output of the function has no information about Alice's input. Therefore, the privacy of Alice is protected.

One can generalize this example and consider the *set-intersection* problem. Here Alice and Bob have sets A and B as their inputs and the output is equal to $f_\cap(A, B) = A \cap B$. For example, think of A and B as sets of e-mail addresses in Alice's and Bob's contact lists—then the output $f_\cap(A, B)$ is the list of the contacts that they have in common. The security here means that: (1) the parties do not learn about each other's input more than they can deduce from their own input and the output, and (2) a malicious party cannot cause the result to be incorrect (e.g., a corrupt Alice cannot falsely make Bob think that some e-mail address is in her contact list). For this example, condition (1) means that for every $a \notin A$, Alice should obtain no information if a is in B (and symmetrically for Bob).

General results and the lack of "fairness." The above examples can be generalized in several ways. First of all, one can consider protocols executed among groups of parties of size larger than two (hence the name MPCs, as opposed to the *two*-party examples above). For example, a multiparty coin-tossing protocol is specified exactly as the two-party one, except that the number of the participants is larger than two.

Second, one can consider more complicated functions than the ones described above. It was shown in Goldreich et al.[14] that for any efficiently computable function f (including "randomized" functions like the one in the coin-tossing example), there exists an efficient protocol that securely computes it, assuming the existence of trapdoor permutations (which is a well-established assumption, widely believed to hold). If a 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. However, if more than half of the parties are malicious, then the malicious parties can terminate the protocol after learning the output, preventing the honest parties from learning it. Note that in case of 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. This problem is visible in the coin-tossing example above, as each party can refuse to open her commitment after she learned what was the bit of the other party. In some cases, this is not a problem since the parties can agree that refusing to open the commitment is equivalent to losing the game.

However, it turns out[9] that in general this problem, called the lack of *fairness*, is unavoidable. Hence, two-party protocols in general do not provide complete fairness.

Why are the MPC not widely used over the Internet? Since the introduction of MPCs there has been a significant effort to make these protocols efficient[4, 10, 16] and sometimes even to use them in the real-life applications such as the online auctions.[7] 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 Internet gambling: it may be intriguing that currently gambling over the Internet is done almost entirely with the help of websites that play the roles of "trusted parties," instead of using a cryptographic coin-flipping protocol to eliminate the need for trust. 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.[18] 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 gambling sites typically charge fees for their service.

In our opinion, there are at least two main reasons why MPCs are not used for online gambling. The first reason is that 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. This means that two-party coin tossing is not secure in practice. More generally, multiparty coin tossing would work only if the majority is honest, which is not a realistic assumption in the fully distributed Internet environment, for instance, *sybil* attacks[11] allow one malicious party to create and control 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: such protocols take care only of the security of the computation and are not "responsible" for ensuring 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 there is no technological way to ensure that the users honestly provide their input to the trusted party. Nothing prevents one party, say Bob, from lying about his feelings and setting $b = 1$ 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 nondigital world, is to use the concept of "reputation": a party caught 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.[19] However, they seem too cumbersome to use in many applications, one reason being that it is unclear how to define the reputation of new users if users are allowed to pick new names whenever they want.[12]

Another option is to exploit the fact that the financial transactions are done electronically. 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 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. Even if theoretically possible, this solution is 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 main contribution of this paper is the introduction of a new paradigm, which we call "MPC protocols on Bitcoin," that provides a solution to both of the problems described above: the lack of fairness and the lack of the link between "real life" and the result of the cryptographic computation. We describe our solution in Section 1.1.

1.1. 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,"[8, 13] where the committer has to reveal his secret within a certain time frame or 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 MPCs."

What is probably more interesting is our second idea, which in some sense 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 MPCs guarantees only that the protocol performs the computation securely, but ensuring that the inputs are correct and the parties do not interrupt the protocol execution is beyond the scope of the security definition. Our observation is that the Bitcoin system can be used to go beyond this standard definition, by constructing protocols that link the inputs and the outputs with 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 lottery 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 protocol works in purely peer-to-peer environment and can be executed between players who 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 main construction is presented in Section 4. Its security is obtained via *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 for the fact that the game terminated prematurely. This protocol uses the timed commitment scheme described above. A 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, where N is the number of players. For the two-party case, this simply means that the deposit is twice the size of the bet.

The only cost that the participants need to pay in our protocols is Bitcoin transaction fees. Most Bitcoin transactions are currently free. However, the participants of our protocols need to make a small number of nonstandard transactions (the so-called "strange transactions," see Section 2), for which there is usually some small fee (currently around 0.0001 ฿ ≈ $0.04).[b] To keep the exposition simple, we present our results assuming that the fees are zero. 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 explained above. 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 MPC or zero-knowledge compilers, and hence our protocols are very efficient. The only cryptographic primitives that we use are commitment schemes, implemented using 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. We executed our transactions on the real Bitcoin. We provide a description of these transactions and a reference to them in the Bitcoin block chain.[c]

1.2. Independent and subsequent work

Usage of Bitcoin to create a secure and fair two-player lottery has been independently proposed by Back and Bentov.[3] We provide a detailed comparison between their protocol and ours in the extended version of this paper.

In the subsequent work,[1,2] we show how to extend the ideas from this paper to construct a fair two-party protocol for any functionality, in such a way that the execution of this protocol has "financial consequences." More precisely, in the first paper,[1] we show how to solve this problem under the assumption that the Bitcoin transactions are nonmalleable (see Andrychowicz et al.[1,2] for more on this notion), and in Andrychowicz et al.,[2] we show how to modify the protocol from Andrychowicz et al.[1] to obtain a protocol that is secure in the current version of Bitcoin. Some alternative ideas for obtaining fairness in the multiparty protocols were developed independently by Bentov and Kumaresan.[5,15]

1.3. Applications and future work

Although, as argued in the extended version of this paper, 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 example of a task that can be implemented using our techniques is a protocol for selling secret information for Bitcoins. Imagine Alice and Bob know a description of a set X containing some valuable information. For example, X can contain some sensitive data that is hard to find (say: personal data signed by a secret key of some public authority). Alice knows some subset A of X and Bob knows a subset B of X. Their goal is to sell to each other the elements of $A \cup B$ in such a way that they will pay to each other only for the elements they did not know in advance. In other words, Alice will pay to Bob $(|B \setminus A| - |A \setminus B|)$ ฿ (if this value is negative, then Bob will pay to Alice its negation). Without the MPC techniques, it is not clear how to do it: whenever Alice reveals to Bob some element $a \in A$, Bob can always claim that he already knew a. Moreover, even if MPC techniques are used, Alice has no way to force Bob to pay her the money (and vice-versa). Our tools (developed in the subsequent papers mentioned in Section 1.2) solve this problem: we can design a protocol that transfers exactly the right sum of Bitcoins, and moreover, this happens if and only if both parties really learned the output of the computation!

The above example can be generalized in several different ways. For example, the output can go only to one party (say: Alice), and the condition for the information that Alice is willing to pay for can be much more complicated. For example, Alice can be an intelligence agency that has a special secret function g that specifies what is the value of a given information (for some set of inputs g can even output 0). Then Bob can try to "sell" his information x to Alice setting some minimal value v that it is worth according to him. The protocol would compute $g(x)$ and check if $g(x) \geq v$—if yes, then Alice would learn x and pay v to Bob, and otherwise Alice would learn nothing (and Bob would earn 0).

Finally, let us remark that our protocols can potentially be used for malicious purposes. For example, consider ransomware that encrypts the hard disk of the victim's machine and promises to provide a decryption key only if the victim pays a ransom. Currently, such malicious programs have no way to prove that they will really send the right key if the ransom is payed. With our techniques, one can make delivery of this key secure (in the sense that the payment happens only if the key really decrypts the disk). Another potential risk is attacks on online voting schemes: it is well-known that if these schemes are not receipt-free, then the adversary can buy votes. Our techniques can make such attacks easier, as they eliminate the need of the vote seller to trust the vote buyer.

2. A SHORT DESCRIPTION OF BITCOIN

Bitcoin[17] works as a peer-to-peer network in which the participants jointly emulate a central server that controls the correctness of the transactions. In this sense, it is similar to the concept of the MPC 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

[b] We use "฿"; for the Bitcoin currency symbol.

[c] For example the main transaction (*Compute*) of the three-party lottery is available here: blockchain.info/tx/540d816bd57300209754dd36ffcec1d669bd2068641844783451cd3ef32c8aa4.

the combined computing power of all the other participants of the protocol. Hence, for example, 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 describe the functionality of the trusted party that is emulated by the users.

One of the main problems with digital currencies is 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 ledger. For example, suppose the transactions are of a form: "user A transfers x Bitcoins to user B." In this case, each user can verify if A really has x Bitcoins (i.e., she received it in some previous transactions) and she did not spend it yet. The functionality of the trusted party emulated by the Bitcoin network does precisely this: it maintains a full list of the transactions that happened in the system. The format of 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 2.1. However, for the sake of simplicity, we omit the features of Bitcoin that are not relevant to our work such as transaction fees or how the coins are created.

The Bitcoin ledger is in fact a chain of *blocks* (each block contains transactions) that all the participants are trying to extend. The parameters of the system are chosen in such a way that an extension happens on average once each 10 min. The idea of the block chain is that the longest chain C is accepted as the proper one and appending a new block to the chain takes nontrivial computation. As extending the block chain or creating a new one is very hard, all users will use the same, original block chain. Speaking in more detail, this construction prevents double spending of transactions. If a transaction is contained in a block B_i and there are several new blocks after it, then it is infeasible for an adversary with less than a half of the total computational 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, transactions need 10–20 min for reasonably strong confirmation and 60 min (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. The presence of the transaction in the block is a confirmation of this transaction, but some users may choose to wait for several blocks to get more assurance. In our protocols, we assume that there exists a maximum delay $T_{\max}$ between broadcasting the transaction and its confirmation and that every transaction once confirmed is irreversible.

2.1. Bitcoin transactions

In contrast to the classical banking system, Bitcoin is based on *transactions* instead of *accounts*. A user A has some Bitcoins if in the system there are unredeemed transactions for which he is a recipient. Each transaction has some value (number of Bitcoins which is being transferred) and a recipient *address*. An address is simply a public key pk. Normally, every such key has a corresponding private key sk known only to one user—that user is the owner of the address pk. The private key is used for signing (authorizing) 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 the key pairs using capital letters (e.g., A) and refer to the private key and the public key of A by $A.sk$ and $A.pk$, respectively.

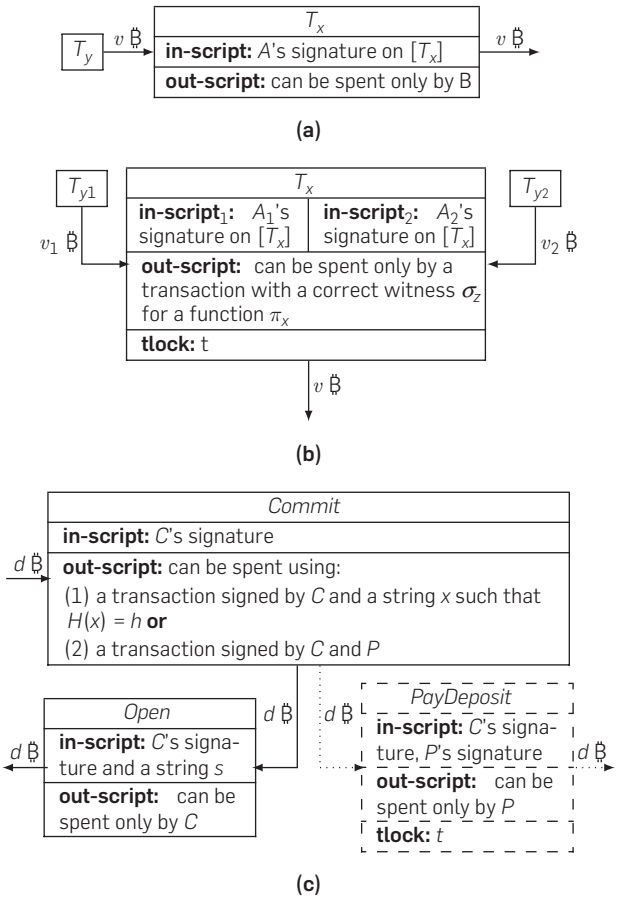
Simplified version. We first describe a simplified version of Bitcoin and then show how to extend it to obtain the description of the real Bitcoin. Let $(A.sk, A.pk)$ and $(B.sk, B.pk)$ be the key pairs belonging to users A and B, respectively. In our simplified view, a transaction describing the fact that an amount υ (called the *value* of a transaction) is transferred from an address $A.pk$ to an address $B.pk$ has the form $T_x = (y, \upsilon, B.pk, \mathsf{sig})$, where y is an index of a previous transaction T_y, and sig is a signature computed using sender's secret key $A.sk$ on the whole transaction excluding the signature itself (i.e., on $(y, \upsilon, B.pk)$). 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 T_y is *redeemed* by T_x. More precisely, the meaning of T_x is that the amount υ of money transferred to $A.pk$ in transaction T_y is transferred further to $B.pk$. The transaction T_x is valid only if (1) $A.pk$ was a recipient of the transaction T_y, (2) the value of T_y was equal to υ, (3) the transaction T_y has not been redeemed earlier, and (4) the signature of A is correct. All these conditions can be verified publicly.

We will present the transactions as boxes. The redeeming of transactions will be indicated with arrows with the value of the transaction. For example, a transaction $T_x = (y, \upsilon, B.pk, \mathsf{sig})$, which transfers υ Bitcoins from A to B, is depicted in Figure 1(a).

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 $T_{y1}, \ldots, T_{y\ell}$. 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: $T_x = (y_1, \ldots, y_\ell, \upsilon, B.pk, \mathsf{sig}_1, \ldots, \mathsf{sig}_\ell)$, where each sig_i is a signature computed using key $A_i.sk$ on the whole message excluding the signatures. The result of such transaction is that $B.pk$ gets the amount υ, provided it is equal to the sum of the values of the transactions $T_{y1}, \ldots, T_{y\ell}$. This happens only if *none* of these transactions has been redeemed before, and *all* the signatures are valid. Each transaction can also have several outputs, which is a way to divide money between several users or get change, but we do not use this feature in our protocols.

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 the hash of the whole transaction, $H(T_x)$. Hence, from now on, we will assume that $x = H(T_x)$. Moreover, each transaction can have a *time-lock* t that tells at what time the

Figure 1. (a) A standard transaction transferring υ Bitcoins from *A* to *B*, (b) a nonstandard transaction with two inputs and a time-lock, (c) the CS protocol.



transaction becomes valid. In this case, we have: $T_x = (y_1, \ldots, y_\ell, \upsilon, B.pk, t, \mathsf{sig}_1, \ldots, \mathsf{sig}_\ell)$. Such a transaction becomes valid only if the time t is reached and all the conditions mentioned earlier are satisfied. Before the time t, the transaction T_x cannot be used (it will not be included into any block before the time t).

The main difference is, however, that in the real Bitcoin, the users have much more flexibility in defining the condition on how the transaction can be redeemed. Consider for a moment the simplest transaction 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 (called *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. In case of standard transactions, 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: $T_x = (y, \pi_x, \upsilon, \sigma_x)$, where $[T_x] = (y, \pi_x, \upsilon)$ is called the *body* of T_x and σ_x is an *input-script*—a witness that is used to make the script π_y evaluate to true on T_x (in standard transactions σ_x is a signature of a sender 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 a hash function or verifying a signature. The generalization to multiple-input transactions with time-locks is straightforward: a transaction has the form $T_x = (y_1, \ldots, y_\ell, \pi_x, \upsilon, t, \sigma_1, \ldots, \sigma_\ell)$, where the body $[T_x]$ is equal to $(y_1, \ldots, y_\ell, \pi_x, \upsilon, t)$, 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_{yi}, (3) none of these transactions has been redeemed before, and (4) the sum of values of transactions T_{yi} is equal to υ.

A box representation of a general transaction with two inputs, $T_x = (y_1, y_2, \pi_x, \upsilon, t, \sigma_1, \sigma_2)$, is depicted in Figure 1(b).

The most common type of transactions is transactions without time-locks or any special script: the input script is a signature, and the output script is a signature verification algorithm. 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 standard transactions (although the nonstandard transactions are also correct according to the Bitcoin description). We believe that in the future accepting the nonstandard transactions will become common. This is important for our applications since our protocols rely heavily on nonstandard transactions.

3. BITCOIN-BASED TIMED COMMITMENT SCHEME

We start with constructing a Bitcoin-based timed commitment scheme. Commitment schemes were already described in Section 1. A simple way to implement a commitment is to use a cryptographic hash function H. To commit to a secret $s \in \{0,1\}^*$, the committer chooses a random string $r \in \{0,1\}^{128}$ and sends to the receiver $c = H(s\|r)$ (where "‖" denotes concatenation). To open the commitment, the committer sends (s, x) and the receiver verifies that $H(s\|x) = c$.

Although incredibly useful in many applications, standard commitment schemes suffer from the following problem (already described in the introduction): there is no way to force the committer to reveal his secret s, and, in particular, if he aborts before the *Open* phase starts, then s 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 recipient if he refuses to open the commitment within some time t agreed by both parties. More precisely, during the commitment phase, the committer makes a deposit in Bitcoins. He will get this deposit back if he opens the commitment before the time t. Otherwise, this deposit will be automatically given to the recipient.

3.1. Construction

Our construction of the Bitcoin-Based Timed Commitment Scheme (CS) will be based on the simple commitment scheme described earlier. The hash function used in Bitcoin is SHA256 and in our protocols we also use it because it can be used in the Bitcoin scripting language. But for clarity, we will still denote it by H in the descriptions of the protocols. Additionally,

we assume that the secret is already padded with random bits so we do not add them or strip them off in our description. In fact, we will later use the CS protocol to commit to long random strings so in that case padding is not necessary.

The basic idea of our protocol is as follows. In the commitment phase, the committer creates a transaction *Commit* with some agreed value d, which serves as the deposit. The only way to redeem the deposit is to post another transaction *Open*, which reveals the secret s. The transaction *Commit* is constructed in such a way that the *Open* transaction has to open the commitment, that is, reveal the secret value s. This means that the money of the committer is "frozen" until he reveals s. To allow the recipient to claim the deposit if the committer does not open the commitment within a certain time period, we also require the committer to send to the recipient a transaction *PayDeposit* that can redeem *Commit* if time t passes.

Technically, it is done by constructing the output script of the transaction *Commit* in such a way that the redeeming transaction has to provide either C's signature and the secret s (which will therefore become publicly known as all transactions are publicly visible) or signatures from both C and R. After broadcasting the transaction *Commit*, the committer creates the transaction *PayDeposit*, which sends the deposit to the recipient and has a time-lock t. The committer signs it and sends it to the recipient. After receiving *PayDeposit*, the recipient checks if it is correct and adds his own signature to it. After that he can be sure that either the committer will open his commitment by the time t or he will be able to use the transaction *PayDeposit* to claim the d ฿ deposit.

The graph of transactions in this protocol is depicted in Figure 1(c). The full description of the protocol can be found in the extended version of this paper.

4. THE LOTTERY PROTOCOL

As discussed in Section 1, as an example of an application of the "MPCs on Bitcoin" concept, we construct a protocol for a lottery executed among two parties: Alice (A) and Bob (B). We say that a protocol is a *fair lottery protocol* if it is *correct* and *secure*.

To define correctness assume that both parties are following the protocol and the communication channel between them is secure (i.e., it reliably transmits the messages between the parties without delay). We assume also that before the protocol starts, the parties have enough funds to play the lottery, including both their stakes (for simplicity we assume that the stakes are equal 1฿) and the money for deposits, because in the protocol we will use the commitment scheme from Section 3. If these assumptions hold, a correct protocol must ensure that at the end of the protocol one party, chosen with uniform probability, has to get the whole pot consisting of both stakes and the other party loses her stake. Additionally, both parties have to get their deposits back.

To define security, look at the execution of the protocol from the point of view of one party, say A (the case of the other party is symmetric) assuming that she is honest. Obviously, A has no guarantee that the protocol will terminate successfully, as the other party can leave the protocol before it is completed. What is important is that A should be sure that she will not lose money because of this termination, for example, the other party should not be allowed to terminate the protocol after he learned that A won. This is formalized as follows: we define the *payoff of* A in the execution of the protocol to be equal to the difference between the money that A invested and the money that she has after the execution of the protocol. We say that the protocol is *secure* if for any strategy of an adversary that controls the network and corrupts one party, the expected payoff of the other, honest party is not negative. We also note that, of course, a dishonest participant can always terminate at a very early stage when she does not know who is the winner—it does not change the payoff of the honest party.

4.1. The protocol

Our protocol is built on top of the classical coin-tossing protocol of Blum[6] described in Section 1. As already mentioned, this protocol does not directly work for our application, so 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. After choosing a random bit b_P, the party $P \in \{A, B\}$ chooses a string s_P sampled uniformly random from $\{0,1\}^{128+b_P}$, that is, the set of strings of length 128 or 129 bits, according to the value of b_P. Party P then commits to s_P using a timed commitment. The winner is determined by the *winner choosing function f*, defined as follows: $f(s_A, s_B) = A$ if $|s_A| = |s_B|$ and B, otherwise, where s_A and s_B are the secret strings chosen by the parties and $|s_P|$ is the length of s_P in bits. It is easy to see that as long as one of the parties draws their bit b_P uniformly, then the output of $f(s_A, s_B)$ is also uniformly random (provided the parties can only choose the strings s_A and s_B to be of length 128 or 129).

First attempt. 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. 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) as mentioned earlier and they exchange the hashes $h_A = H(s_A)$ and $h_B = H(s_B)$. If $h_A = h_B$, then the players abort the protocol.[d] Both parties broadcast their input transactions and send to the other party the links to their appearance in the block chain. If at any point later a party $P \in \{A, B\}$ realizes that the other party is cheating, then the first thing P will do is to "take the money and run," that is, post a transaction that redeems the input transaction. We will call it "halting the execution." This can clearly be done as long as the input transaction has not been redeemed by some other transaction. In the next step, one of the parties constructs a transaction *Compute* defined as follows:

Compute	
in-script$_1$: A's signature	**in-script**$_2$: B's signature
out-script: can be spent using: (1) strings x_A and x_B of length 128 or 129 s.t. $H(x_A) = h_A$, $H(x_B) = h_B$ **and** (2) X's signature, where X is the winner (i.e., $X = f(x_A, x_B)$)	

[d] We would like to thank Iddo Bentov and Ranjit Kumaresan, and independently David Wagner, for pointing out to us that this step is needed. It protects from the *copy attack*: A waits until B commits with his hash h_B and then she commits with the same hash. During the opening phase, A again waits until B reveals his secret s_B and then she reveals the same secret. By doing this A always wins since $f(s_A, s_B) = A$.

Note that the body of *Compute* can be computed from the publicly available information. Hence, this construction can be implemented as follows: first one of the players, say, Bob computes the body of *Compute* and sends his signature on it to Alice. Alice computes the body, adds both signatures to it, and broadcasts the entire transaction *Compute*.

The output script of *Compute* is tricky. To make it evaluate to true on *body*, one needs to provide as "witnesses" the signature of a party P and strings x_A, x_B, where x_A and x_B are the preimages of h_A and h_B (with respect to H). The collision-resistance of H implies that x_A and x_B have to be equal to s_A and s_B (resp.). Hence, it can be satisfied only if the winner choosing function f evaluates to P on input (s_A, s_B). Since only party P knows her private key, only she can later provide a signature that would make the output script evaluate to true.

Before *Compute* appears on the block chain, each party P can "change her mind" and redeem her input transaction, which would make the transaction *Compute* 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 the transaction *Compute* becomes confirmed 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 $f(s_A, s_B) = \mathsf{A}$, then Alice can redeem the transaction *Compute* in a transaction $ClaimMoney_A$ constructed as:

$ClaimMoney_A$
in-script: strings s_A and s_B and A's signature
out-script: can be spent only by A

On the other hand, Bob cannot redeem *Compute*, as the condition $f(s_A, s_B) = \mathsf{B}$ evaluates to false. Symmetrically: if $f(s_A, s_B) = \mathsf{B}$, then only Bob can redeem *Compute* by an analogous transaction $ClaimMoney_B$.

This protocol is obviously correct. It may also look secure, as it is essentially identical to Blum's protocol described before (with a hash function used as the commitment scheme). Unfortunately, it suffers from the following problem: there is no way to guarantee that the parties always reveal 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 $f(s_A, s_B) = \mathsf{A}$). As his money is already "gone" (his input transaction has already been redeemed in transaction *Compute*), he cannot gain anything, but he might 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 3.

The secure version of the scheme. The general idea behind the SecureLottery protocol is that each party first commits to her inputs, using the Bitcoin-based timed commitment scheme, instead of the standard commitment scheme. Recall that the CS protocol can be opened by sending a value s, and this opening is verified by checking that s has required length (either 128 or 129) and 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 receiver. 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 receiver, 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 broadcasts an input transaction. Once these transactions are confirmed, they create the *Compute* transaction in the same way as before, and once it appears on the block chain, they open the commitments. The only difference is that, since they used the CS commitment scheme, they can now "punish" the other party if she did not open her commitment by the time t and claim their deposit. On the other hand, each honest party is always guaranteed to get her deposit back, hence she does not risk anything by investing this money at the beginning of the protocol. The graph of transactions in this protocol is presented in Figure 2.

We also need to comment about the choice of the parameters: t: the time when deposit become available to the receiver and d: the value of the deposit. Our protocol consists of four "rounds" of transactions—in each round, parties wait for the confirmation of all the transactions from this round before proceeding to the next round. Thus, the correct execution of the protocol always terminates within time $4 \cdot T_{\max}$, where $T_{\max}$ is the maximal time needed for a transaction to be confirmed. Because of that we can safely set t to be the start time of the protocol plus $5 \cdot T_{\max}$.

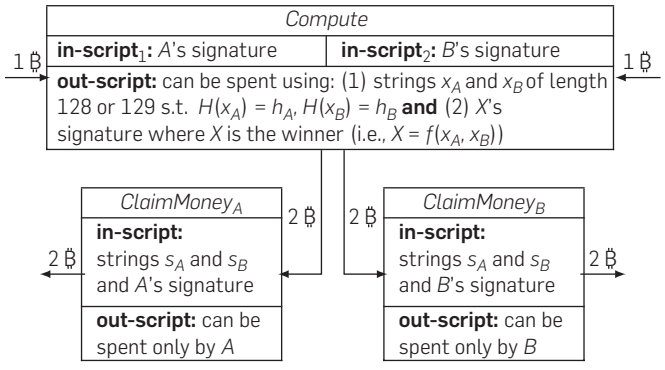
The parameter d should be chosen in such a way that it will fully compensate to each party the fact that the other player aborted. That means that for a two-player lottery, each player should make a deposit equal to two stakes. This way if one party aborts the protocol, then the other party may lose her stake worth 1 ฿, but she gets a deposit of value 2 ฿, so as a result of the protocol executions she earns 1 ฿, what is never worse for her than executing the protocol to the very end.

The complete description of this protocol can be found in the extended version of this paper, where we also show how to generalize it to N parties. In our multiparty solution, the total amount of money invested in the 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.

Acknowledgments

We would like to thank Iddo Bentov and Ranjit Kumaresan for fruitful discussions and for pointing out an error in a

Figure 2. The SecureLottery protocol.



previous version of our lottery. We are also very grateful to David Wagner for carefully reading our paper and for several useful remarks.

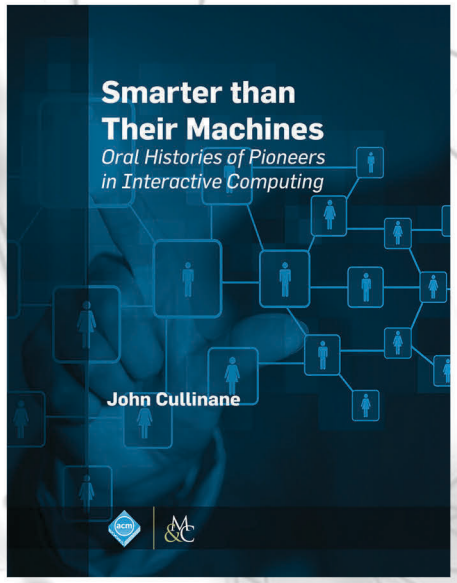
References

1. Andrychowicz, M., Dziembowski, S., Malinowski, D., Mazurek, Ł. Fair two-party computations via bitcoin deposits. In *1st Workshop on Bitcoin Research* (Christ Church, Barbados, March 7, 2014), Springer, Berlin, Germany, 105–121.
2. Andrychowicz, M., Dziembowski, S., Malinowski, D., Mazurek, Ł. On the malleability of bitcoin transactions. In *2nd Workshop on Bitcoin* Research (San Juan, Puerto Rico, January 30, 2015), Springer, Berlin, Germany.
3. Back, A., Bentov, I. Note on fair coin toss via bitcoin, 2013. http://www.cs.technion.ac.il/~idddo/cointossBitcoin.pdf.
4. Ben-David, A., Nisan, N., Pinkas, B. FairplayMP: A system for secure multi-party computation. In *ACM CCS 08: 15th Conference on Computer and Communications Security* (Alexandria, VA, October 27–31, 2008), ACM, NY, 257–266.
5. Bentov, I., Kumaresan, R. How to use bitcoin to design fair protocols. In *Advances in Cryptology – CRYPTO*, 2014. Part II (Santa Barbara, CA, August 17–21, 2014), Springer, Berlin, Germany, 421–439.
6. Blum, M. Coin flipping by telephone. In *Advances in Cryptology – CRYPTO'81* (Santa Barbara, CA, 1981), U.C. Santa Barbara, Department of Electrical and Computer Engineering, 11–15.
7. Bogetoft, P., et al. Secure multiparty computation goes live. In *FC 2009: 13th International Conference on Financial Cryptography and Data Security* (Accra Beach, Barbados, February 23–26, 2009), Springer, Berlin, Germany, 325–343.
8. Boneh, D., Naor, M. Timed commitments. In *Advances in Cryptology – CRYPTO 2000* (Santa Barbara, CA, August 20–24, 2000), Springer, Berlin, Germany, 236–254.
9. Cleve, R. Limits on the security of coin flips when half the processors are faulty. In *Proceedings of the 18th Annual ACM Symposium on Theory of Computing*, STOC '86 (Berkeley, CA, May 28–30, 1986), ACM, NY, 364–369.
10. Damgård, I., et al. Practical covertly secure MPC for dishonest majority — Or: Breaking the SPDZ limits. In *ESORICS 2013: 18th European Symposium on Research in Computer Security* (Egham, UK, September 9–13, 2013), Springer, Berlin, Germany, 1–18.
11. Douceur, J.R. The sybil attack. In *First International Workshop on Peer-to-Peer Systems*, IPTPS '01, 2002.
12. Friedman, E.J., Resnick, P. The social cost of cheap pseudonyms. *J. Econ. Manage. Strat. 10* (2000), 173–199.
13. Garay, J.A., Jakobsson, M. Timed release of standard digital signatures. In *FC 2002: 6th International Conference on Financial Cryptography* (Southampton, Bermuda, March 11–14, 2003), Springer, Berlin, Germany, 168–182.
14. Goldreich, O., Micali, S., Wigderson, A. How to play any mental game or A completeness theorem for protocols with honest majority. In *19th Annual ACM Symposium on Theory of* Computing (New York City, NY, May 25–27, 1987), ACM, NY, 218–229.
15. Kumaresan, R., Bentov, I. How to use bitcoin to incentivize correct computations. In *ACM CCS 2014* (Scottsdale, AZ, November 3–7, 2014), ACM, NY, 30–41.
16. Malkhi, D., Nisan, N., Pinkas, B., Sella, Y. Fairplay – A secure two-party computation system. In *13th Conference on USENIX Security Symposium*, SSYM'04 (San Diego, CA, August 9–13, 2004), USENIX Association, 287–302.
17. Nakamoto, S. Bitcoin: A peer-to-peer electronic cash system. The Cryptography Mailing List, 2008.
18. Post, T.W. Cheating scandals raise new questions about honesty, security of internet gambling. *The Washington Post* November 30, 2008.
19. Resnick, P., Kuwabara, K., Zeckhauser, R., Friedman, E. Reputation systems. *Commun. ACM 43*, 12 (Dec. 2000) 45–48
20. Yao, A.C.-C. How to generate and exchange secrets (extended abstract). In *27th Annual Symposium on Foundations of Computer Science* (Toronto, ON, Canada, October 27–29, 1986), IEEE Computer Society Press, 162–167.

Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Łukasz Mazurek
({marcin.andrychowicz, stefan.dziembowski, daniel.malinowski, lukasz.mazurek}@crypto.edu.pl), Institute of Informatics, University of Warsaw, Warsaw, Poland.

Copyright held by authors. Publication rights licensed to ACM. $15.00.

Watch the author discuss his work in this exclusive *Communications* video. http://cacm.acm.org/videos/secure-multiparty-computations-on-bitcoin



A personal walk down the computer industry road. BY AN EYEWITNESS.

Smarter Than Their Machines: Oral Histories of the Pioneers of Interactive Computing is based on oral histories archived at the Charles Babbage Institute, University of Minnesota. These oral histories contain important messages for our leaders of today, at all levels, including that government, industry, and academia can accomplish great things when working together in an effective way.

ISBN: 978-1-62705-550-5 DOI: 110.1145/2663015
http://books.acm.org
http://www.morganclaypoolpublishers.com/acm