University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Marcin Andrychowicz

Multiparty Computation Protocols Based on Cryptocurrencies

PhD dissertation

Supervisor
dr hab. Stefan Dziembowski, prof. UW
Institute of Informatics
University of Warsaw

May 2015

Author's declaration:
I hereby declare that this dissertation is my own work.

May 28, 2015 .....

Marcin Andrychowicz

Supervisor's declaration:
The dissertation is ready to be reviewed.

May 28, 2015 .....

dr hab. Stefan Dziembowski, prof. UW

Abstract

In this dissertation we show how to use cryptocurrencies (mostly Bitcoin) and techniques coming from the field of cryptocurrencies to construct new types of Multiparty Computation Protocols (MPC), which go beyond the standard definition of MPC.

In the first part of the dissertation we study the MPC protocols, in which parties have access to a Bitcoin-like cryptocurrency. We show that in this setting it is possible to design a two-party MPC protocol for computing a value of an arbitrary function f that has two advantages over classical MPC protocols. First, classical two-party MPC protocols do not guarantee *fairness*, which means that a malicious party can interrupt the execution of a protocol in a state, in which he/she knows the output of the protocol, but the other party does not. In contrast, our MPC protocol do guarantee some kind of fairness. More precisely, a party who has not learnt the output due to the other party's misbehaviour is given a financial compensation. Moreover, the value of the function f being computed can contain instructions like "The party A pays 1 bitcoin to the party B" and our protocol guarantees that such transfers will automatically take place. Therefore, it allows to create two-party protocols with *financial consequences*. We also discuss how to formally verify the correctness and the security of such protocols and show how to protect them against the so-called *malleability* of transactions, which is a problem concerning most cryptocurrencies.

Then, we study cryptographic protocols in a fully peer-to-peer scenario under the assumption that the computing power of the adversary is limited and there is no trusted setup (like a PKI or an unpredictable beacon). We show that every primitive (e.g. Byzantine consensus) that can be realised under the classical assumption that the majority of the parties executing the protocol is honest and there is a PKI available can also be realised under the assumption that the majority of the computing power is controlled by honest parties and there is *no* PKI.

Keywords

multiparty computation protocol, cryptocurrency, Bitcoin, proof-of-work

ACM Computing Classification

Theory of computation, Cryptographic protocols

Streszczenie

Niniejsza praca przedstawia wykorzystanie walut kryptograficznych (głównie Bitcoina) oraz technik wywodzących się z walut kryptograficznych do konstrukcji protokołów do obliczeń wielopodmiotowych (ang. Multiparty Computation Protocols, MPC), które wychodzą poza standardową definicję bezpieczeństwa protokołów MPC.

W pierwszej części pracy analizujemy protokoły MPC, w których strony mają dostęp do kryptowaluty w stylu Bitcoina. Pokazujemy przy tym założeniu konstrukcję protokołu MPC dla dwóch stron do obliczania wartości dowolnej funkcji f, który ma dwie zalety w stosunku do klasycznych protokołów MPC. Po pierwsze, klasyczne protokoły MPC dla dwóch stron nie gwarantują tzw. *sprawiedliwości obliczeń* (ang. fairness), co oznacza, że nieuczciwa strona może przerwać wykonanie protokołu w stanie, w którym jedynie ona zna wynik obliczenia. W odróżnieniu od standardowych protokołów MPC, nasz protokół zapewnia pewien rodzaj sprawiedliwości obliczeń. Dokładniej, strona, która nie pozna wyniku obliczenia, automatycznie otrzymuje rekompensatę finansową. Ponadto, wartość funkcji f może zawierać instrukcje w stylu "Strona A płaci 1 bitcoina stronie B" i nasz protokół gwarantuje, że takie przelewy zostaną automatycznie wykonane. Pozwala to zatem na konstrukcje protokołów z *konsekwencjami finansowymi*. W pracy przedstawiamy również metody formalnej weryfikacji poprawności i bezpieczeństwa tego typu protokołów oraz pokazujemy w jaki sposób chronić te protokoły przed tzw. *kowalnością* (ang. malleability) transakcji, która jest problemem występującym w większości kryptowalut.

Ponadto analizujemy protokoły kryptograficzne w sieciach peer-to-peer, które nie wymagają infrastruktury klucza publicznego ani źródła skorelowanej losowości (ang. random beacon) przy założeniu ograniczonej mocy obliczeniowej przeciwnika. Pokazujemy, że każda funkcjonalność, która może zostać zrealizowana przy założeniu, że większość stron wykonujących protokół jest uczciwa i dostępna jest infrastruktura klucza publicznego może zostać również zrealizowana przy założeniu, że większość *mocy obliczeniowej* jest uczciwa bez założenia o istnieniu infrastruktury klucza publicznego.

Słowa kluczowe

protokół do obliczeń wielopodmiotowych, kryptowaluta, Bitcoin

Klasyfikacja tematyczna ACM

Theory of computation, Cryptographic protocols

Multiparty Computation Protocols Based on Cryptocurrencies*

PhD dissertation summary

Marcin Andrychowicz

May 27, 2015

1 Introduction

My PhD dissertation consists of 6 articles, which I wrote during my PhD studies at the Faculty of Mathematics, Informatics and Mechanics at the University of Warsaw:

1. **Secure Multiparty Computations on Bitcoin** [1], accepted to *IEEE Symposium on Security and Privacy*, 2014,
2. **Fair Two-Party Computations via the Bitcoin Deposits** [2], accepted to *First Workshop on Bitcoin Research (in Assocation with Financial Crypto)*, 2014,
3. **Modeling Bitcoin Contracts by Timed Automata** [3], accepted to *Formal Modeling and Analysis of Timed Systems (FORMATS)*, 2014,
4. **On the Malleability of Bitcoin Transactions** [4], accepted to *Second Workshop on Bitcoin Research (in Assocation with Financial Crypto)*, 2015,
5. **Secure Multiparty Computations on Bitcoin** [5], accepted to *Communications of the ACM (CACM), Research Highlights*, 2015,

*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.

6. **PoW-Based Distributed Cryptography with no Trusted Setup** [6], accepted to *International Cryptology Conference (CRYPTO)*, 2015.

The first 5 articles were written together with my supervisor dr hab. Stefan Dziembowski and two fellow PhD students: Daniel Malinowski and Łukasz Mazurek. The last paper was co-authored only by dr hab. Dziembowski. All of the above papers concern using cryptocurrencies (mainly Bitcoin) or techniques coming from cryptocurrencies for designing new types of Multiparty Computation Protocols (MPC), which go beyond the standard definition of MPC.

A brief introduction to MPC protocols is given in Sec. 2 and a short description of Bitcoin and other cryptocurrencies is presented in Sec. 3. Then, in Sec. 4, I describe the results published in the papers included in this dissertation. Finally, in Sec. 5, I list the publications written during my PhD studies, which are not part of this dissertation.

2 Multiparty Computation Protocols

Multiparty Computation Protocols (MPC, [7, 8]) are protocols which allow a group of mutually distrusting parties to jointly compute a value of an arbitrary function f over their private and secret inputs without disclosing the values of the inputs. More precisely, the security od means that no party (even the one which does not follow the protocol) cannot learn more about the other parties' inputs than it could learn in the *ideal model* in which (a) all parties send their inputs to the trusted third party; (b) the trusted third party computes the value of the function f; and (c) the trusted third party sends the value of the function to all the parties.

As an illustration of what the security of MPC protocols means consider the so-called *marriage problem*. In this problem, two parties called Alice and Bob want to check if they *both* would like to marry each other. Alice's input is a bit a indicating whether she would like to marry Bob and Bob's input b indicates whether he would like to marry Alice. The output of the protocol is the bit indicating whether *both* parties agree to the marriage, so the function f being computed is the conjunction: $f(a, b) = a \wedge b$. Notice that if $a = 1$ then Alice can easily infer b from the output of the protocol as $f(a, b) = a \wedge b = b$. The secrecy in this scenario means that if Alice does not want to marry Bob (i.e. $a = 0$), then she should not learn Bob's input b and the analogical condition should hold for Bob.

Despite very strong security guarantees given by such protocols they are very rarely used in practice. One of the reasons for this situation is the fact that, in general, such protocols do not guarantee the so-called *fairness*, which

means that a malicious party can terminate the execution of a protocol in a state, in which it is the only party knowing the result of the execution. It has been proven that fairness cannot be guaranteed without the assumption that the majority of parties are honest. In particular, MPC protocols do not guarantee fairness in the case when there are only two parties (unless we assume that both parties are honest).

3 Cryptocurrencies

Cryptographic currencies (also called cryptocurrencies) are equivalents of regular money which can be used over the internet. *Bitcoin* [9] is a cryptocurrency introduced in 2008 by an anonymous developer using the pseudonym "Satoshi Nakamoto". Despite its mysterious origins, Bitcoin became the first widely adopted cryptocurrency — its current market capitalisation exceeds $3 bn (May 2015). The most pivotal feature of Bitcoin, which makes it different from regular currencies, is that there is no central authority, which controls the system or could "print" more money. Moreover, Bitcoin has very low transaction fees and provides some level of anonimity.

The rapid success of Bitcoin caused the emergence of many other cryptographic currencies (called *alternative cryptocurrencies* or *altcoins* to distinguish them from Bitcoin). Most of altcoins are just clones of Bitcoin and do not differ from it significantly. Therefore, although I concentrate in this dissertation on Bitcoin as the most popular cryptocurrency, the proposed line of research is universal and concerns the general ideas behind most of cryptocurrencies.

Bitcoin is an extremely interesting topic from the perspective of cryptography, but for the lack of space in this short summary I will only mention the features of Bitcoin which are most relevant to my work. A more detailed introduction to Bitcoin may be found in my papers, e.g. [5].

In Bitcoin, all users of the system are connected using a peer-to-peer network that jointly maintains a public ledger (called *blockchain*) containing all transactions which have taken place in the system. Recall that classical MPC protocols do not guarantee fairness unless the majority of users are honest. The assumption of the honest majority does not make much sense in the internet, where a malicious party can create numerous fake identities (the so-called *Sybil attack*).

Bitcoin overcomes this limitation by using the so-called *Proofs-of-Work (PoW)*, which are cryptographic primitives allowing a party to prove to other parties that it has spent some computational effort on a given task. More precisely, the security of the Bitcoin protocol relies on the assumption that

the majority of the *computing power* in the system belongs to honest parties. Therefore, in order to break the system the adversary would need to control machines with the computing power comparable to the computing power of the rest of the network.

As already mentioned the blockchain jointly maintained by the users contains all transactions that have taken place in the system since its inception. Without digging into the details, we can say a Bitcoin transaction has the form "A user A transfers x bitcoins to a user B" (bitcoin is a currency unit used in Bitcoin). Of course, we need to make sure that the user A has enough funds to make such a transfer. To this end, each transaction has a special string identifying it (it is, in fact, the hash of its binary representation) and has, in fact, the form "A user A transfers x bitcoins, which he/she received in the transaction T, to a user B". Such a transaction is accepted only if the blockchain contains the transaction T and the transaction T transfers x bitcoins to the user A. In this case, we say that the new transaction *claims* the transaction T. Of course, we also need to make sure that a malicious user cannot spend his coins twice (called a *double-spending attack*) and hence the new transaction is only accepted if the blockchain does not already contain another transaction claiming the transaction T.

The ownership of bitcoins is proven using public key cryptography (each user can create a private/public key pair on its own — there is no PKI used). More precisely, the users of the system are identified by their public keys and each transaction contains the public key of the recipient. The transaction claiming the transaction T is considered valid only if it is signed using a private key corresponding to the public key of the recipient of the transaction T.

What is important from the perspective of using Bitcoin for designing MPC protocols is that it allows much more complicated types of transactions (called *non-standard* transactions). Such transactions can contain a puzzle, which the recipient has to solve in order to claim the bitcoins being sent. In more detail, each transaction contains the so-called *output script*, which is a description of some function f. In order to claim the money the recipient has to provide the argument x for which $f(x) = \mathbf{true}$. The function f is written in a special language designed for this purpose, which allows among others verifying cryptographic signatures or computing the values of some hash functions. Moreover, transactions can be time-locked, which means that they become valid only after some time t. For instance, it is possible to send bitcoins to three users in such a way that any two of them can claim the bitcoins after July 1st, 2015 assuming that they will provide the string x, s.t. $H(x) = v$, where H is the SHA-256 hash function and v is some string.

4 Obtained results

4.1 Secure online gambling

Online gambling is a significant market — it is estimated that there are currently 1,700 gambling sites worldwide handling bets worth over $4 bn per year [10]. What may be surprising is that online gamblers are completely unprotected from the abuse by casinos. The casino can easily cheat on its users, e.g. by "rolling biased dices" or simply refusing to pay them the prizes they are entitled to and there have been cases when some casinos have overused their privileged position.

It raises the question whether MPC protocols can be used for online gambling to guarantee the users that the game will be fair. For the sake of simplicity, consider a case of two parties (one of them may, for example, be an online casino) that do not trust each other and would like to bet $1 on a toss of a virtual coin — i.e. each party pays $1 and a party chosen uniformly at random receives $2. There are MPC protocols which allow to draw a uniformly random bit, but can we use them in this scenario? There are two limitations of MPC protocols, which make them unsuitable for this situation. First of all, two-party MPC protocols do not provide *fairness* that means that a malicious party could interrupt a protocol in a state, in which it knows who the winner is, but the other party does not. The second problem is that, even if the coin-tossing was performed successfully and both parties know its result, there is no way to guarantee that the loser will pay the agreed $1 to the winner.

In our first paper [1] (and a short journal article [5] based on it) we showed that it is possible to overcome both of the aforementioned problems assuming that the parties have access to the Bitcoin infrastructure. Namely, we presented the protocol for an arbitrary number of parties that allows them to bet on a toss of a virtual coin. The protocol can be executed in a peer-to-peer environment between anonymous parties who do not trust each other. It enjoys a very strong security guarantee — no matter how the dishonest parties behave, the honest parties will never get cheated on (regardless of the number of malicious parties). More precisely, each honest party can be sure that, once the game starts, it will always terminate and be fair.

The pivotal idea behind the protocol is as follows. At the beginning of the protocol, each party makes a special *deposit* in bitcoins. This deposit is a non-standard transaction crafted in such a way that the party will be able to claim the deposit back if it follows the protocol to the very end regardless of the behaviour of the other parties. If a malicious party tries to cheat, then this deposit will automatically be given to the other parties and will

compensate them for what they could have lost because of the malicious party's misbehaviour.

In [1], not only have we presented the gambling protocol, but we have also introduced a new computational paradigm, in which MPC protocol is executed *on top of* Bitcoin.

4.2 Fair MPC protocols with financial consequences

In our second paper [2], we generalized our results from [1] and presented the two-party MPC protocol for an arbitrary function f, which has two advantages over the classical MPC protocols. The first one is that it provides some kind of *fairness*. More precisely, the parties make bitcoin deposits at the beginning of the protocol. If the parties follow the protocol, they both learn the value of the function f and get their deposits back. However, if one of the parties misbehaves (i.e. interrupts the protocol prematurely or sends an incorrect message), then the other party gets both deposits. In other words, the party that does not learn the result of the computation due to the other party's misbehaviour is given a financial compensation in bitcoins. Hence, this protocol achieves fairness if only the value of the deposit is greater than the loss from a premature interruption of the protocol. Moreover, this protocol does not require any trust and making such deposits is completely safe — a party that follows the protocol is guaranteed to get its deposit back regardless of the other party's behavior.

The second advantage of our protocol is that it allows the result of the function f to include instructions like "Alice pays 5 bitcoins to Bob" and our protocol enforces that such transfers are indeed performed. This happens only if the protocol is carried out to the very end, but again, if one of the parties interrupts the protocol, then its deposit is given to the honest participant. One of the possible applications of this feature is trading digital goods. What is surprising, our protocol allows to trade information in such a way that even a seller does not know exactly what they are selling. The details may be found in our paper.

Our protocol from [2] required a small Bitcoin protocol modification related to the problem of *malleability* of Bitcoin transactions, which is discussed in more detail in the next section.

4.3 Malleability of Bitcoin transactions

Bitcoin transactions are *malleable*, which means that knowing a transaction T everyone can compute a transaction T' that is semantically equivalent to T (it has the same sender, recipient and amount of bitcoins being transferred

as T), but has a different binary representation. This can pose serious problems, because Bitcoin infrastructure tracks transactions using hashes of their binary representations and transactions are broadcast in the peer-to-peer network. In [4] we performed an experimental evaluation of how serious this problem is. We showed that it is relatively easy to perform the "malleability attack", in which an adversary changes the binary representation of a transaction broadcast by some other user. We evaluated the behavior of most of the available Bitcoin wallets (i.e. programs used to send bitcoins) under this attack and discovered a number of serious bugs in the Bitcoin software. Such bugs could cause a state, in which it is impossible (at least temporarily) to make any bitcoin transfers, which means that a user can effectively lose all his bitcoins.

Moreover, the malleability of transactions can also pose problems to Bitcoin contracts. In [4] we also showed a general way of protecting Bitcoin contracts agains malleability attacks, which allows among others to use our protocol from [2] in the current version of the Bitcoin protocol.

4.4 Formal verification of Bitcoin contracts

Protocols working on top of Bitcoin (also called *Bitcoin contracts*) are tricky to write and analyze due to the distributed nature of the block chain and a huge number of possible interleavings. Moreover, the flaws in designing such protocols can be exploited by malicious parties for their financial gain. Therefore, it is an important question whether we can use formal verification to enhance our confidence in the security of the created protocols.

In [3] we presented a framework for modeling Bitcoin contracts using timed automata. The presented method is general and can be used to formally verify the correctness and the security of almost arbitrary Bitcoin contracts. The framework allows to formally verify the statements like "As a result of the protocol execution the party gains one bitcoin or learns a string s satisfying some condition $C(s)$ regardless of the other party's behavior". As a proof-of-concept we used this framework to formally verify the security of two Bitcoin contracts from our previous papers [1, 4] in the UPPAAL Model Checker [11].

4.5 PoW-based distributed cryptography

The approach described so far was to design MPC protocols, in which parties have access to some cryptocurrency. There is, however, a completely different way of enhancing MPC protocols with techniques coming from the field of cryptocurrencies. One of the innovations of cryptocurrencies is that they

operate under the assumption that the majority of the computing power is controlled by honest parties, where the computing power is measured by the number of times a party can compute a value of some function (e.g. SHA-256) in a period of time.

In [6] we studied cryptographic protocols in a fully peer-to-peer scenario under the assumption that the computing power of the adversary is limited and there is no trusted setup (like a PKI or an unpredictable beacon; we assume that even the number of parties executing the protocol is unknown to the participants). We propose a formal model for this scenario and construct two protocols in it.

The first one is a *reliable broadcast* protocol, which operates under the assumption that there is a known bound π on the total computing power of all the parties and that the computing power of the honest parties is a non-negligible fraction of π.

The second protocol requires that the majority of the computing power is controlled by the honest parties and allows to compute the subset of the parties G, s.t.: (1) all parties agree on the set G (consensus); (2) all honest parties belong to G; (3) the majority of the parties in the set G is honest; and (4) each party knows the public key of each party in the set G. Therefore, this shows that every primitive (e.g. Byzantine consensus) that can be realised under the classical assumption that the majority of the parties executing the protocol is honest and there is a PKI available (i.e. each party knows the public key of each other party) can also be realised under the assumption that the majority of the computing power is controlled by honest parties and there is *no* PKI.

5 Other publications

During my PhD studies I also published the following two papers, which I have decided not to include in this dissertation, because they concern neither MPC protocols, nor cryptocurrencies:

1. **Efficient Leakage Resilient Circuit Compilers**, accepted to *RSA Conference Cryptographers' Track (CT-RSA)*, M. Andrychowicz, I. Damgaard, S. Dziembowski, S. Faust, A. Polychroniadou, 2015,

2. **Leakage-Resilient Cryptography over Large Finite Fields: Theory and Practice**, accepted to *International Conference on Applied Cryptography and Network Security (ACNS)*, M. Andrychowicz, D. Masny, E. Persichetti, 2015.

Acknowledgements

First of all, I would like to thank my supervisor, dr hab. Stefan Dziembowski for guiding me during my PhD studies, co-authoring all publications included in this dissertation and introducing me to a great selection of research problems.

I would also like to thank all other people I worked with during my PhD studies, i.e. Sebastian Faust, Daniel Malinowski, Daniel Masny, Łukasz Mazurek, Edoardo Persichetti and Antigoni Polychroniadou.

And last but not least, I would like to thank my wife for her endless support.

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.

References

[1] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Łukasz Mazurek. Secure multiparty computations on bitcoin. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 443–458. IEEE, 2014.

[2] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Łukasz Mazurek. Fair two-party computations via bitcoin deposits. In *Financial Cryptography and Data Security*, pages 105–121. Springer, 2014.

[3] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Łukasz Mazurek. Modeling bitcoin contracts by timed automata. In *Formal Modeling and Analysis of Timed Systems*, pages 7–22. Springer, 2014.

[4] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Łukasz Mazurek. On the malleability of bitcoin transactions. In *Workshop on Bitcoin Research*, 2015.

[5] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Łukasz Mazurek. Secure multiparty computations on bitcoin. In *Communications of the ACM, Reserach Highlights*. ACM, 2015.

[6] Marcin Andrychowicz and Stefan Dziembowski. Pow-based distributed cryptography with no trusted setup. In *International Cryptology Conference*. Springer, 2015.

[7] A. C.-C. Yao. How to generate and exchange secrets. In *FOCS*, 1986.

[8] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. STOC, 1987.

[9] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

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

[11] Gerd Behrmann, Re David, and Kim G. Larsen. A tutorial on uppaal 4.0, 2006.

Secure Multiparty Computations on BitCoin

Marcin Andrychowicz*, Stefan Dziembowski†, Daniel Malinowski‡ and Łukasz Mazurek§
University of Warsaw

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 looser 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.

I. Introduction

Secure multiparty computation (MPC) protocols, originating from the seminal works of Yao [37] and Goldreich et al. [25], 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

*marcin.andrychowicz@crypto.edu.pl
†stefan.dziembowski@crypto.edu.pl
‡daniel.malinowski@crypto.edu.pl
§lukasz.mazurek@crypto.edu.pl

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* [6] 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* [35] 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 [25] 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 [16] 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 [26], [11], [4], [31]. 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 [28], [5], [17] and sometimes even to use them in the real-life applications such as, e.g., the online auctions [22]. 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. [32]). 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, hence 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 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 looser 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. [33] 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 [23].

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 looser 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 looser. 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 [13], or one of its variants [14], [12]. 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] [30]. BitCoin has recently gained a noticeable popularity (its current market capitalization is over 5 billion USD) 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

[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.

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 the pseudo-anonymity[3] — the users are only identified by their public keys that can be easily created, and hence it is hard to link the real person with the virtual party spending the money. 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 [30], the BitCoin wiki webpage en.bitcoin.it (sections particularly relevant to our work are: "Transactions" and "Contracts"), or other papers on BitCoin [29], [15], [2], [34]. In the sequel "Ƀ" 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" [7], [24], 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 payed 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).

[3] A very interesting modification of BitCoin that provides real cryptographic anonymity has been recently proposed in [29].

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 $0.00005\ddot{\text{B}} \approx 0.03\$$). 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 $1\,\ddot{\text{B}}$. 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 [6]. 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 reacher.[4] It is easy to see that Alice and Bob can (inefficiently) determine who has more coins 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

[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.

[5]The reader may be tempted to think that a similar protocol could be used with the eCash [13]. 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).

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, showing general completeness results, providing a more formal framework to analyze the deposit-based technique (this can probably be done using the tools from the "rational cryptography" literature [27], [1], [21]). An interesting research direction is also to construct protocols for games other than simple lotteries (like card, or board games). We also think that our protocols can provide a good motivation for research in the formal analysis of the BitCoin schemes that involve transactions with non-standard scripts. Is it possible to formally specify and verify the properties of such protocols in the spirit of [18]? Such formal approach would probably need to involve some elements of the temporal logic (see e.g. [3]) to capture the properties provided by the time-locks.

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 [20]. Some of these sites are using BitCoin. The largest of them, *SatoshiDice*, has been recently sold for over $12 million [10]. 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" [36]). 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 [9], 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% [9].

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. Related work

Most of the related work has been already described in the earlier sections. Previous papers on BitCoin analyze the BitCoin transaction graph [34], or suggest improvements of the current version of the system. This includes important work of Barber et al. [2] who study various security aspects of BitCoin and Miers et al. [29] 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)

The work most relevant to ours in the BitCoin literature is probably Section 7.1 of[2] where the authors construct a secure "mixer", that allows two parities to securely "mix" their coins in order to obtain unlinkability of the transactions. They also construct commitment schemes with time-locks, however some important details are different, in particular, in the normal execution of the scheme the money is at the end transferred to the receiver. Also, the main motivation of this work is different: the goal of [2] is to fix an existing problem in BitCoin ("linkability"), while our goal is to use BitCoin to perform tasks that are hard (or impossible) to perform by other methods.

Commitments in the context of the BitCoin were already considered in [15], however, the construction and its applications are different — the main idea of [15] 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.

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.

[6]Accessed on 13.11.2013.

The BitCoin ledger is implemented using the concept of the Proofs of Work (PoWs) [19] 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.

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 a signature σ on 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

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

[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.

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:

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

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:

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

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. 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 $x = H(T_x)$.

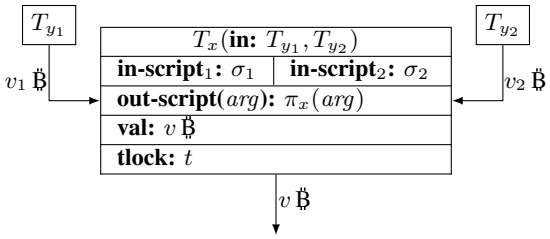
The main difference is, however, that in the real BitCoin the users have much more flexibility in defining the condition on how the transaction T_x can be redeemed. Consider for a moment the simplest transactions where there is just one input and no time-locks. Recall that in the simplified system described 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

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

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:

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

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 transaction 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

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

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.

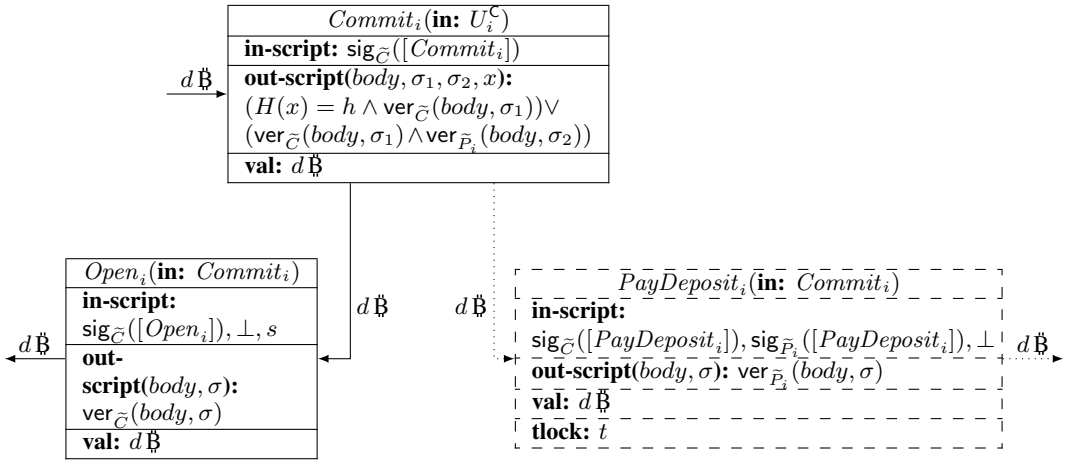
What is a bit less obvious is how to define privacy of the communication between the parties and the Ledger, especially the question of the privacy of the writing procedure. More precisely, the problem is that it is completely unreasonable to assume that a transaction is secret until it appears on the 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') can produce another transaction $\hat{T}$ whose semantics is the same is 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! 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 this assumption. 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.

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

We start with constructing a BitCoin-based timed-commitment scheme [7], [24]. Recall that a commitment scheme [6], [8] 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

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



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 sends each 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 $Open_i, \ldots, Open_n$, 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 $\bot$.

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 "splited" 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 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 commiter 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 $Commit_i$ 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

[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.

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 commuter 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 12). We now state the following lemma, whose proof appears in the full 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).

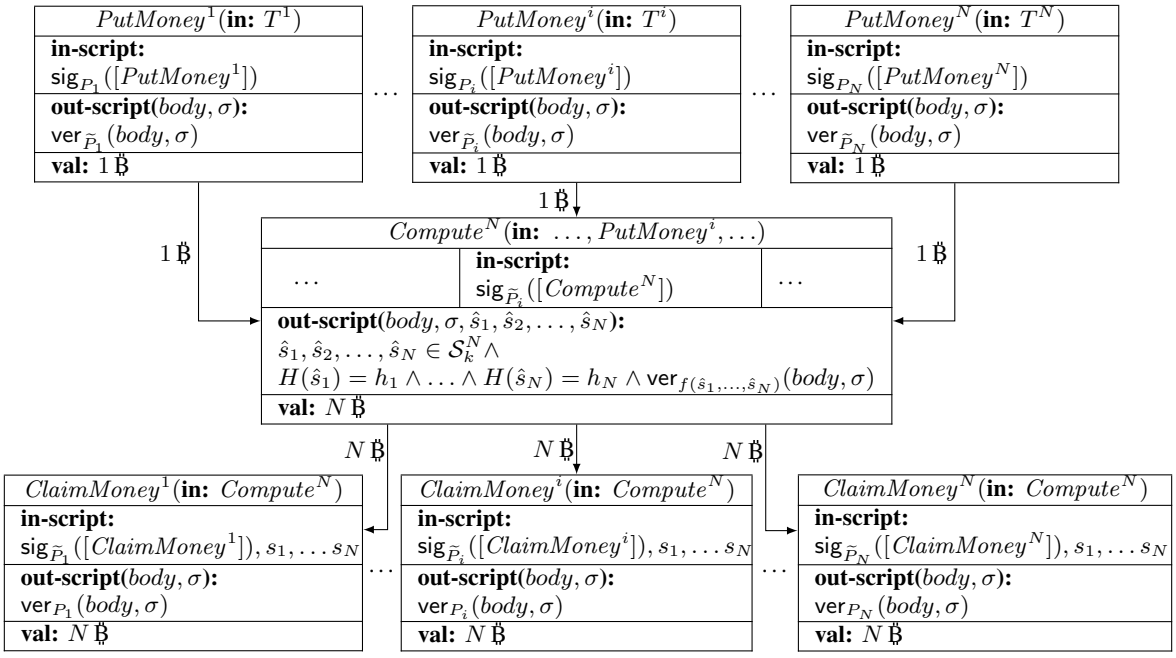
We assume that before the protocol starts, the Ledger contains unredeemed standard transactions $T^1, \ldots, T^N$ known to all the parties, all of value $1\,\ddot{\mathbb{B}}$ and each T^i can be redeemed with a key known only to P_i. Moreover, since we will use the commitment scheme from Section III, 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 $d\,\ddot{\mathbb{B}}$ (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 $N\,\ddot{\mathbb{B}}$ 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 $(N-1)d\,\ddot{\mathbb{B}}$. 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.

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 $T^1, U^1_2, \ldots, U^1_N$, whose recipient is P_i and whose value is: $1\,\ddot{\mathbb{B}}$ in case of T^1 and $d\,\ddot{\mathbb{B}}$ in case of the U^1_j'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 loose 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 $T^1, U^1_2, \ldots, U^1_N$) 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

[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.



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 have value 1 ₿ each and whose recipient is P_i. The Ledger contains also transactions $\{U_j^i\}$, where $i, j \in \{1, \ldots, N\}$ and $i \neq j$, such that each U_j^i can be redeemed by P^i and has value d ₿ (for some parameter d)

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.

Deposits phase:

5) Let t be the current time. For each $i = 1, \ldots, N$ 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.

Execution phase:

6) For each i, player P_i puts transaction $PutMoney^i$ to the Ledger. The players halt if any of those transactions did not appear on the Ledger before time $t + 2 \cdot \max_{\mathsf{Ledger}}$.
7) For each $i \geq 2$, player P_i computes his signature on transaction $Compute^N$ and sends it to the player P_1.
8) 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 Ledger in time $t + 3 \cdot \max_{\mathsf{Ledger}}$, then the players halt.
9) For each i, the player P_i puts his *Open* transactions on 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 revealed 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 ₿.
10) The player, that is the winner $\big($i.e.$P_{f(s_1,\ldots,s_N)}\big)$, gets the pot by sending the transaction $ClaimMoney^{f(s_1,\ldots,s_N)}$ to the Ledger.

Fig. 2. The MultiPlayersLottery protocol.

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 loose 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 [6] 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 SHA-256 hashing. Due to the technical limitations of BitCoin scripting language in its current form[15], instead of random numbers b_i, the parties commit themselves to strings s_i sampled with uniformly random from $\mathcal{S}_k^N := \{0,1\}^{8k} \cup \ldots \cup \{0,1\}^{8(k+N-1)}$, i.e. it is the set of strings of length $k, \ldots, (k+N-1)$ bytes[16], where k is the security parameter. The winner is determined by the *winner choosing function* $f(s_1, \ldots, s_N)$ for N players to be equal P_ℓ 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 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: $1\,฿$

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

[15]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.

[16]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.

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$(body, \sigma_1, \sigma_2, \hat{s}_{\mathsf{A}}, \hat{s}_{\mathsf{B}})$**:** $(\hat{s}_{\mathsf{A}}, \hat{s}_{\mathsf{B}} \in \mathcal{S}_k^2 \wedge H(\hat{s}_{\mathsf{A}}) = h_{\mathsf{A}} \wedge H(\hat{s}_{\mathsf{B}}) = h_{\mathsf{B}} \wedge f(\hat{s}_{\mathsf{A}}, \hat{s}_{\mathsf{B}}) = \mathsf{A} \wedge \mathsf{ver}_{\widetilde{A}}(body, \sigma_1)) \vee$ $(\hat{s}_{\mathsf{A}}, \hat{s}_{\mathsf{B}} \in \mathcal{S}_k^2 \wedge H(\hat{s}_{\mathsf{A}}) = h_{\mathsf{A}} \wedge H(\hat{s}_{\mathsf{B}}) = h_{\mathsf{B}} \wedge f(\hat{s}_{\mathsf{A}}, \hat{s}_{\mathsf{B}}) = \mathsf{B} \wedge \mathsf{ver}_{\widetilde{B}}(body, \sigma_2))$	
val: $2\,\ddot{\text{B}}$	

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 $\widetilde{A}$, hence only she can later provide a signature on 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 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: $2\,\ddot{\text{B}}$

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 T^{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 can not 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, which the corresponding variables 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 one 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 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.

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' is 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 payed 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 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 full 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 can not 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 not impossible. Technically, we do it by assuming that the channel between from each party to the Ledger is private. The protocols secure in this model will be called fair *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 Ƀ

The only difference compared to $PutMoney_1^{\mathsf{B}}$ is the addition of the "$\wedge(H(\hat{s}) = h_{\mathsf{B}})$" 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 Ƀ	

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[17]. 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$(body, \sigma_1, \sigma_2, \hat{s}_{\mathsf{A}}, \hat{s}_{\mathsf{B}})$**:** $(\hat{s}_{\mathsf{A}}, \hat{s}_{\mathsf{B}} \in \mathcal{S}_k^2 \wedge H(\hat{s}_{\mathsf{A}}) = h_{\mathsf{A}} \wedge H(\hat{s}_{\mathsf{B}}) = h_{\mathsf{B}} \wedge f(\hat{s}_{\mathsf{A}}, \hat{s}_{\mathsf{B}}) = \mathsf{A} \wedge \mathsf{ver}_{\widetilde{A}}(body, \sigma_1)) \vee$ $(\hat{s}_{\mathsf{A}}, \hat{s}_{\mathsf{B}} \in \mathcal{S}_k^2 \wedge H(\hat{s}_{\mathsf{A}}) = h_{\mathsf{A}} \wedge H(\hat{s}_{\mathsf{B}}) = h_{\mathsf{B}} \wedge f(\hat{s}_{\mathsf{A}}, \hat{s}_{\mathsf{B}}) = \mathsf{B} \wedge \mathsf{ver}_{\widetilde{B}}(body, \sigma_2)) \vee$ $(\mathsf{ver}_{\widetilde{A}}(body, \sigma_1) \wedge \mathsf{ver}_{\widetilde{B}}(body, \sigma_2))$	
val: $2\,\ddot{\mathbb{B}}$	

$Fuse$(**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: $2\,\ddot{\mathbb{B}}$
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 on his secret s_{B} which Alice should not learn at this point.

[17]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

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 $Compute$ at 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.

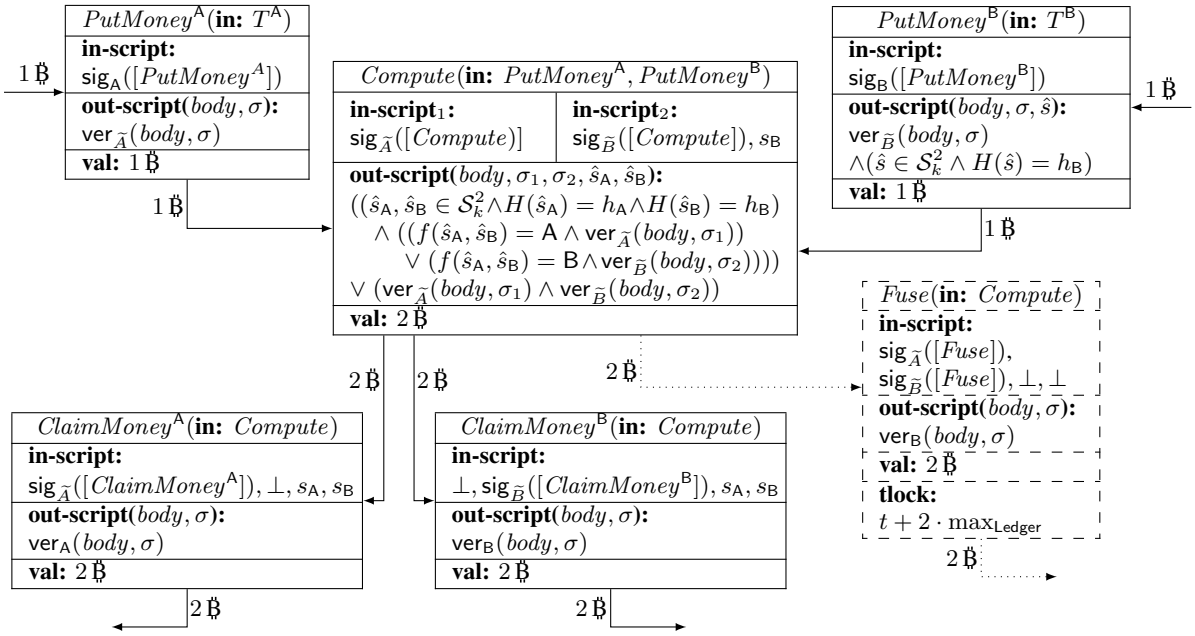
Unfortunately the fact that Alice does not know the complete transaction $Compute$, but only its hash, poses a risk to her. This is because a dishonest Bob can, instead of sending $H(Compute)$, send a hash of some other transaction T in order to obtain the information that can be used to redeem some 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 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 $h_{Compute}$.[18]. 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 $1\ddot{\mathrm{B}}$, which is less than the value of $Fuse$ (equal to $2\ddot{\mathrm{B}}$).

In this way we constructed the $\mathsf{TwoPlayersLottery}$ protocol. Its complete description is presented on Figure 3 (page 22). We now have the following lemma (the proof appears in the full version of this paper).

Lemma 3: The $\mathsf{TwoPlayersLottery}$ 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 Ledger another transaction T, which redeems $PutMoney^{\mathsf{A}}$. If she was lucky and the transaction T was posted on the Ledger before $Compute$, that $Compute$ 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.

[18] 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 $(\bot, \bot, \mathsf{sig}_{\widetilde{A}}([T]), \mathsf{sig}_{\widetilde{B}}([T]))$ (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 $t + 2 \cdot \max_{\mathsf{Ledger}}$



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) Each $P \in \{\mathsf{A}, \mathsf{B}\}$ computes $PutMoney^P$ and post it on Ledger. The players proceed to the next step only once both of these transactions appear on Ledger.

Computation phase:

8) 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.
9) 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:

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

Fig. 3. The TwoPlayersLottery protocol

If the adversary intercepts $Compute$ before it happened (or: if the miner is malicious) then he can post a "mauled" $Compute$ transaction on the Ledger that behaves exactly as the original one, except that it hashes to some other value. Hence $Fuse$ 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 looses 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 their 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 18. 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. Below we present links to some of these transactions on the blockchain.info website. To safe space, all links are relative to the url o 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 Ƀ each and the transaction fees were set to 0.0001 Ƀ 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},
    "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 transactioned 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] I. Abraham, D. Dolev, R. Gonen, and J. Y. Halpern. Distributed computing meets game theory: robust mechanisms for rational secret sharing and multiparty computation. In *PODC*, 2006.

[2] S. Barber, X. Boyen, E. Shi, and E. Uzun. Bitter to better - how to make Bitcoin a better currency. In *FC*, 2012.

[3] D. Basin, C. Caleiro, J. Ramos, and L. Vigano. Distributed temporal logic for the analysis of security protocol models. *TCS*, 2011.

[4] A. Beimel, Y. Lindell, E. Omri, and I. Orlov. $1/p$-Secure multiparty computation without honest majority and the best of both worlds. In *CRYPTO*, 2011.

[5] A. Ben-David, N. Nisan, and B. Pinkas. Fairplaymp: a system for secure multi-party computation. In *CCS*, 2008.

[6] M. Blum. Coin flipping by telephone. In *CRYPTO*, 1981.

[7] D. Boneh and M. Naor. Timed commitments. In *CRYPTO*, 2000.

[8] G. Brassard, D. Chaum, and C. Crépeau. Minimum disclosure proofs of knowledge. *J. Comput. Syst. Sci.*, 1988.

[9] V. Buterin. The bitcoin gambling diaspora, 2013. Bitcoin Magazine.

[10] V. Buterin. Satoshidice sold for $12.4 million, 2013. Bitcoin Magazine.

[11] C. Cachin and J. Camenisch. Optimistic fair secure computation. In *CRYPTO*, 2000.

[12] J. Camenisch, S. Hohenberger, and A. Lysyanskaya. Compact e-cash. In *EUROCRYPT*, 2005.

[13] D. Chaum. Blind signature system. In *CRYPTO*, 1983.

[14] D. Chaum, A. Fiat, and M. Naor. Untraceable electronic cash. In *CRYPTO*, 1988.

[15] J. Clark and A. Essex. CommitCoin: Carbon dating commitments with Bitcoin - (short paper). In *FC*, 2012.

[16] R. Cleve. Limits on the security of coin flips when half the processors are faulty. STOC, 1986.

[17] I. Damgård et al. Practical covertly secure mpc for dishonest majority - or: Breaking the spdz limits. In *ESORICS*. 2013.

[18] D. Dolev and A. C. Yao. On the security of public key protocols. *Information Theory, IEEE Transactions on*, 1983.

[19] C. Dwork and M. Naor. Pricing via processing or combatting junk mail. In *CRYPTO*, 1992.

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

[21] J. A. Garay et al. Rational protocol design: Cryptography against incentive-driven adversaries. In *FOCS*. 2013.

[22] P. Bogetoft et al. Secure multiparty computation goes live. In *FC*, 2009.

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

[24] J. Garay and M. Jakobsson. Timed release of standard digital signatures. In *FC*, 2003.

[25] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. STOC, 1987.

[26] S. Dov Gordon and J. Katz. Partial fairness in secure two-party computation. In *EUROCRYPT*, 2010.

[27] J. Y. Halpern and V. Teague. Rational secret sharing and multiparty computation: Extended abstract. In *STOC*, 2004.

[28] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay — a secure two-party computation system. In *USENIX Security Symposium*, 2004.

[29] I. Miers, C. Garman, M. Green, and A. D. Rubin. Zerocoin: Anonymous distributed e-cash from bitcoin. *IEEE S&P*, 2012.

[30] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[31] B. Pinkas. Fair secure two-party computation. In *Advances in Cryptology – EUROCRYPT 2003*, 2003.

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

[33] P. Resnick, K. Kuwabara, R. Zeckhauser, and E. Friedman. Reputation systems. *Commun. ACM*, 43(12):45–48, December 2000.

[34] D. Ron and A. Shamir. Quantitative analysis of the full bitcoin transaction graph. In *FC*. 2013.

[35] A. Shamir, R. Rivest, and L. Adleman. Mental poker, April 1979. Technical Report LCS/TR-125, Massachusetts Institute of Technology.

[36] L. von Ahn, N. J. Hopper, and J. Langford. Covert two-party computation. In *STOC*, 2005.

[37] A. C.-C. Yao. How to generate and exchange secrets. In *FOCS*, 1986.

Fair Two-Party Computations via the Bitcoin Deposits*

Marcin Andrychowicz**, Stefan Dziembowski***, Daniel Malinowski† and Łukasz Mazurek‡

University of Warsaw

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

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

Our protocols build upon the ideas of our recent paper "Secure Multiparty Computations on Bitcoin" (Cryptology ePrint Archive, Report 2013/784). Compared to that paper, our results are more general, since our protocols allow to compute any function, while in the previous paper we concentrated only on some specific tasks (commitment schemes and lotteries). On the other hand, as opposed to "Secure Multiparty Computations on Bitcoin", to obtain security we need to modify the Bitcoin specification so that the transactions are "non-malleable" (we discuss this concept in more detail in the paper).

1 Introduction

In our recent paper [2] we put forward a new concept dubbed "secure multiparty computations (MPCs) on Bitcoin". On a high level the idea of this concept is as follows. Recall that the MPCs [28,19] are protocols that allow a group of mutually distrusting parties to "emulate" a trusted third party functionality in a secure way. Examples of such functionalities include lotteries, auctions, voting schemes and many more. It is known since 1980s that for any efficiently-computable functionality there exists an efficient protocol that emulates it, assuming that the majority of the participants is honest and that certain computational problems are intractable. If there is no honest majority (in particular: if there are just two parties and one of them is cheating), then such protocols also exist, but in general they do not provide *fairness*, i.e. a dishonest party can prevent the other parties from learning their outputs, after she learned it herself [9,16].

* 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.

** marcin.andrychowicz@crypto.edu.pl

*** stefan.dziembowski@crypto.edu.pl, on leave from the University of Rome *La Sapienza*

† daniel.malinowski@crypto.edu.pl

‡ lukasz.mazurek@crypto.edu.pl

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

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

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

In practice it will make sense to use this protocol if the potential gain from a premature termination is lower than the deposit that the party pays. As the potential applications of our protocols let us mention the *contract signing* problem, which has been extensively studied in cryptography since 1980s [17,5,11,13]. Informally, the challenge in this line of work is to design the protocols where two parties simultaneously sign a document M in a fair way, i.e. it should be impossible for one party, say Alice, to obtain Bob's signature on M without Bob obtaining Alice's signature on M (and vice-versa). It was shown by Even and Yacobi [17] that this task is in general impossible to achieve, and since then there has been a substantial effort to overcome this impossibility result

in various ways (e.g. by assuming an existence of a trusted third party). Since obviously a signing procedure can be modeled as a two-party functionality, hence one can use our protocol to achieve fairness. If the value of the contract is lower than the deposit paid by each party, then clearly the parties will have no incentive to cheat. Moreover, if one party, say Alice, cheats then Bob will earn Alice's deposit (plus he will get his own deposit back), which will compensate his loses resulting from the fact that Alice cheated during the contract signing protocol. Of course, our protocols can be used in several other applications that rely on a fair exchange of secrets, such as certified e-mail systems [30,3,1] or non-repudiation protocols [29].

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

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

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

Related work. As described above our paper builds upon the ideas from our previous paper [2], and hence most of the work relevant to that paper is also relevant to this one. Usage of Bitcoin to create a secure and fair two-player lottery has been independently proposed by Back and Bentov in [4]. Similarly to [2], their protocol makes use of the

[1] A real-life example of such situation is the recent case when the German tax authorities paid 4 million euro to an anonymous informant for a CD containing information about the German tax evaders with bank accounts in Switzerland [12].

time-locked transactions, but the purpose they are used for is slightly different. Their protocol uses time-locks to get the deposit back if the protocol is interrupted, while this paper and [2] use time-locks to make a financial compensation to an honest party, whenever the other party misbehaves.

Usage of timed-commitments to achieve fairness in MPC has been already proposed in a number of papers, e.g. [7,18,24], but this line of research uses a completely different approach from ours. It is based on a gradual release of information and if the protocol is interrupted prematurely than both parties can reconstruct the result with a huge computational effort. The fairness of two-party computation has been also studied by Gordon et al. [16], who showed that complete fairness can be achieved for some functions being computed, e.g. Boolean and/or, but not xor. In contrast, our construction works for an arbitrary function.

Improvements to Bitcoin have been suggested in an important work of Barber et al. [15] who study various security aspects of Bitcoin and Miers et al. [14] who propose a Bitcoin system with provable anonymity. The idea to use some concepts from the MPC literature appeared already in Section 7.1 of [15] where the authors construct a secure "mixer", that allows two parties to securely "mix" their coins in order to obtain unlinkability of the transactions. They also construct commitment schemes with time-locks, however some important details are different, in particular, in the normal execution of the scheme the money is at the end transferred to the receiver. Also, the main motivation of this work is different: the goal of [15] is to fix an existing problem in Bitcoin ("linkability"), while our goal is to use Bitcoin to perform tasks that are hard (or impossible) to perform by other methods.

Commitment schemes and zero-knowledge proofs in the context of the Bitcoin were already considered in [8], however, the construction and its applications are different — the main idea of [8] 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 [25], Example 1), but the application described there is different: the "deposit" is a method for a party with no reputation to prove that she is not a spambot by temporarily sacrificing some of her money.

The Bitcoin wiki "Contracts page" [25] contains several interesting multiparty protocols, and in some sense our work can be viewed as an effort to extend the set of possible types of contracts. We note that the main features that distinguishes our work from most of them is (1) we do not want to rely on any trusted third parties (like the "mediators") and (2) the focus of our protocols is to protect the input privacy.

The problem of the malleability of the transactions has been noticed before and described in [27]. Malleability is a problem for most of the protocols using time-locks (e.g. [4,26]) and Examples 1, 5, and 7 in [25], but is usually not even mentioned, probably because it is believed that it will be eliminated in the future versions of the Bitcoin protocol. In contrast, our lottery protocol from [2] is not susceptible to the malleability problem.

2 A description of Bitcoin

We assume reader's familiarity with the basic principles of the Bitcoin. Let us only briefly recall that the Bitcoin currency system consists of *addresses* and *transactions* between them. An address is simply a public key pk (technically an address is a *hash* of pk). We will frequently denote key pairs using the capital letters (e.g. A). We will also use the following convention: if $A = (sk, pk)$ then $\mathsf{sig}_A(m)$ denotes a signature on a message m computed with sk and $\mathsf{ver}_A(m, \sigma)$ denotes the result (true or false) of the verification of a signature σ on message m with respect to the public key pk.

Each Bitcoin transaction can have multiple inputs and outputs. Inputs of a transaction T_x are listed as triples $(y_1, a_1, \sigma_1), \ldots, (y_n, a_n, \sigma_n)$, where each y_i is a hash of some previous transaction T_{y_i} (our proposal, described in Section 3, is to change it, but for a moment let us stick to the current version of the system), a_i is an index of the output of T_{y_i} (we say that T_x *redeems the* a_i*-th output of* T_{y_i}) and σ_i is called an *input-script* . The outputs of a transaction are presented as a list of pairs $(v_1, \pi_1), \ldots, (v_m, \pi_m)$, where each v_i specifies some amount of coins (called the *value of the* i*-th output of* T_x) and π_i is an *output-script* . A transaction can also have a time-lock t, meaning that it is valid only if time t is reached. Hence, altogether transaction's most general form is: $T_x = ((y_1, a_1, \sigma_1), \ldots, (y_n, a_n, \sigma_n), (v_1, \pi_1), \ldots, (v_m, \pi_m), t)$. The *body of* T_x[2] is equal to T_x without the input-scripts, i.e.: $((y_1, a_1), \ldots, (y_n, a_n), (v_1, \pi_1), \ldots, (v_m, \pi_m), t)$, and denoted by $[T_x]$. One of the most useful properties of Bitcoin is that the users have flexibility in defining the condition on how the transaction T_x can be redeemed. This is achieved by the input- and the output-scripts. One can think of an output-script as a description of a function whose output is Boolean. A transaction T_x defined above is valid if for *every* $i = 1, \ldots, n$ we have that $\pi'_i([T_x], \sigma_i)$[3] evaluates to true, where π'_i is the output-script corresponding to the a_i-th output of T_{y_i}. Another conditions that need to be satisfied are that the time t has already passed and $v_1 + \cdots + v_m \leq v'_1 + \cdots + v'_n$ where each v'_i is the value of the a_i-th output of T_{y_i}. The scripts are written in the Bitcoin scripting language.

Following [2] we will present the transactions as boxes. The redeeming of transactions will be indicated with arrows (cf. e.g. Fig. 1). The transactions where the input script is a signature, and the output script is a verification algorithm are the most common type of transactions and are called *standard transactions*. The address against which the verification is done will be called a *receiver* of this transaction. Currently some miners accept only such transactions. However, there exist other ones that do accept the non-standard (also called *strange*) transactions, one example being a big mining pool called *Eligius*.

We use the security model defined in [2]. For the lack of space we only sketch it here. We assume that the parties are connected by an insecure channel and have access to the Bitcoin chain, which is the only "trusted component" in the system. We assume that each party can access the current contents of the block chain, and post messages on it. Let $\mathsf{max}_{\mathsf{BB}}$ be the is maximal possible delay between broadcasting the transaction and

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

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

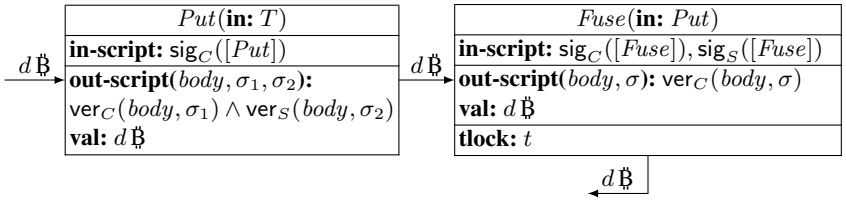


Fig. 1. The graph of transactions for a situation when a user locks d Ƀ. This is an exemplary situation when the problem of malleability arises. C and S denote the pairs of keys hold respectively by the client and the server. t is a moment of time, when the user can take his deposit back. T denotes an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

including it in the block chain. We do not assume that this communication is private. For simplicity we also assume that the transaction fees are zero, but our model and security statements can be easily modified to take into account the non-zero fees.

3 Bitcoin Improvement Proposal

One of the problems with constructing multi-party protocols using Bitcoin is the "malleability" of transactions. This problem has been noticed before by the Bitcoin community[4] as it concerns several Bitcoin protocols that use the advanced features of the scripting language. Essentially, the problem is that, given a valid transaction T, it is possible for everyone to construct a different valid transaction T', which is functionally equivalent to T, but has a different hash. The malleability of transactions comes from the fact, that a hash of a transaction is computed over the whole transaction including its input scripts. On the other hand, signatures are computed only over the body of the transaction, which means that they do not cover the input scripts[5]. Therefore, one can tweak an input script in a way that does not change its functionality (e.g. by adding *push* and *pop* operations[6]) and create a transaction, which is also correct (the signatures are still valid as the input scripts are not signed), and functionally equivalent to the original transaction, yet its hash is different.

To understand why malleability of transactions may be a problem consider a situation, when a client wants to prove to a server that he is not a spambot by locking (making unspendable for a particular amount of time) some amount of bitcoins[7]. To achieve this, the client should create a transaction such that he can not redeem it on his own. But he has to be sure, that he will eventually get his money back after some time. This could be resolved by using a transaction with a time-lock (see Fig. 1 for a graph

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

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

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

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

of transactions) — the client first creates a transaction Put spending his money, which can be redeemed only by a transaction signed by him and the server (so they can agree to return the deposit to the client at any time). Then he sends the hash of this transaction to the server and the server returns a transaction $Fuse$ with a signature of the server on it [8] — this transaction sends back the deposit to the client after some time. So now the client may broadcast the first transaction, and after some time he may use the $Fuse$ transaction to get back his deposit. This is exactly where the problem of malleability arises: if an adversary sees the transaction Put after it is broadcast, but before it is included in the block chain (as the transactions are broadcast in a peer-to-peer network), he can create and broadcast a transaction Put', which is functionally equivalent to Put, but has a different hash. Then, if Put' is included in the block chain first, the original Put becomes invalidated. As a result the $Fuse$ will not be correct (it contains a hash of Put, which never appeared in the block chain), so the client may lose his money.

A source of the malleability problem is that a hash of a transaction depends on its input scripts. In some situation this dependence is itself a problem, because we may not know the input scripts of the transaction T while signing a transaction redeeming T. In next section we present a possible solution for these problems. It requires a small modification of the Bitcoin specification. We believe that this modification could be implemented in the future in Bitcoin. We discuss why it does not decrease the security of Bitcoin.

Our modification. In the current version of Bitcoin protocol, each transaction contains a hash of the transaction it spends. That hash is computed over the *whole* transaction. We propose to compute those hashes over the transaction without its input scripts (i.e. over the *body* of the transaction), so they would be computed in the same way the hashes for transactions' signatures are currently being computed. That means that the transaction would have the same hash value regardless of its input scripts.

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

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

Now consider, what in fact is changed with this modification. The input scripts are used only to show that the transaction is authorized to redeem the other transactions. So two correct transactions which differ only in the input scripts are equivalent — they prove in two different ways that the Bitcoin transfer is authorized. It is not possible

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

that the block chain contains two such transactions. That is why the hash still uniquely identifies the redeemed transaction.[9]

4 Simultaneous Bitcoin-based timed commitment scheme

In this section we present a modification of the Bitcoin-based timed commitment scheme introduced in [2]. To make the paper self-contained we first recall the original timed-commitment scheme $\mathsf{CS}(\mathsf{C}, d, t, s)$ of [2], and then we describe our modified scheme.

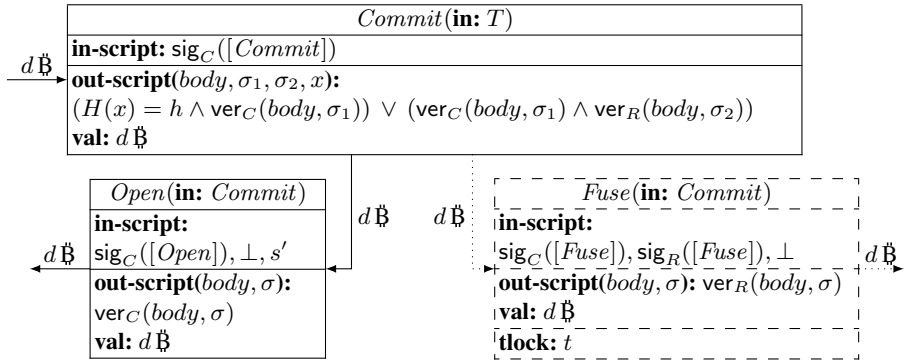
Timed-commitments of [2]. Recall that a (standard) commitment scheme is a protocol between two parties: a committer C and a receiver R. The protocol contains of two phases. In the first one, called the *commitment phase*, C *commits* to some secret string s by interacting with R. What is important is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

Several other commitment schemes have been constructed over the last 2 decades. One inherent problem with all of them is related to the fairness issue in the two-party computation protocols (see Sect. 1). Namely, there is no way to force C to open the commitment. This problem has negative consequences for several applications. Consider, e.g., a simple protocol in which two parties (call them again C and R) want to "flip a coin", i.e., to select a bit $b \leftarrow \{0,1\}$ uniformly at random. A simple protocol of Blum [6] for this problem works as follows: (1) C commits to some random bit $c \leftarrow \{0,1\}$, (2) R selects a random bit $r \leftarrow \{0,1\}$ and sends it to C, (3) C opens his commitment, and the output of the protocol is computed as $b = c \oplus r$. This protocol is obviously secure, informally because the hiding property of the commitment scheme guarantees that R does not know c when he chooses r, and the binding property prevents C from changing c after he learned r. Unfortunately, there is no way to force R to complete Step (3) and to open the commitment. Hence, C he can make the protocol "crash" without producing the output, depending on what the output is.

As a remedy to this problem [2] propose to use Bitcoin in the following way. During the commitment phase the committer has to put aside a "deposit". Assume its value is

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

d Ƀ, and it comes from an unredeemed standard transaction T, whose receiver is C. The committer gets his money back once he opens the commitment. If he does not open the commitment within some time t then the money can be claimed by the receiver. This is implemented using the Bitcoin scripts and time-locks on top of the hash-based commitment scheme described above. Let C and R be the respective key-pairs of C and R. The transactions used in this implementation are as follows (the scripts' arguments, which are omitted are denoted by $\perp$):



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

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

It is easy to see how this timed-commitment scheme solves the problem of fairness in the coin-flipping protocol described above: if C did not open the commitment scheme on time then he is "punished" financially for this, and R gets a compensation. Unfortunately, this commitment scheme does not solve the fairness problem in general. This is because for the general two-party computation protocols we need something stronger. More precisely, the problem is that this commitment scheme forces the committer to reveal his secret (or to pay a fine), no matter how the other party behaves. To see why it is a problem, imagine two parties, called Alice and Bob holding secrets denoted respectively s_A and s_B. Suppose that the protocol instructs both of them to commit to their secrets and then to reveal them (in fact this is exactly the situation that we have in our two-party scheme in Sect. 5). If they just run two instantiations of the CS scheme, then

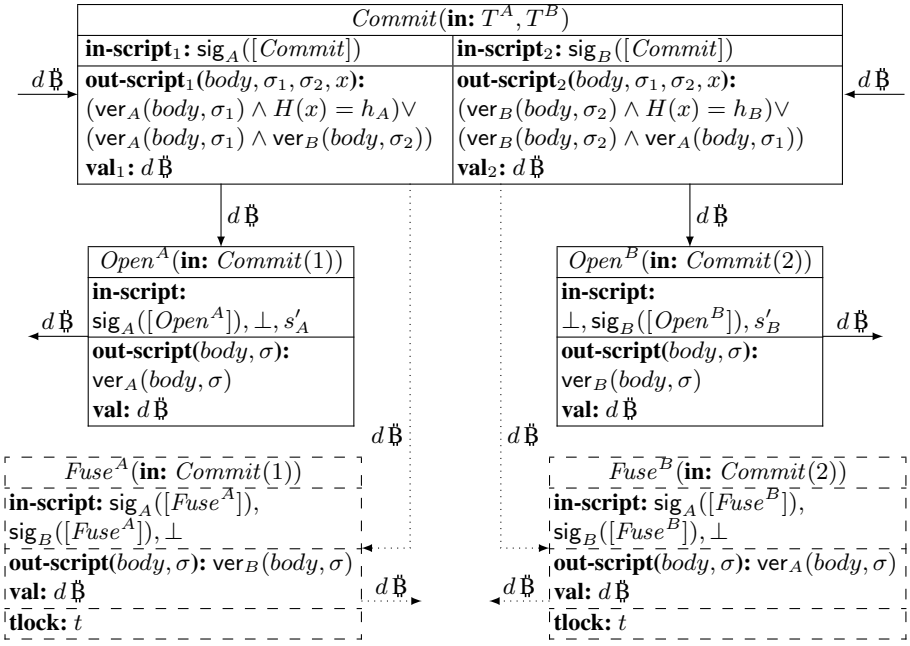
one party, say Alice, can interrupt the protocol where she is the committer, after Bob has already made a commitment. In that case Bob will be forced to reveal his secret share or lose his deposit. Hence, it is important that both commitment schemes are executed *simultaneously*, i.e. it is not possible that as a result of the protocol one of the parties is committed to her secret and the other one is not. A construction of such a commitment scheme is one of our two main contributions and is presented below.

Simultaneous Bitcoin-based timed commitment scheme. The protocol is denoted by $\mathsf{SCS}(\mathsf{A}, \mathsf{B}, d, t)$, where A and B are the parties executing the protocol, d is the value of the deposits in Ƀ, t is the timestamp — the parties should open the commitments before that time, and s_A, s_B are the secrets. We assume that A and B are the respective key pairs of A and B and the block chain contains unredeemed transactions T^A and T^B, both of a value d, whose receivers are A and B respectively. The protocol is depicted on Fig. 2. The commitment phase is denoted by $\mathsf{SCS.Commit}(\mathsf{A}, \mathsf{B}, d, t)$ and the opening phase is denoted by $\mathsf{SCS.Open}(\mathsf{A}, \mathsf{B}, d, t)$. Let α be the security parameter.

The security definition of the SCS protocol is very similar to the security definition of the CS protocol of [2] described above. We model the hash function H used in the protocol as a random oracle. We require that the commitment is hiding and binding. We allow a negligible (in α) error probabilities in both hiding and biding. The protocol can be interrupted during the commitment phase — in this case the parties do not lose any bitcoins and do not learn the other party's secret. The only difference between the CS protocol and the SCS protocol is that if the SCS protocol is not interrupted during the commitment phase, then *both* parties are committed. This means that an honest party can be sure that her opponent either reveals the secret by the time t or transfers d Ƀ to her. Moreover, it is guaranteed that the party which reveals a secret would get her deposit back. Again, we allow negligible probabilities that the above statements do not hold.

We construct the SCS protocol assuming the Bitcoin modification from Section 3. The detailed description of the SCS protocol is presented on Fig. 2. In SCS protocol we assume that both parties already know the hashes h_A and h_B of *both* secrets concatenated with some random strings ρ_A and ρ_B (resp.). More precisely: $h_A := H(s_A||\rho_A)$ and $h_B := H(s_B||\rho_B)$, where $\rho_A \leftarrow \{0,1\}^\alpha$ and $\rho_B \leftarrow \{0,1\}^\alpha$. The reason for this will become clear in Sec. 5. The idea behind the protocol is as follows. First the parties use the existing transactions T^A and T^B to construct the transaction $Commit$. The transaction $Commit$ has two outputs — one is used to commit A to s_A and the other one to commit B to s_B. The first output can be claimed by A with revealing her secret or after time t by B. The latter option is technically achieved by signing at the very beginning of the protocol a transaction $Fuse^A$, which redeems $Commit$, can be claimed only by B and has a time-lock t. The second output of $Commit$ is analogous. The proof of the following lemma appears in the extended version of this paper.

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



Pre-condition:

1. A holds the key pair A and B holds the key pair B.
2. A knows the secret s_A, B knows the secret s_B, both players know the hashes $h_A = H(s'_A)$ and $h_B = H(s'_B)$, where $s'_A := (s_A||\rho_A)$, $s'_B := (s_B||\rho_B)$, and $\rho_A, \rho_B \leftarrow \{0,1\}^\alpha$ are random strings known only to A and B respectively.
3. There are two unredeemed transactions T^A, T^B of value $d\,\ddot{\mathrm{B}}$, which can be redeemed with the keys A and B respectively.

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

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

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

7. A and B broadcast the transactions $Open^A$ and $Open^B$ respectively, what reveals the secrets s_A and s_B.
8. If within time t the transaction $Open^A$ does not appear on the block chain, then B broadcasts the transaction $Fuse^A$ and gets $d\,\ddot{\mathrm{B}}$. Similarly, if within time t the transaction $Open^B$ does not appear on the block chain, then A broadcasts the transaction $Fuse^B$ and gets $d\,\ddot{\mathrm{B}}$.

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

5 Two-party computation

The concept of secure two-party computations has already been informally described in the introduction. For the lack of space we do not provide full security definitions of these protocols, and only briefly sketch the constructions. The reader may refer to [10,20] for more on this topic. A common paradigm [21] for constructing secure multiparty protocols is to: (1) create a protocol secure only against passive (also called "semi-honest") adversaries, i.e. adversaries, which honestly perform the protocol, and then (2) "compile" such a protocol to be secure against any type of adversarial behavior.

The problem that such a compiler needs to address is that a malicious party can send a different message than she is supposed to send according to the protocol. One can deal with this problem using the zero-knowledge protocols [23]. This is possible since in every protocol a message which should be sent by a party is determined by (a) the public inputs, (b) the party's private inputs, (c) the messages that she received earlier, and (d) the party's internal randomness. The idea is to attach to each message a zero-knowledge proof that this message was computed correctly. Since a message can depend on private inputs and the internal randomness of the sender (which are not known to the receiver), the players commit at the beginning of the protocol to their private inputs and the randomness and later use these commitments in the proof (they actually never open them). Moreover, we need to ensure that the bits used as internal randomness are indeed random, but it can be easily achieved by masking them with the bits chosen by the other party. More details can be found, e.g., in [20].

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

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

Our construction is based on the two-party computation protocol by Goldreich and Vainish [22]. We do not provide the details of this protocol here (for its full description the reader may consult, e.g. [10]). Let us just describe its most relevant part. The property which we take advantage of is that at the end of the protocol's execution the parties hold additive shares of the result of the computation, but none of the parties learned anything about the actual output. This means that the parties holds respectively bit strings s_A and s_B, such that the result of the computation is equal to $s_A \oplus s_B$. In the original protocol, the parties reconstruct the result by revealing their shares. More

precisely, each party sends its share to the other party and makes a zero-knowledge proof that it is indeed its share of the result. Of course, one of the parties has to reveal her share first (or at least a part of it) and the other party can quit the protocol at this moment, leaving the honest party with no information about the output[10].

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

The idea behind FairComp protocol is as follows. Suppose that the parties are called Alice and Bob. At the very beginning Alice and Bob agree on a value of a deposit equal to d ฿. Then they execute the two-party protocol [22,10] together with the zero-knowledge proofs in order to make it secure against the active adversary. However, they *do not* reconstruct the result. Then, Alice sends a hash h_A of her share concatenated with some random string to Bob and makes a zero-knowledge proof that she indeed computed h_A in that way. Similarly, Bob sends h_B to Alice and makes an analogous proof. Later, the parties execute SCS protocol to simultaneously commit themselves to respectively h_A and h_B. When the commitment is done, the parties reveal their shares. If any of them does not reveal its share, the honest party can claim the opponent's deposit. The description of the protocol is presented on Fig. 3. We now have the following lemma whose proof appears in the extended version of this paper.

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

6 Extensions

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

Probably the most interesting extension is to make the payoffs in the FairComp protocol depend on the result of the computation. More precisely, the FairComp protocol can be easily extended to handle a situation when the result of the computation determines the winner, which will be given some reward (an agreed amount of bitcoins). To achieve this it is enough to add a third output with the value equal to the value of the reward to the $Commit$ transaction used in the execution of SCS.Commit in Step. 4 of FairComp protocol. The output script would take as arguments both secrets s'_A, s'_B and a signature. It would check if both provided secrets are correct ($H(s'_A) = h_A \wedge H(s'_B) = h_B$), compute s_A and s_B as prefixes of respectively s'_A and

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

Pre-condition:

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

The computation phase

The parties execute the two-party protocol of Goldreich and Vainish [22] additionally secured against an active adversary with zero-knowledge proofs, but they do not reconstruct the secret. At the end of the execution A holds s_A and B holds s_B, such that the result of the computation in equal to $s_A \oplus s_B$.

The commitment phase

1. A computes her secret s'_A as a concatenation of her share s_A and some random string ρ_A of length α, where α is a security parameter.
2. A sends $h_A := H(s'_A)$ to B and makes a zero-knowledge proof to B that this value is indeed equal to $H(s_A||\rho_A)$ for some string ρ_A.
3. Similarly, B computes $s'_B := s_B||\rho_B$ for some random string ρ_B of length α, sends $h_B := H(s'_B)$ to A and makes an analogous proof.
4. The parties execute $\mathsf{SCS.Commit}(\mathsf{A}, \mathsf{B}, d, t)$ protocol for some moment of time t in the future.

The opening phase

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

Fig. 3. The FairComp protocol.

s'_B, compute the actual result $(s := s_A \oplus s_B)$, check which party is a winner and verify if the signature is the winner's signature on that transaction (this idea is very similar to the ones used in [4] and [2]).

The idea described above can be further extended to handle a situation, where the reward may be split arbitrarily among the parties depending on the result of the computation, e.g. the result is a fraction between 0 and 1, which determines how big part of the reward will be given to one of the parties (the other party gets the rest of the reward). Suppose that the reward is equal to 1 ₿. The parties have to add to $Commit$ transaction, not one additional output, but a number of them — one with value 0.5 ₿, one with value 0.25 ₿, one with value 0.125 ₿ and so on[11]. Similarly as earlier, each output script expects both secrets and a signature. It computes the results of the computation, checks, which party should be given the appropriate part of the reward and verifies if the signature is that party's signature.

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

7 Acknowledgments

We would like to thank the anonymous BITCOIN workshop reviewers for their comments.

References

1. M. Abadi and N. Glew. Certified email with a light on-line trusted third party: Design and implementation. WWW '02.
2. M. Andrychowicz, S. Dziembowski, D. Malinowski, and Ł. Mazurek. Secure Multiparty Computations on Bitcoin. Cryptology ePrint Archive, 2013. `http://eprint.iacr.org/2013/784`.
3. Giuseppe Ateniese and Cristina Nita-Rotaru. Stateless-recipient certified e-mail system based on verifiable encryption. In *Topics in Cryptology – CT-RSA 2002*.
4. Adam Back and Iddo Bentov. Note on fair coin toss via bitcoin, 2013. `http://www.cs.technion.ac.il/~idddo/cointossBitcoin.pdf`.
5. Michael Ben-Or, Oded Goldreich, Silvio Micali, and Ronald L. Rivest. A fair protocol for signing contracts. *IEEE Transactions on Information Theory*, 36(1):40–46, 1990.
6. M. Blum. Coin flipping by telephone. In *CRYPTO*, 1981.
7. D. Boneh and M. Naor. Timed commitments. In *CRYPTO '00*.
8. J. Clark and A. Essex. CommitCoin: Carbon dating commitments with Bitcoin. In *FC*, 2012.
9. R. Cleve. Limits on the security of coin flips when half the processors are faulty. STOC '86.
10. Ronald Cramer. Introduction to secure computation. In *Lectures on Data Security*, '98.
11. Ivan Damgård. Practical and provably secure release of a secret and exchange of signatures. In *EUROCRYPT '93*.
12. Der Spiegel International. Swiss Bank Data: German Tax Officials Launch Nationwide Raids, April 2013.
13. Birgit Pfitzmann et al. Optimal efficiency of optimistic contract signing. In *PODC '98*.
14. I. Miers et al. Zerocoin: Anonymous distributed e-cash from bitcoin. *IEEE S&P*, 2012.
15. S. Barber et al. Bitter to better - how to make Bitcoin a better currency. In *FC*, 2012.
16. S. Gordon et al. Complete fairness in secure two-party computation. *J. ACM*, 58(6), 2011.
17. S. Even and Y. Yacobi. Relations among public key signature schemes., 1980. Technical Report 175, Computer Science Dept., Technion, Israel.
18. J. A. Garay and M. Jakobsson. Timed release of standard digital signatures. In *FC '02*.
19. O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. STOC, 1987.
20. Oded Goldreich. *The Foundations of Cryptography - Volume 2, Basic Applications*. 2004.
21. Oded Goldreich, Silvio Micali, and Avi Wigderson. Proofs that yield nothing but their validity and a methodology of cryptographic protocol design. In *FOCS '86*.
22. Oded Goldreich and Ronen Vainish. How to solve any protocol problem - an efficiency improvement. In *CRYPTO '87*.
23. S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM J. Comput.*, 18(1):186–208, February 1989.
24. Benny Pinkas. Fair secure two-party computation. In *EUROCRYPT*, pages 87–105, 2003.
25. Bitcoin wiki. Contracts. en.bitcoin.it/wiki/Contracts, Accessed on 24.11.2013.
26. Bitcoin wiki. Dominant Assurance Contracts. en.bitcoin.it/wiki/Dominant_Assurance_Contracts, Accessed on 19.01.2014.
27. Bitcoin wiki. Transaction malleability. en.bitcoin.it/wiki/Transaction_Malleability, Accessed on 20.1.2014.
28. A. C.-C. Yao. How to generate and exchange secrets. In *FOCS*, 1986.
29. J. Zhou and D. Gollmann. A fair non-repudiation protocol. In *IEEE S&P*, 1996.
30. Jianying Zhou and Dieter Gollmann. Certified electronic mail. In *ESORICS '96*.

Modeling Bitcoin Contracts by Timed Automata*

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

Cryptology and Data Security Group
www.crypto.edu.pl
University of Warsaw

Abstract. Bitcoin is a peer-to-peer cryptographic currency system. Since its introduction in 2008, Bitcoin has gained noticeable popularity, mostly due to its following properties: (1) the transaction fees are very low, and (2) it is not controlled by any central authority, which in particular means that nobody can "print" the money to generate inflation. Moreover, the transaction syntax allows to create the so-called *contracts*, where a number of mutually-distrusting parties engage in a protocol to jointly perform some financial task, and the fairness of this process is guaranteed by the properties of Bitcoin. Although the Bitcoin contracts have several potential applications in the digital economy, so far they have not been widely used in real life. This is partly due to the fact that they are cumbersome to create and analyze, and hence risky to use.

In this paper we propose to remedy this problem by using the methods originally developed for the computer-aided analysis for hardware and software systems, in particular those based on the timed automata. More concretely, we propose a framework for modeling the Bitcoin contracts using the timed automata in the UPPAAL model checker. Our method is general and can be used to model several contracts. As a proof-of-concept we use this framework to model some of the Bitcoin contracts from our recent previous work. We then automatically verify their security in UPPAAL, finding (and correcting) some subtle errors that were difficult to spot by the manual analysis. We hope that our work can draw the attention of the researchers working on formal modeling to the problem of the Bitcoin contract verification, and spark off more research on this topic.

1 Introduction

Bitcoin is a digital currency system introduced in 2008 by an anonymous developer using a pseudonym "Satoshi Nakamoto" [23]. Despite of its mysterious origins, Bitcoin became the first cryptographic currency that got widely adopted — as of January 2014 the Bitcoin capitalization is over €7 bln. The enormous success of Bitcoin was also widely covered by the media (see e.g. [16,5,25,21,22]) and even attracted the attention of several governing bodies and legislatures, including the US Senate [21]. Bitcoin owes its popularity mostly to the fact that it has no central authority, the transaction fees are

* 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.

** On leave from the *Sapienza* University of Rome.

very low, and the amount of coins in the circulation is restricted, which in particular means that nobody can "print" money to generate inflation. The financial transactions between the participants are published on a public ledger maintained jointly by the users of the system.

One of the very interesting, but slightly less known, features of the Bitcoin is the fact that it allows for more complicated "transactions" than the simple money transfers between the participants: very informally, in Bitcoin it is possible to "deposit" some amount of money in such a way that it can be claimed only under certain conditions. These conditions are written in the form of the *Bitcoin scripts* and in particular may involve some timing constrains. This property allows to create the so-called *contracts* [27], where a number of mutually-distrusting parties engage in a Bitcoin-based protocol to jointly perform some task. The security of the protocol is guaranteed purely by the properties of the Bitcoin, and no additional trust assumptions are needed. This Bitcoin feature can have several applications in the digital economy, like creating the assurance contracts, the escrow and dispute mediation, the rapid micropayments [27], the multi-party lotteries [8]. It can also be used to add some extra properties to Bitcoin, like the certification of the users [17], or creating the secure "mixers" whose goal is to enhance the anonymity of the transactions [19]. Their potential has even been noticed by the media (see, e.g., a recent enthusiastic article on the *CNN Money* [22]).

In our opinion, one of the obstacles that may prevent this feature from being widely used by the Bitcoin community is the fact that the contracts are tricky to write and understand. This may actually be the reason why, despite of so many potential applications, they have not been widely used in real life. As experienced by ourselves [6,7,8], developing such contracts is hard for the following reasons. Firstly, it's easy to make subtle mistakes in the scripts. Secondly, the protocols that involve several parties and the timing constraints are naturally hard to analyze by hand. Since mistakes in the contracts can be exploited by the malicious parties for their own financial gain, it is natural that users are currently reluctant to use this feature of Bitcoin.

In this paper we propose an approach that can help designing secure Bitcoin contracts. Our idea is to use the methods originally developed for the computer-aided analysis for hardware and software systems, in particular the timed automata [1,2]. They seem to be the right tool for this purpose due to the fact that the protocols used in the Bitcoin contracts typically have a finite number of states and depend on the notion of time. This time-dependence is actually two-fold, as (1) it takes some time for the Bitcoin transactions to be confirmed (1 hour, say), and (2) the Bitcoin transactions can come with a "time lock" which specifies the time when a transaction becomes valid.

Our contribution We propose a framework for modeling the Bitcoin contracts using timed automata in the UPPAAL model checker [9,18] (this is described in Sec. 2). Our method is general and can be used to model a wide class of contracts. As a proof-of-concept, in Sec. 3 we use this framework to model two Bitcoin contracts from our previous work [8,6]. This is done manually, but our method is quite generic and can potentially be automatized. In particular, most of the code in our implementation does not depend on the protocol being verified, but describes the properties of Bitcoin system. To model a new contract it is enough to specify the transactions used in the contract, the knowledge of the parties at the beginning of the protocol and the protocol followed

by the parties. We then automatically verify the security of our contracts in UPPAAL (in Sec. 3.1). The UPPAAL code for the contracts modeled and verified by us is available at the web page `http://crypto.edu.pl/uppaal-btc.zip`.

Future work We hope that our work can draw the attention of the researchers working on formal modeling to the problem of the Bitcoin contracts verification, and spark off more research on this topic. What seems especially interesting is to try to fully automatize this process. One attractive option is to think of the following workflow: (1) a designer of a Bitcoin contract describes it in UPPAAL (or, possibly, in some extension of it), (2) he verifies the security of this idealized description using UPPAAL, and (3) if the idealized description verifies correctly, then he uses the system to "compile" it into a real Bitcoin implementation that can be deployed in the wild. Another option would be to construct a special tool for designing the Bitcoin contracts, that would produce two outputs: (a) a code in the UPPAAL language (for verification) and (b) a real-life Bitcoin implementation.

Of course, in both cases one would need to formally show the soundness of this process (in particular: that the "compiled" code maintains the properties of the idealized description). Hence, this project would probably require both non-trivial theoretical and engineering work.

Preliminaries Timed automata were introduced by Alur and Dill [1,2]. There exist other model checkers based on this theory, like Kronos [31] and Times [4]. It would be interesting to try to implement our ideas also in them. Other formal models that involve the notion of the real time include the timed Petri nets [10], the timed CSP [26], the timed process algebras [30,24], and the timed propositional temporal logic [3]. One can try to model the Bitcoin contracts also using these formalisms. For the lack of space, a short introduction to UPPAAL was moved to Appendix D. The reader may also consult [9,18] for more information on this system.

We assume reader's familiarity with the public-key cryptography, in particular with the signature schemes (an introduction to this concept can be found e.g. in [20,12]). We will frequently denote the 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. We will also use the following convention: if $A = (A.sk, A.pk)$ then $\mathsf{sig}_A(m)$ denotes a signature on a message m computed with $A.sk$ and $\mathsf{ver}_A(m, \sigma)$ denotes the result (true or false) of the verification of a signature σ on message m with respect to the public key $A.pk$. We will use the "฿" symbol to denote the Bitcoin currency unit.

1.1 A short description of Bitcoin

Since we want the exposition to be self-contained, we start with a short description of Bitcoin, focusing only on the most relevant parts. For the lack of space we do not describe how the coins are created, how the transaction fees are charged, and how the Bitcoin "ledger" is maintained. A more detailed description of Bitcoin is available on the *Bitcoin wiki* site [11]. The reader may also consult the original Nakamoto's paper [23].

Introduction In general one of the main challenges when designing a 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 have 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_p", signed by the user X. In this case each user can verify if money from transaction T_p has not been already spent by X. The main difficulty in designing the fully-distributed peer-to-peer currency systems is to devise a system where the users jointly maintain the ledger in such a way that it cannot be manipulated by an adversary and it is publicly-accessible.

In Bitcoin this problem is solved by a cryptographic tool called *proofs-of-work* [15]. We will not go into the details of how this is done, since it is not relevant to this work. Let us only say that the system works securely as long as no adversary controls more computing power than the combined computing power of all the other participants of the protocol[1]. The Bitcoin participants that contribute their computing power to the system are called the *miners*. Bitcoin contains a system of incentives to become a miner. For the lack of space we do not describe it here.

Technically, the ledger is implemented as a chain of blocks, hence it is also called a "block chain". When a transaction is posted on the block chain, it can take some time before it appears on it, and even some more time before the user can be sure that this transaction will not be cancelled. However, it is safe to assume that there exists an upper bound on this waiting time (1-2 hours, say). We will denote this time by `MAX_LATENCY`.

As already highlighted in the introduction, the format of the Bitcoin transactions is in fact quite complex. Since it is of a special interest for us, we describe it now in more detail. The Bitcoin currency system consists of *addresses* and *transactions* between them. An address is simply a public key pk[2]. Normally every such key has a corresponding private key sk known only to one user, which is an *owner* of this address. 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.

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. 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

[1] It is currently estimated [25] that the combined computing power of the Bitcoin participants is around 64 exaFLOPS, which exceeds by factor over 200 the total computing power of world's top 500 supercomputers, hence the cost of purchasing the equipment that would be needed to break this system is huge.

[2] Technically an address is a *cryptographic 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.

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 the value of T_y and v 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: $T_x = (y_1, \ldots, y_\ell, B.pk, v, \mathsf{sig}_{A_1}(y_1, B.pk, v), \ldots, \mathsf{sig}_{A_\ell}(y_\ell, B.pk, v))$, 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 *time lock* t that tells at what time in the future the transaction becomes valid. The lock-time t can refer either to a measure called the "block index" or to the real physical time. In this paper we only consider the latter type of time-locks. In this case we have $T_x = (y_1, \ldots, y_\ell, B.pk, v, t, \mathsf{sig}_{A_1}(y_1, B.pk, v, t), \ldots, \mathsf{sig}_{A_\ell}(y_\ell, B.pk, v, t))$. 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 and to divide transactions with large value into smaller portions. We ignore this fact in our description since we will not use it in our protocols.

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 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, its 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*[3] of T_x and σ_x is a "witness" that is used to make the script π_y evaluate to

[3] In the original Bitcoin documentation this is called "simplified T_x". Following our earlier work [8,6,7] we chosen to rename it to "body" since we find the original terminology slightly misleading.

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 [28], which is stack-based and similar to the Forth programming language. It is on purpose not Turing-complete (there are no loops in it), since the scripts need to evaluate in (short) finite time. 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: $T_x = (y_1, \ldots, y_\ell, \pi_x, v, t, \sigma_1, \ldots, \sigma_\ell)$, 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). An example of a graphical representation of a transaction is depicted in Fig. 1.

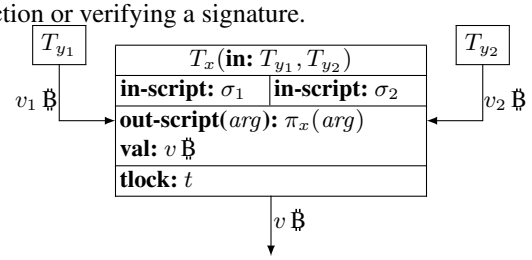


Fig. 1. A graphical representation of a transaction $T_x = (y_1, y_2, \pi_x, v, t, \sigma_1, \sigma_2)$.

The transactions where the input script is a signature, and the output script is a verification algorithm are the most common type of transactions. We will call them *standard transactions*. Currently some miners accept only such transactions, due to the fact that writing more advanced scripts is hard and error-prone, and anyway the vast majority of users does not use such advanced features of Bitcoin. Fortunately, there exist other miners that do accept the non-standard (also called *strange*) transactions, one example being a big mining pool[4] called *Eligius* (that mines a new block on average once per hour). We also believe that in the future accepting the general transaction will become standard, maybe at a cost of a slightly increased fee. Actually, popularizing the Bitcoin contracts, by making them safer to use, is one of the goals of this work.

2 Modeling the Bitcoin

To reason formally about the security of the contracts we need to describe the attack model that corresponds to the Bitcoin system. The model used in [8,6] was inspired by the approach used in the complexity-based cryptography. This way of modeling protocols, although very powerful, is not well-suited for the automatic verification of cryptographic protocols. In this section we present the approach used in this paper, based on timed-automata, using the syntax of the UPPAAL model checker.

In our model each party executing the protocol is modeled as a timed automaton with a structure assigned to it that describes the party's knowledge. States in the automaton describe which part of the protocol the party is performing. The transitions in the automaton contain conditions, which have to be satisfied for a transition to be taken and actions, which are performed whenever a transition is taken.

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

The communication between the parties may be modeled in a number of various ways. It is possible to use synchronization on channels offered by UPPAAL and shared variables representing data being sent between the parties. In all protocols verified by us, the only messages exchanged by the parties were signatures. We decided to model the communication indirectly using shared variables — each party keeps the set of known signatures, and whenever a sender wants to send a signature, he simply adds it to the recipient's set.

The central decision that needs to be made is how to model the knowledge of the honest parties and the adversary. Our representation of knowledge is symbolic and based on Dolev-Yao model [13], and hence we assume that the cryptographic primitives are perfectly secure. In our case it means, for example, that it is not possible to forge a signature without the knowledge of the corresponding private key, and this knowledge can be modeled in a "binary" way: either an adversary knows the key, or not. The hash functions will be modeled as random oracles, which in particular implies that they are collision-resilient and hard to invert. We also assume that there exists a secure and authenticated channel between the parties, which can be easily achieved using the public key cryptography. Moreover, we assume that there is a fixed set of variables denoting the private/public pairs of Bitcoin keys. A Bitcoin protocol can also involve secret strings known only to some parties (like e.g. a string s in the commitment protocol in Appendix B), we assume that there is a fixed set of variables denoting such strings. For each private key and each secret string there is a subset of parties, which know them, but all public keys and hashes of all secret strings are known to all the parties (if this is not the case then they can be broadcast by parties knowing them).

A block chain is modelled using a shared variable (denoted `bc`) keeping the status of all transactions and a special automaton, which is responsible for maintaining the state of `bc` (e.g. confirming transactions).

In the following sections we describe our model in more details.

2.1 The keys, the secret strings, and the signatures

We assume that the number of the key pairs in the protocol is known in advance and constant. Therefore, key pairs will be simply referred by consecutive natural numbers (type **Key** is defined as an integer from a given range). Secret strings are modelled in the same way. As already mentioned we assume that all public keys and hashes of all secrets are known to all parties.

```
typedef struct {
  Key key;
 TxId tx_num;
Nonce input_nonce;
} Signature;
```

Fig. 2. The signatures type.

Moreover, we need to model the signatures over transactions (one reason is that they are exchanged by the parties in some protocols). They are modelled by structures containing a transaction being signed, the key used to compute a signature and an `input_nonce`, which is related to the issue of transaction malleability and described in Appendix A.

2.2 The transactions

We assume that all transactions that can be created by the honest parties during the execution of the protocol comes from a set, which is known in advance and of size T. Additionally, the adversary can create his own transactions. As explained later (see Sec. 2.4 below) we can upper-bound the number of adversarial transactions by T. Hence the total upper bound on the number of transactions is $2T$.

For simplicity we refer the transactions using fixed identifiers instead of their hashes. We can do this because we know all the transactions, which can be broadcast in advance (compare Sec.2.4 and Appendix A for further discussion). A single-input and single-output transaction is a variable of a record type **Tx** defined in Fig. 3. The `num` field is the identifier of the transaction, and the `input` field is the identifier of its input transaction. The `value` field denotes the value of the transaction (in ₿). The `timelock` field indicates the time lock of the transaction, and the `timelock_passed` is a boolean field indicating whether the `timelock` has passed.

```
typedef struct {
        TxId num;
        TxId input;
         int value;
         int timelock;
        bool timelock_passed;
      Status status;
       Nonce nonce;
        bool reveals_secret;
      Secret secret_revealed;
OutputScript out_script;
} Tx;
```

Fig. 3. The transactions type

The `status` field is of a type **Status** that contains following values: `UNSENT` (indicating that transaction has not yet been sent to the block chain), `SENT` (the transaction has been sent to the block chain and is waiting to be confirmed), `CONFIRMED` (the transaction is confirmed on the block chain, but not spent), `SPENT` (the transaction is confirmed and spent), and `CANCELLED` (the transaction was sent to the block chain, but while it was waiting for being included in the block chain its input was redeemed by another transaction).

The `out_script` denotes the output script. In case the transaction is standard it simply contains a public key of the recipient of the transaction. Otherwise, it refers to a hard-coded function which implements the output script of this transaction (see Sec. 2.5 for more details).

The inputs scripts are modelled only indirectly (the fields `reveals_secret` and `secret_revealed`). More precisely, we only keep information about which secrets are included in the input script (see e.g. the CS protocol in Fig. 9, transaction $Open$).

The above structure can be easily extended to handle multiple inputs and outputs.

2.3 The parties

The parties are modelled by timed automata describing protocols they follow. States in the automata describe which part of the protocol the party is performing. The transitions in the automata

```
typedef struct {
            bool know_key[KEYS_NUM];
            bool know_secret[SECRETS_NUM];
    int[0,KNOWN_
SIGNATURES_SIZE] known_signatures_size;
       Signature known_signatures
                 [KNOWN_SIGNATURES_SIZE];
} Party;
```

Fig. 4. The parties type.

contain conditions, which have to
be satisfied for a transition to be
taken and actions, which are performed whenever a transition is taken. An example of such automaton appears in Fig. 7. and is described in more details in Appendix D.2. The adversary is modelled by a special automaton described in Sec. 2.4.

Moreover, we need to model the knowledge of the parties (both the honest users and the adversary) in order to be able to decide whether they can perform a specific action in a particular situation (e.g. compute the input script for a given transaction). Therefore for each party, we define a structure describing its knowledge. More technically: this knowledge is modelled by a record type **`Party`** defined in Fig. 4.

The boolean tables `know_key[KEYS_NUM]` and `know_secret[SECRETS_NUM]` describe the sets of keys and secrets (respectively) known to the party: `know_key[i]` $=$ `true` if and only if the party knows the `i`-th secret key, and `know_secret[i]` $=$ `true` if and only if the party knows the `i`-th secret string. The integer `known_signatures_size` describes the number of the additional signatures known to the party (i.e. received from other parties during the protocol), and the array `known_signatures` contains these signatures.

2.4 The adversary

The real-life Bitcoin adversary can create an arbitrary number of transactions with arbitrary output scripts, so it is clear that we need to somehow limit his possibilities, so that the space of possible states is finite and of a reasonable size. We show that without loss of generality we can consider only scenarios in which an adversary sends to the block chain transactions only from a finite set depending only on the protocol.

The knowledge of an adversary is modeled in the similar way to honest parties, but we do not specify the protocol that he follows. Instead, we use a generic automaton, which (almost) does not depend on the protocol being verified and allows to send to the block chain any transaction at any time assuming some conditions are met, e.g. that the transaction is valid and that the adversary is able to create its input script.

We observe that the transactions made by the adversary can influence the execution of the protocol only in two ways: either (1) the transaction is identical to the transaction from the protocol being verified or (2) the transaction redeems one of the transactions from the protocol. The reason for above is that honest parties only look for transactions of the specific form (as in the protocol being executed), so the only thing an adversary can do to influence this process is to create a transaction, which looks like one of these transactions or redeem one of these. Notice that we do not have to consider transactions with multiple inputs redeeming more than one of the protocol's transactions, because there is always an interleaving in which the adversary achieves the same result using a number of transactions with single inputs. The output scripts in the transactions of type (2) do not matter, so we may assume that an adversary always sends them to one particular key known only to him.

Therefore, without loss of generality we consider only the transactions, which appear in the protocol being verified or transactions redeeming one of these transactions. Hence, the total number of transactions, which are modeled in our system is twice as big as the number of transactions in the original protocol.

The adversary is then a party, that can send an arbitrary transaction from this set if only he is able to do so (e.g. he is able to evaluate the input script and the transaction's input is confirmed, but not spent). If the only actions of the honest parties is to post transactions on the block chain, then one can assume that this is also the only thing that the adversary does. In this case his automaton, denoted `Adversary` is very simple: it contains one state and one loop, that simply tries to send an arbitrary transaction from the mentioned set. This is depicted in Fig. 5 on page 10 (for a moment ignore the left loop).

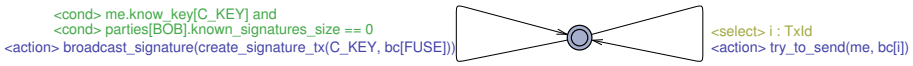


Fig. 5. The automaton for the Adversary

In some protocols the parties besides of posting the transactions on the block chain, also exchange messages with each other. This can be exploited by the adversary, and hence we need to take care of this in our model. This is done by adding more actions in the `Adversary` automaton. In our framework, this is done manually. For example in the protocol that we analyze in Sec. 3 Alice sends a signature on the $Fuse$ transaction (cf. Step 3, Fig. 9, Appendix B). This is reflected by the left loop in the `Adversary` automaton in Fig. 5, which should be read as follows: if the adversary is able to create a signature on the $Fuse$ transaction and Bob did not receive it yet, then he can send it to Bob.

Of course, our protocols need to be analyzed always "from the point of view of an honest Alice" (assuming Bob is controlled by the adversary) and "from the point of view of an honest Bob" (assuming Alice is controlled by the adversary). Therefore, for each party we choose whether to use the automaton describing the protocol executed by the parties or the already mentioned special automaton for an adversary.

2.5 The block chain and the notion of time

In Bitcoin whenever a party wants to post a transaction on the block chain she broadcasts it over a peer-to-peer network. In our model this is captured as follows. We model the block chain as a shared structure denoted `bc` containing the information about the status of all the transactions and a timed automaton denoted `BlockChainAgent` (see Fig. 6), which is responsible for maintaining the state of `bc`. One of the duties of `BlockChainAgent` is ensuring that the transactions which were broadcast are confirmed within appropriate time frames.

In order to post a transaction `t` on the block chain, a party `p` first runs the `try_to_send(` `Party` `p`, `Tx` `t`) function, which broadcasts the transaction if it is legal. In particular, the `can_send` function checks if (a) the transaction has not been already sent, (b) all its inputs are confirmed and unredeemed and (c) a given party `p` can create the corresponding input script. The only non-trivial part is (c) in case of non-standard transactions, as this check is protocol-dependent. Therefore, the exact condition on when the

party p can create the appropriate input script, has to be extracted manually from the description of the protocol. If all these tests succeed, then the function communicates the fact of broadcasting the transaction using the shared structure bc.

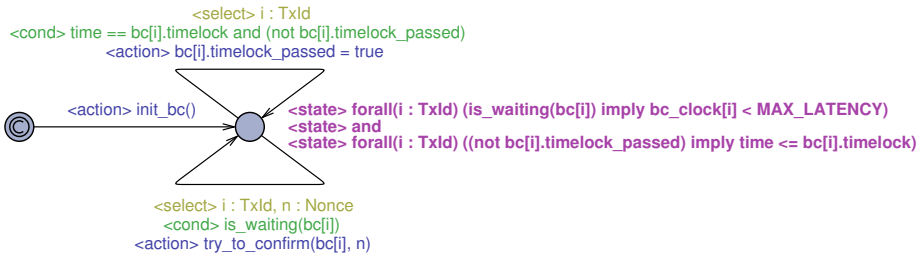


Fig. 6. The `BlockChainAgent` automaton

Once a transaction t has been broadcast, the `BlockChainAgent` automaton attempts to include it in the block chain (lower loop in Fig. 6). The `BlockChainAgent` automaton also enforces that every transaction gets included into the block chain in less than `MAX_LATENCY` time, which is a constant that is defined in the system. This is done by the invariant on the right state in Fig. 6 that guarantees that every transaction is waiting for confirmation less than `MAX_LATENCY`.

Eavesdropping on the network The other issue with the block chain is that the peers in the network can see transactions before they are confirmed. Therefore if a transaction t contains (e.g. in its input script) a secret string x then an adversary can learn the value of x before t is confirmed and for example use it to create a different transaction redeeming the input of t (a similar scenario is possible for a two-party lottery protocol from [8], which is only secure in a "private channel model"). To take such possibilities into account, broadcasting a transaction results in disclosure of the secret string x, what in our model corresponds to setting appropriate knowledge flags for all parties.

Malleability of transactions `BlockChainAgent` automaton is also responsible for choosing the nonces, which imitate the attacks involving the malleability of transactions. This is described in details in Appendix A.

3 Modeling the Bitcoin-based timed commitment scheme from [8]

In this section we describe the "contract-dependent" part of our model. Our method of modeling and verifying Bitcoin contracts as timed automata is generic and can be applied to a large class of Bitcoin contracts (and even possibly automatized as described in the paragraph "Future work" on page 3). However, it is easier to describe it using a concrete example. As a proof-of-concept we constructed the automata corresponding to

a very simple contract called the "Bitcoin-based timed commitment scheme" from [8]. For the lack of space we only sketch informally what the protocol is supposed to do. In the protocol one of the parties (called Alice) commits herself to a secret string s. A key difference between this protocol and classic commitment schemes is that Alice is forced to open the commitment (i.e. reveal the string s) until some agreed moment of time (denoted PROT_TIMELOCK) or pay 1 ₿ to Bob. The full description can be found in Appendix B. Although the verification of correctness is quite straightforward in this case, we would like to stress that our method is applicable to more complicated contracts, like the NewSCS protocol from [8] (see Section 3.2), for which the correctness is much less obvious.

3.1 The results of the verification

Before running the verification procedure in UPPAAL it is necessary to choose, which parties are honest and which are malicious.

In UPPAAL it is done by selecting an automaton following the protocol or the malicious automaton for an adversary described in Sec. 2.4 for each of the parties. We started with verification of the security from the point of view of honest Bob. To this end we used an honest automaton for Bob (see Fig. 7) and an adversary automaton described before for Alice (see Fig. 5).

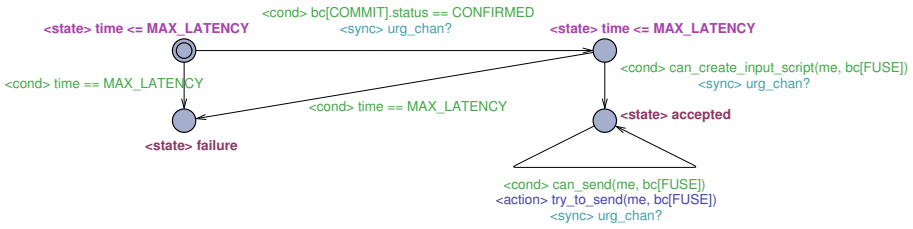


Fig. 7. The automaton for an honest Bob in timed-commitment scheme

The property that we checked is the following:

```
A[] (time >= PROT_TIMELOCK+MAX_LATENCY) imply
      (hold_bitcoins(parties[BOB]) == 1 or parties[BOB].know_secret[0]
                                        or BobTA.failure),
```

which, informally means: "after time PROT_TIMELOCK + MAX_LATENCY one of the following cases takes place: either (a) Bob earned 1 ₿, or (b) Bob knows the committed secret, or (c) Bob rejected the commitment in the commitment phase". This is exactly the security property claimed in [8], and hence the verification confirmed our belief that the protocol is secure. We verified the security from the point of view of Alice in the similar way.

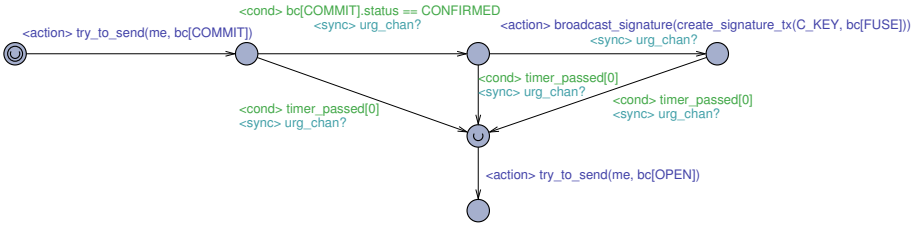


Fig. 8. The automaton for an honest Alice in timed-commitment scheme

The property we verified means that Alice does not lose any bitcoins in the execution of the protocol (even if Bob is malicious).

As a test we also run the verification procedure on the following two statements:

```
A[] (time >= PROT_TIMELOCK) imply (parties[BOB].know_secret[0])
A[] (time >= PROT_TIMELOCK) imply (hold_bitcoins(parties[ALICE]) == 1).
```

The first one states that after time `PROT_TIMELOCK` Bob knows the secret (which can be not true, if Alice refused to send it). The second one states that after time `PROT_TIMELOCK` Alice holds 1 ₿ (which occurs only if Alice is honest, but not in general). The UPPAAL model checker confirmed that these properties are violated if one of the parties is malicious, but hold if both parties follow the protocol (i.e. when honest automata are used for both parties). Moreover, UPPAAL provides diagnostic traces, which are interleavings of events leading to the violation of the property being tested. They allow to immediately figure out, why the given property is violated and turned out to be extremely helpful in debugging the automata.

3.2 The NewSCS protocol from [8]

We also modeled and verified the Simultaneous Commitment Scheme (NewSCS) protocol from [8], which is relatively complicated as it contains 18 transactions. To understand it fully the reader should probably look in the [8], but as reference we included the description of these contracts in Appendix C. Informally speaking, the NewSCS scheme is a protocol that allows two parties, Alice and Bob, to simultaneously commit to their secrets (s_A and s_B, respectively) in such a way that each commitment is valid only if the other commitment was done correctly. Using UPPAAL we automatically verified the following three conditions, which are exactly the security statements claimed in [6]:

- After the execution of the protocol by two honest parties, they both know both secrets and hold the same amount of coins as at the beginning of the protocol, which in UPPAAL syntax was formalized as:

```
A[] (time >=  PROT_TIMELOCK+MAX_LATENCY) imply
   (parties[ALICE].know_secret[SB_SEC] and parties[BOB].know_secret[SA_SEC]
     and hold_bitcoins(parties[ALICE]) == 2
     and hold_bitcoins(parties[BOB]) == 2)
```

  (here `SA_SEC` and `SB_SEC` denote the secrets of Alice and Bob, respectively, and 2 is the value of the deposit).
- An honest Bob cannot lose any coins as a result of the protocol, no matter how the dishonest Alice behaves:

```
2) A[] (time >= PROT_TIMELOCK) imply hold_bitcoins(parties[BOB]) >= 2
```

– If an honest Bob did not learn Alice's secret then he gained Alice's deposit as a result of the execution.

```
3) A[] ((time >= PROT_TIMELOCK+2*MAX_LATENCY) imply
         ((parties[ALICE].know_secret[SB_SEC]
             and !parties[BOB].know_secret[SA_SEC])
         imply hold_bitcoins(parties[BOB]) >= 3))
```

The analogous guarantees hold for Alice, when Bob is malicious. The verification of each of the mentioned properties took less than a minute on a dual-core 2.4 GHz notebook. We confirmed that the protocol NewSCS is correct, but there are some implementation details, which are easy to miss and our first implementation (as an automaton) turned out to contain a bug, which was immediately found due to the verification process and diagnostic traces provided by UPPAAL. We describe this in more detail in Appendix C.1. Moreover UPPAAL turned out to be very helpful in determining the exact time threshold for the time locks, for example we confirmed that the time at which the parties should abort the protocol claimed in [7] ($t - 3$MAX_LATENCY) is strict.

These experiments confirmed that the computer aided verification and in particular UPPAAL provides a very good tool for verifying Bitcoin contracts, especially since it is rather difficult to assess the correctness of Bitcoin contracts by hand, due to the distributed nature of the block chain and a huge number of possible interleavings. Therefore, we hope that our paper would encourage designers of complex Bitcoin contracts to make use of computer aided verification for checking the correctness of their constructions.

References

1. R. Alur and D. L. Dill. Automata for modeling real-time systems. In *ICALP'90*.
2. R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 1994.
3. R. Alur and T. A. Henzinger. A really temporal logic. *Journal of the ACM*, 1994.
4. T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. TIMES - a tool for modelling and implementation of embedded systems. TACAS '02.
5. Marc Andreessen. Why Bitcoin Matters, Jan 2013. The New York Times, dealbook.nytimes.com/2014/01/21/why-bitcoin-matters, accessed on 26.01.2014.
6. M. Andrychowicz, S. Dziembowski, D. Malinowski, and Ł. Mazurek. Fair two-party computations via the bitcoin deposits. Cryptology ePrint Archive, Report 2013/837, 2013. http://eprint.iacr.org/2013/837, accepted to the 1st Workshop on Bitcoin Research.
7. M. Andrychowicz, S. Dziembowski, D. Malinowski, and Ł. Mazurek. How to deal with malleability of Bitcoin transactions. *ArXiv e-prints*, December 2013.
8. M. Andrychowicz, S. Dziembowski, D. Malinowski, and Ł. Mazurek. Secure Multiparty Computations on Bitcoin. Cryptology ePrint Archive, 2013. http://eprint.iacr.org/2013/784, accepted to the 35th IEEE Symposium on Security and Privacy (Oakland) 2014.
9. Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on uppaal 4.0, 2006.
10. Bernard Berthomieu and Michel Diaz. Modeling and verification of time dependent systems using time Petri nets. *IEEE Trans. Softw. Eng.*, 17(3):259–273, March 1991.
11. Bitcoin. Wiki. en.bitcoin.it/wiki/.
12. H. Delfs and H. Knebl. *Introduction to Cryptography: Principles and Applications*. Information Security and Cryptography. Springer, 2007.

13. D. Dolev and A. C. Yao. On the security of public key protocols. *Information Theory, IEEE Transactions on*, 1983.
14. Danny Dolev, Cynthia Dwork, and Moni Naor. Nonmalleable cryptography. *SIAM Journal on Computing*, 30(2):391–437, 2000.
15. C. Dwork and M. Naor. Pricing via processing or combatting junk mail. In *CRYPTO*, 1992.
16. The Economist. The Economist explains: How does Bitcoin work?, Apr 2013. www.economist.com/blogs/economist-explains/2013/04/economist-explains-how-does-bitcoin-work, accessed on 26.01.2014.
17. G. Ateniese et al. Certified bitcoins. Cryptology ePrint Archive, Report 2014/076.
18. J. Bengtsson et al. UPPAAL - a tool suite for automatic verification of real-time systems. In *Hybrid Systems*, volume 1066 of *Lecture Notes in Computer Science*, 1995.
19. S. Barber et al. Bitter to better - how to make bitcoin a better currency. FC'12.
20. J. Katz and Y. Lindell. *Introduction to Modern Cryptography (Chapman & Hall/Crc Cryptography and Network Security Series)*. Chapman & Hall/CRC, 2007.
21. T. B. Lee. Here's how Bitcoin charmed Washington. www.washingtonpost.com/blogs/the-switch/wp/2013/11/21/heres-how-bitcoin-charmed-washington, accessed on 26.01.2014.
22. David Z. Morris. Bitcoin is not just digital currency. It's Napster for finance, Jan 2014. CNN Money, finance.fortune.cnn.com/2014/01/21/bitcoin-platform, accessed on 26.01.2014.
23. S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.
24. Xavier Nicollin and Joseph Sifakis. The algebra of timed processes, atp: Theory and application. *Inf. Comput.*, 114(1):131–178, 1994.
25. R. Cohen. Global Bitcoin Computing Power Now 256 Times Faster Than Top 500 Supercomputers, Combined! Forbes, www.forbes.com/sites/reuvencohen/2013/11/28/global-bitcoin-computing-power-now-256-times-faster-than-top-500-supercomputers-combined/.
26. G. M. Reed and A. W. Roscoe. A timed model for communicating sequential processes. *Theor. Comput. Sci.*, 58(1-3):249–261, June 1988.
27. Bitcoin wiki. Contracts. en.bitcoin.it/wiki/Contracts, Accessed on 26.01.2014.
28. Bitcoin wiki. Script. en.bitcoin.it/wiki/Script, Accessed on 26.01.2014.
29. Bitcoin wiki. Transaction malleability. en.bitcoin.it/wiki/Transaction_Malleability, Accessed on 26.01.2014.
30. Wang Yi. CCS + time = an interleaving model for real time systems. In *Automata, Languages and Programming*, volume 510 of *Lecture Notes in Computer Science*. 1991.
31. Sergio Yovine. Kronos: a verification tool for real-time systems. *Journal on Software Tools for Technology Transfer*, 1, October 1997.

A Malleability of transactions

Let us now describe the reason why we introduced the variables of a type **`Nonce`** in the transaction and the signatures. The reason is that they are used to model the fact that the transactions in Bitcoin can be slightly modified without changing its functionality. Malleability is a very general concept introduced in cryptography in a seminal paper of Dolev et al. [14]. It is a (usually) undesired property of the cryptographic schemes that very informally means that an adversary, after seeing an output of a scheme, can produce another output that is in some non-trivial way "related" to the original output.[5] As it turns out, malleability of transaction is a feature of Bitcoin, which poses a serious

[5] For example, the one time pad encryption scheme (see, e.g., [20]) defined as: $\mathrm{Enc}(K, M) := K \oplus M$ and $\mathrm{Dec}(K, C) := K \oplus C$ (where "$\oplus$" denotes the coordinate-wise xor) *is* malleable since by negating every bit in C one obtains a ciphertext C' of a message M' that is equal

risk for almost all contracts using time locks. We now briefly describe this problem. More detailed descriptions can be found in [7,29].

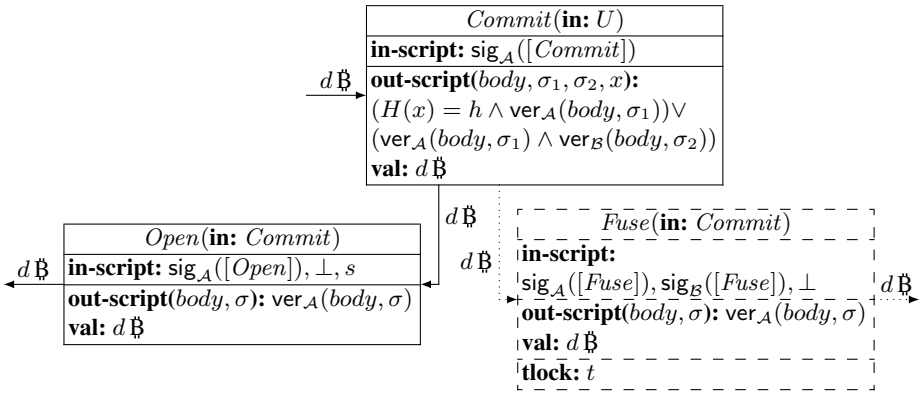
Bitcoin transactions are malleable in the following way: given a valid transaction t, an adversary is able to create a functionally equivalent and valid transaction t' which has a different hash, even if he does not know the secret key used to produce t. Both transactions have the same input transactions, the same output script and differ only in the hash. Therefore, the following scenario is possible. A transaction t is sent to the block chain, which technically means that it is broadcast over the network. An adversary can therefore see t and create and broadcast t'. If he is lucky then eventually t' becomes included in the block chain, instead of t. It means that we can not assume the knowledge of the hash of the transaction before it is confirmed. It poses a serious problem for most of the protocols using time locks, because such protocols often need to sign a transaction s which redeems t, *before* t is broadcast. Such transaction s contains a hash of t and in case when t' is included in the block chain instead of t, the transaction s becomes invalid.

In our model the transactions are addressed by fixed identification numbers instead of hashes, so a special technique should be applied to take the malleability of transactions into account. To this end we extend the transaction structure with a "malleability nonce", which contains an integer from a fixed and small set $\mathbf{Nonce}$ (in most cases a set $\mathbf{int}[0, 1]$ of size 2 is enough). Each transaction, which has not been sent to the block chain has the `nonce` field equal to 0, which indicates that the transaction was not modified. Whenever a transaction is being confirmed, its malleability nonce is set to a random value. Moreover, the structure describing the signature on the transaction t which redeems s (see Sec. 2.1) contains the assumed malleability nonce of s (set when the transaction t was being signed) and is considered valid only if this nonce matches the real nonce of the transaction s. Therefore, if the transaction s was confirmed after the signature structure was created, the signature may or may not become invalidated, what catches the issue of malleability in real Bitcoin network.

The above solution with "malleability nonces" makes the assumption that whenever s is modified, then only the transaction t becomes invalid. In reality it would also influence another transaction u which redeems t and so on. However, in all Bitcoin protocols we are aware of, this is not an issue as the signatures are computed at most "one step forward". It is also not difficult to extend the model to handle "malleability nonce of the second (or arbitrary) level" if necessary.

to a "negated" M. Note that this works even if one does not know K. Since the one-time pad is perfectly secure, thus the non-malleability is not implied by the standard security of an encryption scheme.

B Bitcoin-based timed commitment scheme



Pre-condition:

1. The key pair of Alice is $\mathcal{A}$ and the key pair of Bob is $\mathcal{B}$.
2. Alice already sent to Bob the value $h = H(s)$, where s is the value she wants to commit to.
3. The block chain contains a confirmed and unredeemed transaction U, which can be redeemed with key $\mathcal{A}$ of value d ₿.

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

1. Alice sends to the block chain the transactions $Commit$.
2. If within time `MAX_LATENCY` the $Commit$ transaction does not appear on the block chain, or if it looks incorrect (e.g. it differs in the h value) then Bob aborts.
3. Alice creates the body of the transaction $Fuse$, signs it and sends the signature to Bob.
4. Bob accepts the commitment if he receives the appropriate signature from Alice.

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

5. Alice sends to the block chain the transactions $Open$, what reveals the secret string s.
6. If within time t the transaction $Open$ does not appear on the block chain then Bob sends to the block chain the transaction $Fuse$ and earns d ₿.

Fig. 9. The $\mathsf{CS}(Alice, Bob, d, t, s)$ protocol. The scripts' arguments, which are omitted are denoted by $\perp$.

A *commitment scheme* is one of the basic cryptographic primitives (see, e.g., [12]). It is executed between two parties: Alice and Bob. In the first phase (called, the *commitment phase*) Alice *commits* herself to some secret value s. That means that she sends to Bob a value $c = f(s)$ for some agreed randomized function f. In the second phase (called the *opening phase*) Alice sends to Bob s plus some extra information.

From the commitment scheme we require two properties — *hiding* and *binding*. These properties will be satisfied if we use f as $f(s) = H(s||r)$ where H is a cryptographic hash function and r is a random, fix length padding (if we model H as a random oracle). In the Bitcoin-based timed commitment scheme this construction is used as a building block (with often used in Bitcoin function SHA-256 as a hash function). In its description we assume, that the padding was already added and simply use H instead of f.

One of the problems with the classical commitment scheme is that there is no way to force Alice to open the commitment and hence reveal the secret. To remedy this, in [8] we proposed to use the Bitcoin system to punish financially Alice in case she does not open the commitment. This is done as follows. Assume that Alice wants to commit herself to some secret s. To do it, she creates and sends to the block chain a Bitcoin transaction ($Commit$) that contains a commitment $f(s)$ and has some value d ฿ (this money is called the "deposit"). This transaction can be spent in one of two ways: either by revealing the secret (this is an expected action of Alice, she will have to reveal the secret to get her money back), or the signatures of both Alice and Bob (by this Bob can punish Alice). After the transaction is included in the block chain, Alice creates the $Fuse$ transaction — it spends the $Commit$ transaction and sends the money to Bob. Alice sets a time lock t (which forces her to open the commitment by time t) adds her signature and sends it to Bob. Bob accepts the commitment if the transactions are correct (e.g. they have proper values, time lock, and $Fuse$ has a good signature) and adds his signature to the $Fuse$ transaction. Otherwise he quits. Alice should open the commitment (i.e. send to the block chain the transaction $Open$ spending the $Commit$ transaction; it will contain the secret) before the time lock ends to get back her money. If she does not do that, then Bob sends the $Fuse$ transaction to get agreed amount of bitcoins. The graph of transactions and a detailed description of the protocol is presented in Fig. 9. We require that this scheme, besides of being binding and hiding, has the following extra properties:

1. honest Alice will never lose her money and she will always open the commitment,
2. if both Alice and Bob are honest, then Bob will accept the commitment,
3. if honest Bob accepts the commitment, then he will learn the secret or gain agreed value of bitcoins.

We verified these conditions using UPPAAL and the model described in this paper (see Sec. 3.1).

C Simultaneous Commitment scheme from [7]

Pre-conditions:

1. A holds the key pair A and B holds the key pair B.
2. A knows the secret s_A, B knows the secret s_B, both players know the hashes $h_{s_A} = H(s_A)$ and $h_{s_B} = H(s_B)$.
3. There are four unredeemed transactions T_1^A, T_2^A and T_1^B, T_2^B, which can be redeemed with the keys A and B respectively, each having the value of d ₿.

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

1. A draws a random string r_A and B draws a random string r_B.
2. The parties execute $\mathsf{CS.Commit}(\mathsf{A}, \mathsf{B}, d, t, r_A)$ and $\mathsf{CS.Commit}(\mathsf{B}, \mathsf{A}, d, t, r_B)$ using T_1^A and T_1^B respectively. The former execution will be denoted CS^A and the latter CS^B. Recall that the parties quit the whole NewSCS protocol if they detect the misbehavior of the other party during one of the CS.Commit executions.

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

3. Both players construct the body of the transaction $Commit$ using T_2^A and T_2^B as inputs.
4. A signs the transaction $Commit$ and sends the signature to B.
5. B signs the transaction $Commit$ and broadcasts it.
6. Both parties wait until the transaction $Commit$ is confirmed.
7. If the transaction $Commit$ does not appear on the blockchain until the time $t - 3\mathsf{max}_{\mathsf{BB}}$, where $\mathsf{max}_{\mathsf{BB}}$ is the maximal possible delay between broadcasting the transaction and including it in the blockchain, then A immediately redeems the transaction T_2^A and after T_2^A is redeemed she opens her CS^A commitment and quits the protocol. Analogously, if A did not send her signature to B until the time $t - 3\mathsf{max}_{\mathsf{BB}}$, then B opens his CS^B commitment and quits the protocol.

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

8. A and B broadcast the transactions $Open^A$ and $Open^B$ respectively, what reveals the secrets s_A and s_B.
9. After the transactions $Open^A$ and $Open^B$ are confirmed, A and B open their CS commitments.
10. If A did not broadcast $Open^A$ until time t, then depending on whether she opened her commitment CS^A or not, B broadcasts $Fuse^A$ or $\mathsf{CS}^A.Fuse$ to get extra d ₿ (in addition to $2d$ ₿ already claimed from $Open^B$ and $\mathsf{CS}^B.Open$). Similarly, if B misbehaved then A broadcasts $Fuse^B$ or $\mathsf{CS}^B.Fuse$ to get her extra d ₿.

Fig. 10. The description of the NewSCS protocol.

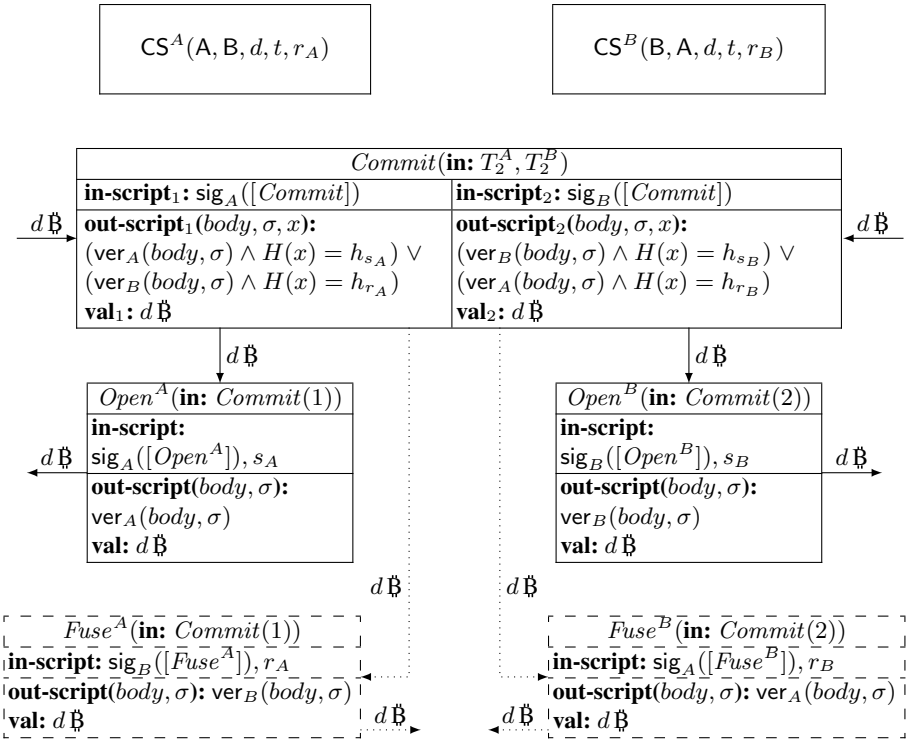


Fig. 11. The graph of transactions for the NewSCS protocol. Two boxes labeled with $\mathsf{CS}(\ldots)$ denote the transactions broadcast in the appropriate execution of the Bitcoin-based timed commitment scheme. h_x denotes the value $H(x)$, but it is used in the output scripts to stress that the value of the hash is directly included in the transaction (instead of value of x and an application of the hash function).

C.1 A bug in the first version of NewSCS

As mentioned in Section 3.2 the protocol NewSCS is correct, but there are some implementation details, which are easy to miss and our first implementation turned out to contain a bug, which was immediately found due to verification process. More precisely, in Step 10 of the NewSCS protocol it is said that Bob sends to the block chain one of the two transactions: $Fuse^A$ or $\mathtt{CS}^A.Fuse$ and as a result gains 1 bitcoin, what means that at least one of these two transactions becomes confirmed. Therefore, our first implementation just tried to send these two transactions. However, it turns out that there is a strategy of malicious Alice and an interleaving of events in which none of these transactions becomes confirmed and it is necessary to try to sends the transaction $Fuse^A$ again after waiting for time `MAX_LATENCY`. UPPAAL provides diagnostic traces, which allows to easily find bugs like the one mentioned.

D UPPAAL

D.1 A very short introduction to timed automata and UPPAAL

For the lack of space we only sketch the description of UPPAAL and the semantics of the timed automata (focusing on the most relevant features). The reader may consult [9,18] for more information on this topic.

In UPPAAL the automata are generally finite-state automata equipped with clocks. UPPAAL system consists of some number of timed automata, clocks and variables of discrete, bounded sets (also user defined, like records) and functions working on variables and clocks. The state of the system consists of locations of the automata, values of the variables and the clocks evaluation. The clocks evaluate to positive real numbers, but because of bounded set of possible constraints number of different state it which the system can be is finite.

The only possible transitions in UPPAAL are moving all the clocks forward with the same value or to use some of edges to change the state of some of the automata. Both kind of transitions are correct only if all *invariants* and *guards* are satisfied. The invariants are properties of the locations and have to be satisfied each time the system is in this location. The locations may have also names (used in a verification), be urgent (time cannot pass when any automaton is in such location) or committed (the system immediately has to use an edge outgoing from such location). Moreover, edges can use selectors to set a local variable to any value of some type. When an edge is used, it may also run a connected update — it changes values of variables and resets some clocks. The pairs of edges may also be synchronized — in such case they can be used, but only together. The synchronization may also be urgent — then it has to be used if only it can be used. More details and a syntax used in UPPAAL can be found in Appendix D.2.

UPPAAL comes also with a simulator (to e.g. run transitions in a random order) and a verifier. The verifier checks whether the given properties (written in the simplified version of TCTL — timed computation tree logic) are satisfied by the system.

D.2 UPPAAL syntax

Let us analyze the UPPAAL syntax on the example of automaton for Bob from a timed commitment scheme (Fig. 12). It has 4 states: the upper-left is a starting state, the upper-right corresponds to a situation when the $Commit$ transaction is confirmed, but the signature on $Fuse$ from Step. 3 has not yet been received by Bob. The lower-left state (denoted *failure*) means that the commitment was not accepted by Bob and the lower-right (denoted *accepted*) state means that Bob accepted the commitment (Step 4).

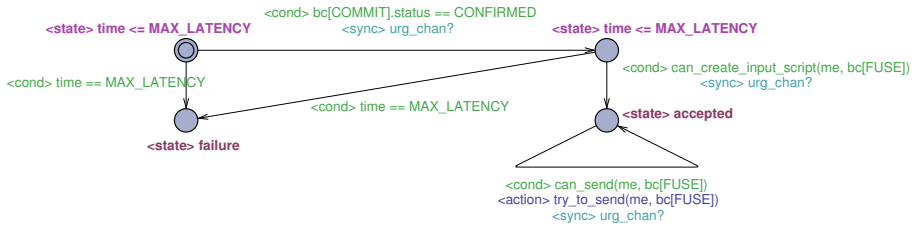


Fig. 12. The automaton for Bob in timed-commitment scheme

There are 5 types of labels on the picture[6]:

<select> These labels are placed on the edges to set a local variable used in following condition and/or action to any value from a specified set, e.g. `i : TxId`, `cond(i)`, `action(i)` placed on an edge means that this edge can be used for any `i` from `TxId`, provided `cond(i)` is true and then `action(i)` is performed. The mentioned automation for Bob does not have any edges of this kind.

<cond> These labels are placed on the edges and they represent conditions (called *guards* in UPPAAL), which have to be satisfied, e.g. `bc[COMMIT].status == CONFIRMED` means that this transition can be taken only if the transaction *Commit* has been already confirmed and `can_create_input_script(me, bc[FUSE])` checks if Bob is able to create input script for *Fuse* transaction, what is equivalent to the fact that he has already received the signature from Step. 3 (Fig. 9).

<action> These labels are placed on the edges and they represent actions (called *updates* in UPPAAL), i.e. functions which are called whenever a transition is taken. In our example there is just one action `try_to_send(me, bc[FUSE])`, which checks whether it is possible to send the given transaction in the current state (e.g. it was not sent earlier, its inputs are confirmed, but not spent, the current party is able to evaluate the corresponding input scripts) and changes the state of the transaction to `SENT` if all conditions are met.

<state> These labels, placed on the states, represent names of the states (*failure* and *accepted*) and *invariants* specifying conditions, which has to be satisfied in the given state. Execution in which the invariant is not satisfied are ignored by UPPAAL. Therefore a node with an invariant `time <= MAX_LATENCY` and an outgoing edge with a guard `time == MAX_LATENCY` guarantee that the edge will be taken exactly at time `MAX_LATENCY` (Moreover it makes impossible to enter the state after time `MAX_LATENCY`, but it is not important in our example.)

<sync> These labels, placed on the edges, represent synchronization channels. Here they are used for a special purpose of obtaining urgent transitions. This is the technical trick described in Sec. D.3, but for understanding this picture it is enough to know that the automaton is not allowed to wait whenever there is an edge available with the label `urg_chan?`.

[6] We color them using the default UPPAAL coloring. We also decided to add special text labels (in the "<>" brackets) in the Figures of automata to make the text readable when printed black and white

D.3 Helper automaton

A transition is called *urgent*, when it has to be taken immediately whenever it is possible. More precisely, the time cannot pass, whenever there is an automaton with an urgent transaction available (i.e. with a satisfied guard). UPPAAL does not provide urgent transitions, but there exists a simple workaround, described e.g. in [9]. The solution is based on the so-called urgent *channels*. Here, the "urgency" means that the time cannot pass whenever there is an automaton with an available transition synchronizing on an urgent channel (availability means that in particular there is another automaton, which can synchronize on the same channel). Therefore, it is enough to mark edges we would like to be urgent with a synchronization label `urg_chan`? and create a special automaton (called `Helper` in our implementation) with one state and a loop with label `urg_chan`! for some urgent channel `urg_chan`.

UPPAAL does not allow to put guards involving clocks on edges with synchronization on urgent channels, so another workaround is needed to achieve this functionality. The simple solution is to check on the edge a value of some boolean shared variable and make sure that the value of this variable is always the same as the value of the clock condition we would like to have on the edge. The correct value of the shared variable can be maintained by another loop in the `Helper` automaton. The `Helper` automaton is presented in Fig. 13.

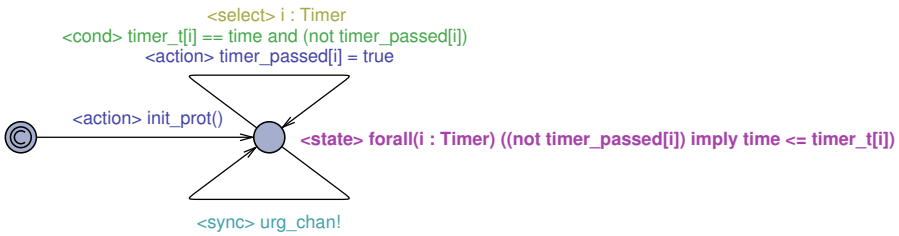


Fig. 13. The `Helper` automaton

On the Malleability of Bitcoin Transactions*

Marcin Andrychowicz**, Stefan Dziembowski***, Daniel Malinowski†, Łukasz Mazurek‡

University of Warsaw

Abstract. We study the problem of the malleability of the Bitcoin transactions. Our first two contributions can be summarized as follows:

(i) we perform practical experiments on Bitcoin that show that it is very easy to maul Bitcoin transactions with high probability, and

(ii) we analyze the behavior of the popular Bitcoin wallets in the situation when their transactions are mauled; we conclude that most of them are to some extend not able to handle this situation correctly.

The contributions in points (i) and (ii) are experimental. We also address a more theoretical problem of protecting the Bitcoin distributed contracts against the "malleability" attacks. It is well-known that malleability can pose serious problems in some of those contracts. It concerns mostly the protocols which use a "refund" transaction to withdraw a financial deposit in case the other party interrupts the protocol. Our third contribution is as follows:

(iii) we show a general method for dealing with the transaction malleability in the Bitcoin contracts. In short: this is achieved by creating a malleability-resilient "refund" transaction which does not require any modification of the Bitcoin protocol.

1 Introduction

Malleability is a term introduced in cryptography by Dolev et al. [15]. Very informally, a cryptographic primitive is *malleable* if its output C can be transformed ("mauled") to some "related" value C' by someone who does not know the cryptographic secrets that were used to produce C. For example, a symmetric encryption scheme $(\mathrm{Enc}, \mathrm{Dec})$ is malleable if the knowledge of a ciphertext $C = \mathrm{Enc}(K, M)$ suffices to compute C' such that $M' = \mathrm{Dec}(K, C')$ is not equal to M, but is related to it (e.g. M' is equal to M with the first bit set to 0). It is easy to see that the standard cryptographic security definitions (like the semantic security of the encryption schemes) in general do not imply non-malleability, and hence the non-malleability is usually viewed as an additional (but often highly desirable) feature of the cryptosystems. Since its introduction the concept of non-malleability was studied profoundly, mostly by the theory community, in several different contexts including the encryption and commitment schemes, zero-knowledge [15], multiparty computation protocols [12], hash functions and one-way functions [10], privacy amplification [14], tamper-resilient encoding [16], and many others. Until last year, however, it remained largely out of scope of the interests of the security practitioners.

This situation has changed dramatically, when the MtGox Bitcoin exchange suspended its trading in February 2014, announcing that around 850,000 bitcoins belonging to customers were stolen by an attacker exploiting the "malleability of the Bitcoin transactions" [18]. Although there is a good

* 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. Moreover, Łukasz Mazurek is a recipient of the Google Europe Fellowship in Security, and this research is supported in part by this Google Fellowship.

** marcin.andrychowicz@crypto.edu.pl

*** stefan.dziembowski@crypto.edu.pl

† daniel.malinowski@crypto.edu.pl

‡ lukasz.mazurek@crypto.edu.pl

evidence that MtGox used the malleability only as an excuse [13], this announcement definitely raised the awareness of the Bitcoin community of this problem, and in particular, as argued in [13] it massively increased the attempts to exploit this weakness for malicious purposes.

The fact that the Bitcoin transactions are malleable has been known much before the MtGox collapse [21]. Briefly speaking, "malleability" in this case means that, given a transaction T, that transfers x bitcoins from an address A to address B (say), it is possible to construct another transaction T' that is syntactically different from T, but semantically it is identical (i.e. T' also transfers x bitcoins from A to B)[1]. This can be done by anybody, and in particular by an adversary who does not know A's private key. On a high level, the source of the malleability comes from the fact that in the current version of the Bitcoin protocol, each transaction is identified by a hash on its *whole* contents, and hence in some cases such a T' will be considered to be a different transaction than T.

There are actually several ways T' can be produced from T. One can, e.g. exploit the malleability of the signature schemes used in Bitcoin, i.e., the fact that given a signature σ (computed on some message M with a secret key sk) it is easy to compute another valid signature σ' on M (with respect to the same key sk)[2]. Since the standard Bitcoin transactions have a form $T =$ (message M, signature σ on M), thus $T' = (M, \sigma')$ is a valid transaction with the same semantics as T, but syntactically different from T. Another method is based on the fact that Bitcoin permits more complicated transactions than those in the format described above. More precisely, in the so-called "non-standard transactions" the "σ" part can be in fact a script in the *Bitcoin scripting language*. Since this language is stack-based, thus adding dummy PUSH and POP instructions to σ produces σ' that is operationally equivalent to σ, yet, from the syntactic point of view it is different. See, e.g, [22,13] for more detailed list of different ways in which the Bitcoin transactions can be mauled.

Recall that in order to place a transaction T on the block chain a user simply broadcasts T over the network. Thus it is easy for an adversary $\mathcal{A}$ to learn T before it is included in the block chain. Hence he can produce a semantically equivalent T' and broadcast T'. If $\mathcal{A}$ is lucky then the miners will include T' into the block chain, instead of T. At the first sight this does not look like a serious problem: since T' is equivalent to T, thus the financial effect of T' will be identical to the effect of T. The reason why malleability may cause problems is that typically in Bitcoin the transactions are identified by their hashes. More precisely (cf., e.g, [8]), an *identifier* (TXID) of every transaction $T = (M, \sigma)$ is defined to be equal to $H(M, \sigma)$, where H is the doubled SHA256 hash function. Hence obviously TXID of T will be different than TXID of T'.

There are essentially three scenarios when this can be a problem. The first one comes from the fact that apparently some software operating on Bitcoin does not take into account the malleability of the transactions. The alleged MtGox bug falls in this category. More concretely, the attack that MtGox claimed to be the victim of looked as follows: (1) a malicious user P deposits x coins on his MtGox account, (2) the client P asks MtGox to transfer his coins back to him, (3) MtGox issues a transaction T transferring x coins to P, (4) the user P launches the malleability attack, obtaining T' that is equivalent to T but has a different TXID (assume that T' gets included into the block chain instead of T), (5) the user complains to MtGox that the transaction was not performed, (6) MtGox checks that there is no transaction with the TXID $H(T)$ and concludes that the user is right, hence MtGox credits the money back to the user's account. Hence effectively P is able to withdraw his coins twice. The whole problem is that, of course, in Step (6) MtGox should have searched not for the transaction with TXID $H(T)$, but for any transaction semantically equivalent to T.

[1] For a short description of Bitcoin and the non-standard transactions see Section 2.

[2] This is because Bitcoin uses the Elliptic Curve Digital Signature Algorithm (ECDSA) that has the property that for every signature $\sigma = (r, s) \in \{1, \ldots, N-1\}^2$ the value $\sigma' = (r, N-s)$ is also a valid signature on the same message and with respect to the same key as σ. Moreover,

The second scenario is related to the first one in the sense that it should not cause problems if the users are aware that malleability attacks can happen. It is connected to the way in which the procedure of "giving change" is implemented in Bitcoin. Suppose a user A has $3\,฿$ in an unspent transaction T_0 on the block chain, and he wants to transfer $1\,฿$ to some other user B. Typically, what A would do in this case is: create a transaction T_1 that has input T_0 and has two outputs: one that can be claimed by B and has value $1\,฿$, and the one that can be claimed by himself (i.e. A) and has value $2\,฿$. He would then post T_1 on the block chain. If he now creates a further transaction T_2 that claims money from T_1 *without* waiting for T_1 to appear on the block chain, then he risks that T_2 will be invalid, in case someone mauls T_1.

The third scenario is much more subtle, as it involves the so-called *Bitcoin contracts* which are protocols to form financial agreements between mutually distrusting Bitcoin users. In this paper we assume readers familiarity with this Bitcoin feature. For an introduction to it see, e.g., [19,4,3] (in this paper we use the notation from [4,3]). Recall that contracts are more complicated than normal money transfers in the standard transactions. To accomplish their goal contracts use the non-standard transactions. One of the common techniques used in constructing such contracts is to let the users sign a transaction T_1 before its input T_0 is included in the block chain. Very informally speaking, the problem is that T_1 is constructed using the TXID $H(T_0)$, which means that an adversary that mauls T_0 into some (equivalent but syntactically different) T_0' can make the transaction T_1 invalid. There are many examples of such situations. One of them is the "Providing a deposit" contract from [19], which we describe in more detail in Section 4, where we also explain this attack in more detail. Note that, unlike in the first two scenarios, this problem cannot be mitigated by patching the software.

1.1 Possible fixes to the Bitcoin malleability problem

There are several ways in which one can try to fix the problems caused by the malleability of Bitcoin transactions. For example one can try to modify Bitcoin in order to eliminate malleability from it. Such proposals have indeed been recently put forward. In particular, Pieter Wuille [22] proposed a number of ad-hoc methods to mitigate this problem, by restricting the syntax of Bitcoin transactions. While this interesting proposal may work in practice, it is heuristic and it comes with no formal argument. In particular, it implicitly assumes that the only way to maul the ECDSA signatures is the one described in Footnote 2, and we are not aware of any formal proof that this is indeed the case.

In our previous paper [3] we proposed another modification of Bitcoin which eliminates the malleability problem. The idea of this modification is to identify the transactions by the hashes of their *simplified* versions (excluding the input scripts). With this modification one can of course still modify the input script of the transaction, but the modified transaction would have the same hash. Unlike [22] this solution does not rely on heuristic properties of the signature schemes. On the other hand, the proposal of [22] may be easier to implement in practice, since it requires milder modifications of the Bitcoin specification.

Another solution proposed recently by Peter Todd [1] is to introduce a new instruction `OP_CHECKLOCKTIMEVERIFY` for the Bitcoin scripting language that allows a transaction output to be made unspendable until some point in the future. It does not concern the problem of malleability directly, but using this opcode would allow to easily create Bitcoin contracts resilient to malleability.

Unfortunately, changing the Bitcoin is in general hard, since it is not controlled by any central authority, and hence the modifications done without proper care can result in a catastrophic fork, i.e. a situation where there is a disagreement among the honest parties about the status of transactions[3].

[3] An example of such a fork was experienced by the Bitcoin community in March 2013, when it was caused by a bug in a popular mining client software update [11]. Fortunately it was resolved manually, but it is still remembered as one of the moments when Bitcoin was close to collapse.

Hence it is not clear if such modifications will be implemented in close future. It is therefore natural to ask what can be done, assuming that the Bitcoin system remains malleable.

First of all, fortunately, as described above in many cases malleability is not a problem if the software is written correctly, and therefore the most obvious thing to do is the following.

Direction 1: Educate the Bitcoin software developers about this issue. Convince them that it is a real threat and they should always test their software against such attacks.

The only context in which the malleability cannot be dealt with by better programming are the Bitcoin contracts. Hence a natural research objective is as follows.

Direction 2: Develop a technique that helps to deal with the malleability of the Bitcoin transactions in the Bitcoin contracts.

The goal of this paper is to contribute to both of these tasks.

1.2 Our contribution

The technical contents of this paper is divided into two parts corresponding to the research directions described above. We first focus on "Direction 1" (this is done in Section 3). Since most of the software practitioners will probably only care about problems where the threat is real, not theoretic, we executed practical experiments that examine the feasibility of the malleability attacks. It turns out that these attacks are quite easy to perform, even by an attacker that has resources comparable to those of an average user. In fact, our experiments show that it is relatively easy to achieve success rates close to 50%.

We then analyze the behavior of popular Bitcoin clients when they are exposed to such attacks. Our results indicate that all of them show a certain resilience to such attacks. In particular we did not identify weaknesses that would allow users to steal money (as argued in Section 1.3 this is in fact something that one would expect from the beginning). On the other hand, we identified a number of smaller weaknesses in most of these clients. In particular, we observed that in many cases the malleability attack results in making the user unable to issue further transactions, unless he "resets" the client. In most cases such a reset can be performed relatively easy, in one case it required an intervention of a technically-educated user (restoring the backup files), and in two cases there seemed to be no way to perform such action.

This shows that some of the Bitcoin developers seem to still ignore the malleability problem, despite of the fact that over 8 months have passed since the infamous MtGox statement.

The second part (contained in Section 5) of this paper concerns the "Direction 2". We provide a general technique for transforming a Bitcoin contract that is vulnerable to the malleability attacks (but secure otherwise), into a Bitcoin contract that is secure against such attacks. Our method covers all known to us cases of such contracts, in particular, those listed on the "Contracts" page of the Bitcoin Wiki [19], and the lottery protocol of [5]. It can also be applied to [3], what gives the first fair Two-Party Computation Protocol (with financial consequences) for any functionality whose fairness is guaranteed by the Bitcoin deposits, and which, unlike the original protocol of [3] can be used on the current version of Bitcoin[4].

[4] The protocol of [3] was secure only under the assumption that the Bitcoin is modified to prevent the malleability attacks.

Related work. Some of the related work was already described in the previous sections. The idea of using Bitcoin to guarantee fairness in the multiparty protocols and to force the users to respect the outcome of the computation was proposed independently in [4,3] and in [6] (and implicitly in [5]), and was also studied in [7]. The protocols of [4] and [5] work only for specific functionalities (i.e. are not generic), and [5] is vulnerable to the malleability attack. The protocols of [3,6] are generic, but are insecure against the malleability attack. Also the protocol of [7] seems to be insecure against such attacks.

1.3 Ethical issues

We realize that performing the malleability attacks against the Bitcoin can raise questions about the ethical aspects of our work. We would like to stress that we were only attacking transactions that were issued by ourselves (and we never tried to maul transactions coming from third parties). It is also clear that performing such attacks cannot be a threat to stability of the whole Bitcoin system, since, as reported by [13] Bitcoin remained secure even against attacks on a much higher scale ([13] registered 25,752 individual malleability attacks involving 286,076 bitcoins just on two days of February 2014).

Let us also note that even before we started our work we could safely assume that none of the popular Bitcoin clients is vulnerable to the malleability attacks to the extent that would allow malicious users to steal money, as it is practically certain that any such weakness would be immediately exploited by malicious users. In fact, as argued in [13] such malicious attempts were probably behind the large number of malleability attacks immediately after the MtGox collapse. In other words: the experiments that we performed were almost certainly performed by several hackers before us. We believe that therefore making these results public is in the interest of the whole Bitcoin community.

2 Bitcoin description

We assume reader's familiarity with the basic principles of Bitcoin. For general description of Bitcoin, see e.g. [17,20,4]. For the description of non-standard transaction scripts, see [19,4,3]. Let us only briefly recall that the Bitcoin currency system consists of *addresses* and *transactions* between them. An address is simply a public key pk (technically an address is a *hash* of pk). We will frequently denote key pairs using the capital letters (e.g. A). We will also use the following convention: if $A = (sk, pk)$ then $\mathsf{sig}_A(m)$ denotes a signature on a message m computed with sk and $\mathsf{ver}_A(m, \sigma)$ denotes the result (true or false) of the verification of a signature σ on message m with respect to the public key pk.

Each Bitcoin transaction can have multiple inputs and outputs. Inputs of a transaction T_x are listed as triples $(y_1, a_1, \sigma_1), \ldots, (y_n, a_n, \sigma_n)$, where each y_i is a hash of some previous transaction T_{y_i}, a_i is an index of the output of T_{y_i} (we say that T_x *redeems the* a_i*-th output of* T_{y_i}) and σ_i is called an *input-script*. The outputs of a transaction are presented as a list of pairs $(v_1, \pi_1), \ldots, (v_m, \pi_m)$, where each v_i specifies some amount of coins (called the *value of the* i*-th output of* T_x) and π_i is an *output-script*. A transaction can also have a time-lock t, meaning that it is valid only if time t is reached. Hence, altogether transaction's most general form is: $T_x = ((y_1, a_1, \sigma_1), \ldots, (y_n, a_n, \sigma_n), (v_1, \pi_1), \ldots, (v_m, \pi_m), t)$. The *body of* T_x[5] is equal to T_x without the input-scripts, i.e.: $((y_1, a_1), \ldots, (y_n, a_n), (v_1, \pi_1), \ldots, (v_m, \pi_m), t)$, and denoted by $[T_x]$.

One of the most useful properties of Bitcoin is that the users have flexibility in defining the condition on how the transaction T_x can be redeemed. This is achieved by the input- and the output-scripts. One can think of an output-script as a description of a function whose output is Boolean. A

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

transaction T_x defined above is valid if for *every* $i = 1, \ldots, n$ we have that $\pi'_i([T_x], \sigma_i)$[6] evaluates to true, where π'_i is the output-script corresponding to the a_i-th output of T_{y_i}. Another conditions that need to be satisfied are that the time t has already passed, $v_1 + \cdots + v_m \leq v'_1 + \cdots + v'_n$ where each v'_i is the value of the a_i-th output of T_{y_i} and each of these outputs has not been already spent. The scripts are written in the Bitcoin scripting language. Following [4] we will present the transactions as boxes. The redeeming of transactions will be indicated with arrows (cf. e.g. Fig. 3). The transactions where the input script is a signature, and the output script is a verification algorithm are the most common type of transactions and are called *standard transactions*. The address against which the verification is done will be called a *recipient* of this transaction.

We use the security model defined in [4]. In particular, we assume that each party can access the current contents of the block chain, and post messages on it. Let Δ be the is maximal possible delay between broadcasting the transaction and including it in the block chain.

3 Experiments

The implementation of the malleability attack. In order to perform malleability attacks we have implemented a special program called adversary (using the *bitcoinj* [2] library). This program is connected as a peer to the Bitcoin network and listens for transactions sending bitcoins to a particular address $Addr$[7] owned by us. Whenever such a transaction is received by the program, it mauls the transaction[8] and broadcasts the modified version of the transaction.

The effectiveness of the attack is measured by the percent of cases in which the mauled transaction becomes confirmed and the original one invalidated. It depends on the fraction of the network (and hence miners), which receives the mauled transaction before the original one. In order to achieve a high effectiveness we need to push the modified transaction to the whole peer-to-peer network as fast as possible. Therefore, the adversary connects to many more peers than a typical Bitcoin client, more precisely it maintains on average 1000 connections[9]. Moreover, it connects directly to some nodes, which are known to be maintained by the mining pools[10] and sends the mauled transaction to them in the first place.

Effectiveness analysis In order to measure the effectiveness of our attack we set up another machine called victim, which makes hundreds of transactions sending bitcoins to the address $Addr$. More concretely we measured the effectiveness of mauling transactions under 3 different circumstances:

A) The IP address of a victim is not known to the adversary.
B) The IP address of a victim is known to the adversary. In this case the adversary can connect directly to the victim, what allows him to discover transactions broadcast by the victim much faster. In our experiments both machines — adversary and victim were located in the same local network, which in some cases can be possible also in real life (a motivated adversary can connect to the local network used by the victim). Our experiments showed that connecting directly to the victim greatly increases the effectiveness of the attack.

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

[7] In our experiments we used addresses `13eb7BFXgHeXfxrdDev1ehrBSGVPG6obu8` and `115g32FHp77hQpuuWpw8j8RYKPvxD1AXyP`.

[8] By changing (r, s) into $(r, N - s)$ in the ECDSA signature, see Footnote 2 on Page 2.

[9] Average Bitcoin client maintains about 8 connections.

[10] More precisely it connects to the nodes maintained by mining pools *GHash.IO* and *Eligius*.

C) The victim is aware of the malleability problem and tries to protect against it by broadcasting her transactions on a higher number of connections. In our experiment the victim was connected to 100 peers on the network and her IP address was not known to the adversary.

The results of our experiments are presented on Fig. 1. The effectiveness depends on the nodes to

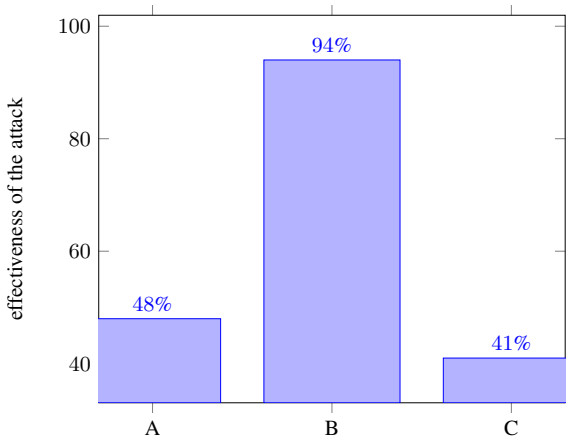


Fig. 1. Effectiveness of the attack under different circumstances.

which the victim is connected, so after each transaction she drops all her connections and establishes new ones. Moreover, the effectiveness depends on the distribution of mined blocks among miners in the testing period, so experiments were performed over longer periods of time (e.g. 24 hours). In order to exclude the influence of the factors like physical proximity of adversary and victim (in experiments A and C) we performed part of them with victim and adversary running on machines far away from each other.

Clients testing. In order to determine the significance of the malleability problem for individual Bitcoin users, we decided to test how the most popular Bitcoin wallets behave when a transaction is mauled. We have tested 14 Bitcoin wallets listed in [9]. In every test we performed several Bitcoin transfers from the wallet to *Addr*. During the tests the adversary was trying to maul every transaction addressed to this address.

We observed that (1) most of the clients determine whether the transaction is confirmed by looking for a transaction with a matching hash in the block chain, and (2) the clients are likely to receive from the network the modified transaction and list it in the account's history as a separate transaction. This can lead to unpredictable behavior of the wallet of many types, in most cases being hard to precisely describe. In Table 1 we present the results of our tests. In a nutshell, we have observed the following types of behavior of the clients:

(a) the wallet incorrectly computes the balance,
(b) the wallet is unable to make an outgoing transaction because it assumes that the conflicting transaction will be confirmed in the future,
(c) the application crashes.

Note that if the client refuses to make an outgoing transaction and it is not possible to remove the conflicting transaction from the history, the user will be unable to spend his bitcoins forever.

Moreover, because of the "giving change" procedure, an attack against a single transaction can potentially make all the money in the wallet unspendable (cf. the second scenario on Page 2).

Going more into the details: *Bitcoin Core* and *Xapo* appear to handle the malleability problem correctly. For example *Bitcoin Core* detects the malleability attack and marks it as "double spending" (indicating it with an exclamation mark). *Green Address* and *Armory* display incorrect balances (however, when it comes to allowing a user to make a transaction, they seem to take into account the real balance). *Blockchain.info*, *Coinkite*, *Coinbase*, *Electrum*, *MultiBit*, *Bitcoin Wallet*, and *KnC Wallet*, may display the conflicting transaction, which will never be confirmed and hence prevent the user from creating correctly the next transaction. Fortunately, these clients give the user an option to "reset the list of transactions", which makes this problem disappear. In *Hive* we were able the obtain the effect of "resetting" the transaction list only by a manual action (which may be non-trivial for a non-technically educated user). The problem is much more severe in case of *BitGo* and *Mycelium*, which also display the conflicting transaction, but there appears to be no way to reset the list. We note that, since BitGo is a web-client, thus the wrong transaction can be removed by the administrator.[11]

Wallet name	**Type**	**(a)**	**(b)**	**(c)**	**when the problem disappears**
Bitcoin Core	Desktop				-
Xapo	Web				-
Armory	Desktop	✘			never
Green Address	Destop	✘			never
Blockchain.info	Web	✘	✘		after six blocks without confirmation
Coinkite	Web	✘	✘		after several blocks without confirmation
Coinbase	Web	✘	✘		after several hours
Electrum	Desktop	✘	✘		after application reset
MultiBit	Desktop	✘	✘		after "Reset block chain and transactions" procedure
Bitcoin Wallet	Mobile	✘	✘		after "Reset block chain" procedure"
KnC Wallet	Mobile	✘	✘	✘	after "Wallet reset" procedure
Hive	Desktop	✘	✘	✘	after restoring wallet from backup
BitGo	Web	✘	✘		never
Mycelium	Mobile	✘	✘		never

Table 1. Results of testing 14 Bitcoin wallets listed on [9] against malleability attack. The tests took place in October 2014 and hence may not correspond to the current software version. Value *never* in the last column means that we could not figure out a way to solve the problem and it did not disappear on its own after a few days.

4 Malleability in Bitcoin Contracts

As shown in the earlier sections malleability of Bitcoin transactions can pose a problem to users if Bitcoin clients or services they are using have bugs in their implementation. But when these bugs are fixed then there should be no problems or dangers in typical usage of Bitcoin. Unfortunately malleability is a bigger problem for the security of the Bitcoin contracts. In this section we will describe a (known) malleability attack on a protocol for a deposit. Later we will also identify other protocols that are vulnerable to the malleability attack.

[11] In fact, the BitGo administrators reacted to our experiments by contacting us directly.

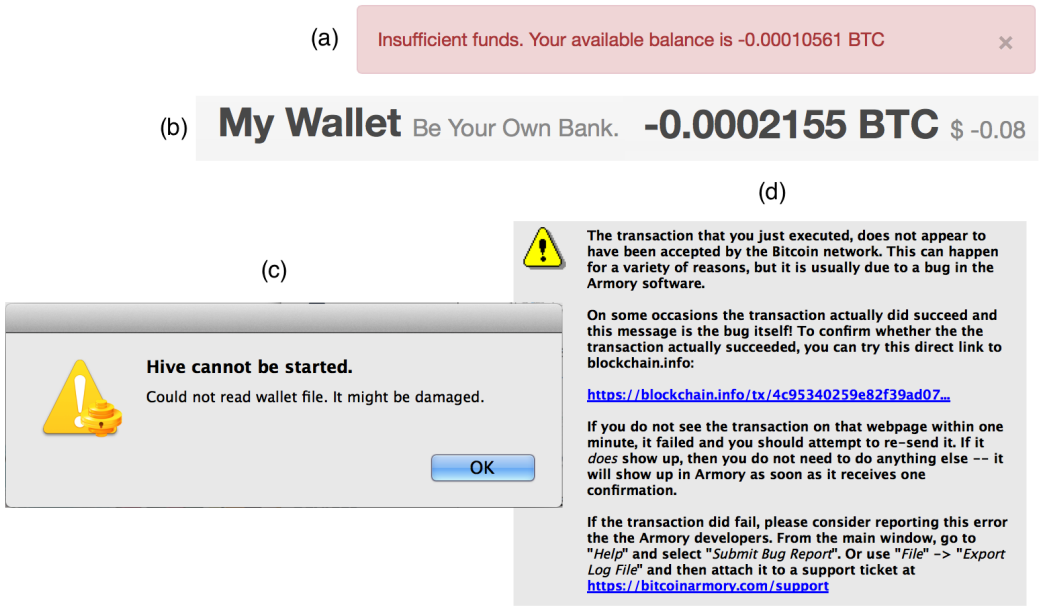


Fig. 2. Different behavior of clients during malleability attacks: (a) BitGo, (b) Blockchain.info., (c) Hive, and (d) Armory.

4.1 The deposit protocol

The Deposit protocol [19] is executed between parties A and B. To remain consistent with the rest of this paper we describe it here using the notation from [4,3] (cf. Section 2). The idea of this protocol is to allow A to create a financial deposit worth d ฿ for some period of time. A has to be sure that after time t she will get her money back and B has to be sure that A will not be able to claim her money earlier. One of the possible applications of this protocol is the scenario when B is a server and A is a user that wants to open an account on the server B. In this case B wants to be sure that A is a human and not a bot that creates the accounts automatically. To assure that, B forces A to create a small deposit for some time. For an honest user this should not be a big problem, because the deposit is small and she will get this money back. On the other hand it makes it expensive to create many fake accounts (for some malicious purposes), because the cumulative value of the deposits would grow huge.

We will now describe the deposit protocol in an informal way. The main idea of this protocol is fairly simple: A "deposits" her money using a transaction $Deposit$ that can be spent only using the signatures of both A and B. To be sure that this money will go back to her she creates a transaction $Fuse$ that spends $Deposit$. This transaction needs to be signed by B, and hence A asks B to sign it, and only after A receives this signature she posts $Deposit$ on the block chain. In order to prevent $\mathcal{A}$ from claiming her money too early $Fuse$ contains a timelock t. In more detail the protocol looks as follows:

1. At the beginning A and B exchange their public keys used for signing Bitcoin transactions and they agree on the deposit size d and time t at which the deposit will be freed.
2. Then A creates a transaction $Deposit$ of value d ฿, but she does not broadcast it. This transaction is constructed in such a way that to spend it someone has to include both signatures of A and B.

3. Afterwards A creates the transaction $Fuse$ that has a time lock t, spends the transaction $Deposit$ and sends the money back to her. This transaction is also not yet broadcast.
4. Then A sends the transaction $Fuse$ to B, he signs it and sends back to A.
5. Only then A sends the transaction $Deposit$ to the block chain.
6. After time t she sends the transaction $Fuse$ to get the deposit back.

The graph of transactions in the Simple deposit protocol is presented on Fig 3. The security proper-

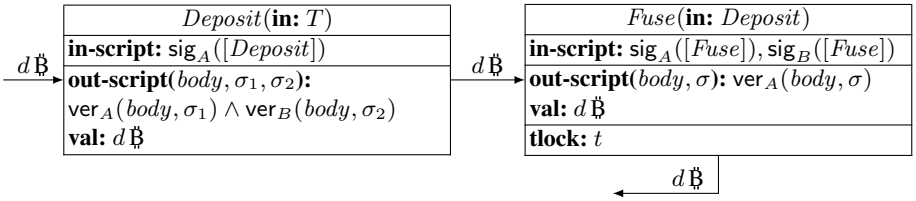


Fig. 3. The Deposit protocol (vulnerable to malleability) from [19].

ties that one would expect from this protocol are as follows:

(a) A is not able to get her deposit back before the time t (assuming that B follows the protocol).
(b) A will not lose her deposit, i.e. she will get d Ƀ back before the time $(t + \Delta)$ (where Δ denotes the maximum latency between broadcasting transaction and its confirmation).

It is easy to see that (a) holds, since there is no way $Deposit$ can be spent before time t (as it requires B's signature to be spent). One would be tempted to say that also (b) holds, since A can always claim her money back by posting $Fuse$ on the block chain. Unfortunately, it turns out that this argument strongly relies on the fact that $Deposit$ was posted on the block chain *exactly* in the same version as the one that was used to create the $Fuse$ transaction. Hence, if the adversary mauls $Deposit$ and posts some $Deposit'$ (syntactically different, but semantically equivalent to $Deposit$) then the $Fuse$ transaction will not be able to spend it (as it expects its input to have TXID equal to $H(Deposit)$, not $H(Deposit')$).

4.2 Other protocols vulnerable to the malleability attack

In this section we will list other known Bitcoin contracts that are vulnerable to the malleability attack. The problem with all of them is that they are creating a transaction spending another transaction before the latter is included in the block chain. Each of this protocols can be made resistant to malleability using our technique described in the next section.

- "Example 5: Trading across chains" from [19][12]
- "Example 7: Rapidly-adjusted (micro)payments to a pre-determined party" from [19][13]

[12] The malleability problem occurs in Step 3 when Party A generates Tx2 (the contract) which spends Tx1, and then asks B to sign it and send it back. This happens before Tx1 is included in the block chain and hence if later the attacker succeeds in posting a mauled version Tx1′ of Tx1 on the block chain, then the transaction Tx2 becomes invalid.

[13] The malleability problem is visible in Step 3, where the refund transaction T2 is created. This transaction depends on the transaction T1 that is not included in the block chain at the time when both parties sign it (in Step 3 and 4).

– Back and Bentov's lottery protocol [5][14]
– *Simultaneous Bitcoin-based timed commitment scheme* protocol from [3][15]

5 Our technique

In this section we will show how to fix the Deposit protocol to make it resistant to malleability. This technique can be used also to other protocols e.g. those listed in section 4.2. Recall that the reason why the protocols from Sections 4.1 and 4.2 were vulnerable to the malleability attacks was that one party, say A, had to obtain a signature of the other party (B) on a transaction T_1, whose input was a transaction T_0, and this had to happen *before* T_0 was posted on the block chain (in case of the Deposit protocol T_0 and T_1 were the $Deposit$ and the $Fuse$ transactions, resp.). Our main idea is based on the observation that, using the properties of the Bitcoin scripting language, we can modify this step by making T_0 spendable not by using the B's signature, but by providing a preimage s of some value h under a hash function H (where H can be, e.g., the SHA256 hash function available in the Bitcoin scripting language)[16]. In other words, the T_0's spending condition

$$\textbf{out-script}(body, \ldots, \sigma)\colon \cdots \wedge \mathsf{ver}_B(body, \sigma)$$

(cf. the Deposit protocol in Fig. 3) would be replaced by

$$\textbf{out-script}(body, \ldots, x)\colon \cdots \wedge H(x) = h,$$

where $h = H(s)$ is communicated by B to A, and s is chosen by B at random. This would allow A to spend T_0 no matter how it is mauled by the adversary, provided that A learns s. At the first sight this solution makes little sense, since there is no way in which B can be forced to send s to A (obviously in every protocol considered above s would need to be sent to A some time *after* T_0 appears on the block chain, as otherwise a malicious A could spend T_0 immediately). Fortunately, it turns out that this problem can be fixed by adding one more element to the protocol. Namely, we can use the *Bitcoin-based timed commitment scheme* from [4] which is a protocol that does exactly what we need here: it allows one party, called the *Committer* (in our case: B) to *commit* to a string s by sending $h = H(s)$ to the *Recipient* (here: A). Later, B can *open* the commitment by sending s to A (before this happens s is secret to $\mathcal{A}$). The special property of this commitment scheme is that the users can specify a time t until which B has to open the commitment. If he does not do it by this time, then he is forced to pay a fine (in bitcoins) to A. As shown in [4], the Bitcoin-based timed commitment scheme is secure even against the malleability attacks. For completeness we present this protocol in more detail in the next section.

5.1 Bitcoin-based timed commitment scheme

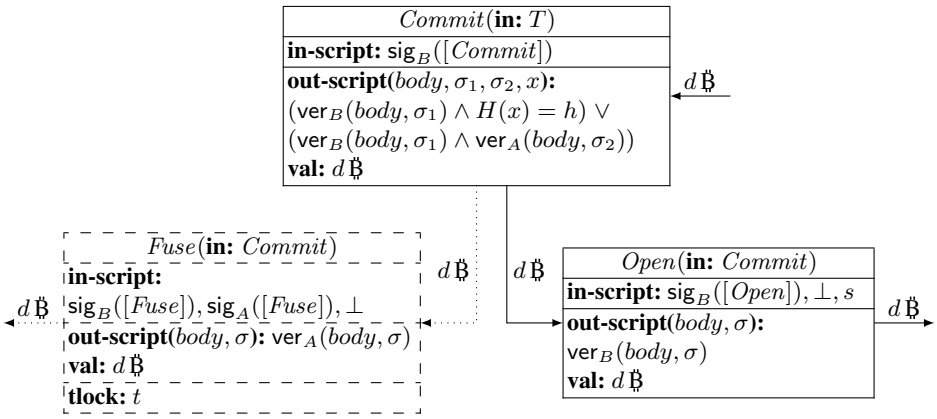
The Bitcoin-based timed commitment scheme protocol (CS) is being executed between the Committer B and the Recipient A. During the commitment phase the Committer commits himself to some

[14] The problem occurs in Steps 4 and 7, where the "refund_bet" and "refund_reveal" transactions are signed before theirs input transactions "bet" and "reveal" are broadcast.

[15] The problem is visible in Step 2 of the $Commit$ phase of this protocol (the $Fuse^{\mathsf{A}}$ and $Fuse^{\mathsf{B}}$ transactions are created before their input transaction $Commit$ appears on the block chain).

[16] Such transactions are called *hash locked* in the Bitcoin literature. Notice that having an output script, which requires only preimages and no signatures is not secure, because anyone who notices in the network a transaction trying to redeem such output script learns the preimages and can try to redeem this output script on his own. In our case the output script of the transaction T_0 requires also a signature of A, but we omit it (...) to simplify the exposition.

string s by revealing its hash $h = H(s)$. Moreover the parties agree on a moment of time t until which the Committer should open the commitment, i.e. reveal the secret value s. The protocol is constructed in such a way that if the Committer does not open the commitment until time t, then the agreed amount of d Ƀ is transferred from the Committer to the Recipient. More precisely, at the beginning of the protocol the Committer makes a deposit of d Ƀ, which is returned to him if he opens the commitment before time t or taken by the Recipient otherwise.



Pre-conditions:

1. The protocol is being executed between the Committer B holding the key pair B and the Recipient A holding the key pair A.
2. The Committer knows the secret string s.
3. The block chain contains an unredeemed transaction T with value d Ƀ, which can be redeemed with the key B.

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

1. The Committer computes $h = H(s)$ and broadcasts the transaction $Commit$. This obviously means that he reveals h, as it is a part of the transaction $Commit$.
2. The Committer waits until the transaction $Commit$ is confirmed. Then, he creates a body of the transaction $Fuse$, signs it and sends the signature to the Recipient.
3. If the Recipient does not receive the signature or the signature or transaction $Commit$ are incorrect then he quits the protocol.

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

4. Not later than at the time $(t - \Delta)$ the Committer broadcasts the transaction $Open$, what reveals the secret s.
5. If within time t the transaction $Open$ does not appear on the block chain then the Recipient broadcasts the transaction $Fuse$ to gain d Ƀ (or learn the secret s if the Committer sends the transaction $Open$ in the meantime and it is included in the block chain).

Fig. 4. The CS protocol from [4]. The scripts' arguments, which are omitted are denoted by $\perp$.

The graph of transactions and the full description of the CS protocol is presented on Fig. 4. The main idea is that if the Committer is honest then only the transactions $Commit$ and $Open$ will be used (to commit to s and to open s respectively). If, however, the Committer refuses to open his commitment, then the Receiver will post the $Fuse$ transaction on the block chain and claim B's deposit. Observe that $Fuse$ is time-locked and therefore a malicious receiver cannot claim the money before time t (and after time t he can do it only if B did not open the commitment). The reader may refer to [4] for more details. Note that even if the transaction $Commit$ is maliciously changed before being included in the block chain, the protocol still succeeds because the transaction $Fuse$ is created after $Commit$ is included in the block chain, so it always contains the correct hash of $Commit$. Therefore, the CS protocol is resistant to the transaction malleability. The properties of the CS protocol are as follows:

(a) The Recipient has no information about the secret s before the Committer broadcast the transaction $Fuse$ (this property is called *hiding*).
(b) The Committer cannot open his commitment in a different way than revealing his secret s (this property is called *binding*).
(c) The honest Committer will never lose his deposit, i.e. he will receive it back not later than at the time t.
(d) If the Committer does not reveal his secret s before the time $(t + \Delta)$ then the Recipient will receive d ฿ of compensation.

5.2 The details of our method

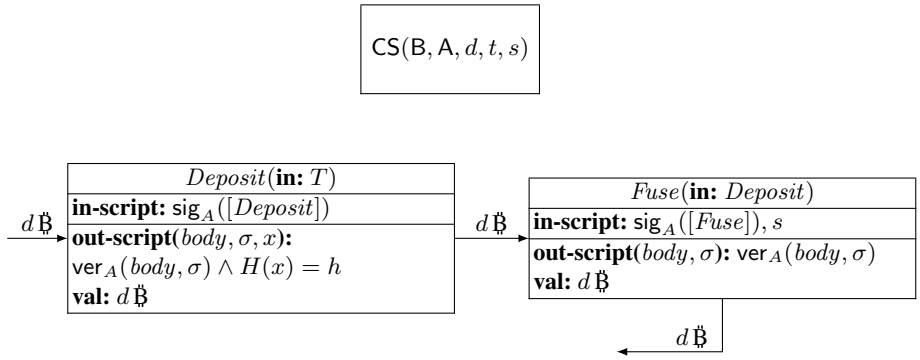
We now present in more detail our solution of the malleability problem in Bitcoin contracts that was already sketched at the beginning of Section 5. It can be used in all of the Bitcoin contracts that are vulnerable to the malleability attacks that we are aware of. In this paper we show how to apply it to the Deposit protocol (described in Section 4.1).

The main idea of our solution is to use the CS protocol instead of standard $Fuse$ transaction. More precisely at the beginning of the protocol B samples a random secret s and commits himself to it using the $Commit$ phase of the CS protocol. Now A knows that B will have to reveal his secret (i.e. the value s s.t. $H(s) = h$) before the time t. So A can create a $Deposit$ transaction in such a way that to spend it she has to provide her signature and the value s. That means that after the time t either B will reveal the value s and A will be able to spend the transaction $Deposit$ or A will get the deposit of B from the CS protocol. Such a modified protocol is denoted NewDeposit. Its graph of transactions and its full description is presented on Fig. 5.

The properties of the NewDeposit protocol are as follows (all of them hold even against the malleability attacks):

(a) A is not able to get her deposit back before the time $(t - \Delta)$.
(b) The honest A will not lose her deposit, i.e. she will get d ฿ back before the time $(t + 2\Delta)$.
(c) Additionally the honest B will not lose his deposit, i.e. he will get it back before the time t.

The proof of the above properties is straightforward and it is omitted because of the lack of space. We note that this protocol may not be well-suited for the practical applications, as it requires the server to make a deposit. Nevertheless, it is a very good illustration of our technique, that is generic and has several other applications.



Pre-conditions:

1. A holds the key pair A and B holds the key pair B.

The commitment phase:

1. B samples a random secret s.
2. The parties execute the $\mathsf{CS.Commit}(\mathsf{B}, \mathsf{A}, d, t, s)$ protocol.
3. A creates the $Deposit$ transaction using the value h from the transaction $\mathsf{CS}(\mathsf{B}, \mathsf{A}, d, t, s).Commit$ and broadcasts it.

The opening phase:

4. B opens the commitment revealing s at the time $(t - \Delta)$.
5. A learns the value of s and broadcasts the $Fuse$ transaction.
6. If B has not opened the commitment until time t then A broadcasts the transaction $\mathsf{CS}(\mathsf{B}, \mathsf{A}, d, t, s).Fuse$ to gain d ₿ (or learn the value s if the Committer sends the transaction $Open$ in the meantime and it is included in the block chain; in that case A may just broadcast the $Fuse$ transaction to get her deposit back).

Fig. 5. The solution of the deposit problem resistant to malleability. $\mathsf{CS}(\mathsf{B}, \mathsf{A}, d, t, r)$ denotes the transactions in the appropriate execution of the CS protocol.

References

1. bips/bip-0065.mediawiki. http://github.com/bitcoin/bips/blob/master/bip-0065.mediawiki, accessed od 10.12.2014.
2. bitcoinj library homepage. http://bitcoinj.github.io, accessed on 20.10.2014.
3. Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Łukasz Mazurek. Fair two-party computations via bitcoin deposits. In Rainer Böhme, Michael Brenner, Tyler Moore, and Matthew Smith, editors, *Financial Cryptography and Data Security*, Lecture Notes in Computer Science, pages 105–121. Springer Berlin Heidelberg, 2014.
4. Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Łukasz Mazurek. Secure multiparty computations on bitcoin. In *Security and Privacy (SP), 2014 IEEE Symposium on Security and Privacy*, May 2014.
5. Adam Back and Iddo Bentov. Note on fair coin toss via bitcoin, 2013. http://www.cs.technion.ac.il/%7Eidddo/cointossBitcoin.pdf.
6. Iddo Bentov and Ranjit Kumaresan. How to use bitcoin to design fair protocols. Lecture Notes in Computer Science, pages 421–439. Springer, August 2014.
7. Iddo Bentov and Ranjit Kumaresan. How to use Bitcoin to design fair protocols. Cryptology ePrint Archive, Report 2014/129, 2014. eprint.iacr.org/2014/129, accepted to ACM CCS'14.

8. Bitcoin.org. Developer reference. http://bitcoin.org/en/developer-reference, accessed on 20.10.2014.
9. Bitcoin.org. List of bitcoin wallets. http://bitcoin.org/en/choose-your-wallet, accessed on 20.10.2014.
10. Alexandra Boldyreva, David Cash, Marc Fischlin, and Bogdan Warinschi. Foundations of non-malleable hash and one-way functions. In Mitsuru Matsui, editor, *Advances in Cryptology – ASIACRYPT 2009*, volume 5912 of *Lecture Notes in Computer Science*, pages 524–541. Springer, December 2009.
11. Vitalik Buterin. Bitcoin network shaken by blockchain fork, March 2013. Bitcoin Magazine, available at http://bitcoinmagazine.com/3668/bitcoin-network-shaken-by-blockchain-fork,.
12. Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. Universally composable two-party and multi-party secure computation. In *34th Annual ACM Symposium on Theory of Computing*, pages 494–503. ACM Press, May 2002.
13. Christian Decker and Roger Wattenhofer. Bitcoin transaction malleability and mtgox. In Miroslaw Kutylowski and Jaideep Vaidya, editors, *Computer Security - ESORICS 2014*, volume 8713 of *Lecture Notes in Computer Science*, pages 313–326. Springer International Publishing, 2014.
14. Yevgeniy Dodis and Daniel Wichs. Non-malleable extractors and symmetric key cryptography from weak secrets. In Michael Mitzenmacher, editor, *41st Annual ACM Symposium on Theory of Computing*, pages 601–610. ACM Press, May / June 2009.
15. Danny Dolev, Cynthia Dwork, and Moni Naor. Nonmalleable cryptography. *SIAM Journal on Computing*, 30(2):391–437, 2000.
16. Stefan Dziembowski, Tomasz Kazana, and Maciej Obremski. Non-malleable codes from two-source extractors. Lecture Notes in Computer Science, pages 239–257. Springer, August 2013.
17. S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.
18. Joe Weisenthal. Bitcoin just completely crashed as major exchange says withdrawals remain halted, Feb. 2014. Business Insider, available at www.businessinsider.com/mtgox-statement-on-withdrawals-2014-2.
19. Bitcoin Wiki. Contracts. http://en.bitcoin.it/wiki/Contracts, accessed on 20.10.2014.
20. Bitcoin Wiki. Main page. http://en.bitcoin.it, accessed on 20.10.2014.
21. Bitcoin Wiki. Transaction malleability. http://en.bitcoin.it/wiki/Transaction_Malleability, accessed on 20.10.2014.
22. Pieter Wuille. Bitcoin improvement proposal: Dealing with malleability. http://github.com/bitcoin/bips/blob/master/bip-0062.mediawiki, accessed on 20.10.2014.

Secure Multiparty Computations on Bitcoin*

Marcin Andrychowicz
University of Warsaw
marcin.andrychowicz
@crypto.edu.pl

Stefan Dziembowski
University of Warsaw
stefan.dziembowski
@crypto.edu.pl

Daniel Malinowski
University of Warsaw
daniel.malinowski
@crypto.edu.pl

Łukasz Mazurek
University of Warsaw
lukasz.mazurek
@crypto.edu.pl

ABSTRACT

Is it possible to design an online protocol for playing a lottery, in a completely decentralized way, i.e. 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 Satoshi Nakamoto[1] [17]. 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, i.e.: can we implement some more advanced financial instruments in a distributed manner? The Bitcoin specification partly answers this question, by providing the so-called "non-standard 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 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 in order 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 $f_{\mathrm{rnd}} : \{\perp\} \times \{\perp\} \to \{0, 1\}$ that takes no inputs and outputs 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

*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.

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

Bob, and simultaneously Bob sends a bit b_B to Alice. They output b is computed as $b := b_A \oplus b_B$ (where "$\oplus$" denotes the xor 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 [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 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 these 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 e.g. $b = 0$ then Bob after learning the output of the function has no information about Alice's input. Therefore the privacy of Alice is protected.

On 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 email 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 email 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 *multi*party computations, 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.

Secondly, one can consider more complicated functions than the ones described above. It was shown in [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 [16, 4, 10] and sometimes even to use them in the real-life applications such as, e.g., 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 $b = 1$ 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 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 "multiparty computation 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 outcome is respected 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 are Bitcoin transaction fees. Most 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 2), for which there is usually some small fee (currently around 0.0001 ฿ $\approx$ \$ 0.04).[2] 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

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

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[3].

1.2 Independent and subsequent work

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

In subsequent work [1, 2] we show how to extend the ideas from this paper in order 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 non-malleable (see [1, 2] for more on this notion), and in [2] we show how to modify the protocol from [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 Iddo Bentov and Ranjit 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, the 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 are 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.

[3]For example the main transaction (*Compute*) of the three-party lottery is available here: blockchain.info/tx/540d816bd57300209754dd36ffcec1d669-bd2068641844783451cd3ef32c8aa4.

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

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 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 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 minutes. 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 non-trivial 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 to 20 minutes 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. 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 a 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.

2.1.1 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 v (called the *value* of a transaction) is transferred from an address $A.pk$ to an address $B.pk$ has the form $T_x = (y, v, B.pk, \mathsf{sig})$, 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 $(y, v, B.pk)$). 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 v 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 v, (3) the transaction T_y has not been redeemed earlier, and (4) the signature of A is correct. All of 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 $T_x = (y, v, B.pk, \mathsf{sig})$, which transfers v bitcoins from A to B, is depicted in Fig. 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 $T_{y_1}, \ldots, T_{y_\ell}$. 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: $T_x = (y_1, \ldots, y_\ell, v, B.pk, \mathsf{sig}_1, \ldots, \mathsf{sig}_\ell)$, 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 v, provided it is equal to the sum of the values of the 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. 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.

2.1.2 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 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 transaction becomes valid. In this case we have: $T_x = (y_1, \ldots, y_\ell, v, B.pk, t, \mathsf{sig}_1, \ldots, \mathsf{sig}_\ell)$. 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: $T_x = (y, \pi_x, v, \sigma_x)$, where $[T_x] = (y, \pi_x, v)$ 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 $T_x = (y_1, \ldots, y_\ell, \pi_x, v, t, \sigma_1, \ldots, \sigma_\ell)$, 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}, (3) none of these transactions has been redeemed before, and (4) the sum of values of transactions T_{y_i} is equal to v.

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

The most common type of transactions are 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.

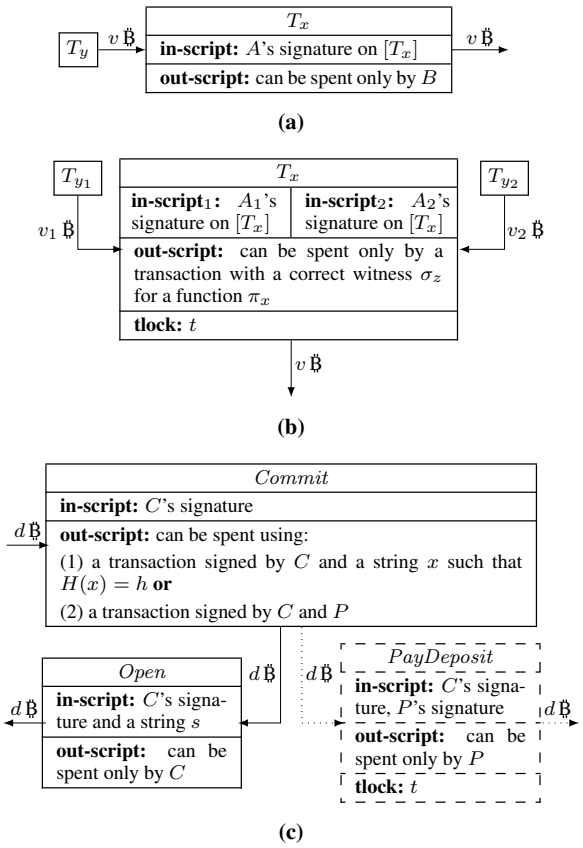


Figure 1: (a) a standard transaction transferring v bitcoins from A to B, (b) a non-standard transaction with two inputs and a time-lock, (c) the CS protocol.

Currently some miners accept only standard transactions (although the non-standard transactions are also correct according to the Bitcoin description). We believe that in the future accepting the non-standard transactions will become common. This is important for our applications since our protocols rely heavily on non-standard transactions.

3. BITCOIN-BASED TIMED COMMITMENT SCHEME

We start with constructing a Bitcoin-based timed commitment scheme. Commitment schemes were already described in the introduction. 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 $Open$, that reveals the secret s. The transaction $Commit$ is constructed in such a way that the $Open$ transaction has to open the commitment, i.e., 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 the introduction, 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 the introduction. 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 $P \in \{\mathsf{A}, \mathsf{B}\}$ chooses a string s_P sampled uniformly random from $\{0,1\}^{128+b_P}$, i.e., 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: $f(s_A, s_B) = \mathsf{A}$ 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).

4.1.1 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[4]. 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 $P \in \{\mathsf{A}, \mathsf{B}\}$ realizes that the other party is cheating, then the first thing P will do is to "take the money and run", i.e., 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. $X = f(x_A, x_B)$)	

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 pre-images 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.

4.1.2 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

[4]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 $f(s_A, s_B) = A$.

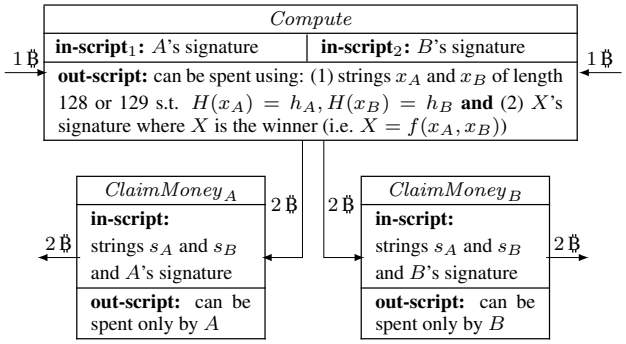


Figure 2: The SecureLottery **protocol.**

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 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 4 "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.

5. REFERENCES

[1] M. Andrychowicz, S. Dziembowski, D. Malinowski, and Ł. Mazurek. Fair two-party computations via bitcoin deposits. In *1st Workshop on Bitcoin Research*, 2014.

[2] M. Andrychowicz, S. Dziembowski, D. Malinowski, and Ł. Mazurek. On the malleability of bitcoin transactions. In *2st Workshop on Bitcoin Research*, 2014.

[3] A. Back and I. Bentov. Note on fair coin toss via bitcoin, 2013. `http://www.cs.technion.ac.il/~idddo/cointossBitcoin.pdf`.

[4] A. Ben-David, N. Nisan, and B. Pinkas. FairplayMP: a system for secure multi-party computation. In *ACM CCS 08: 15th Conference on Computer and Communications Security*, 2008.

[5] I. Bentov and R. Kumaresan. How to use bitcoin to design fair protocols. In *Advances in Cryptology - CRYPTO*, 2014.

[6] M. Blum. Coin flipping by telephone. In *Advances in Cryptology – CRYPTO'81*.

[7] P. Bogetoft et al. Secure multiparty computation goes live. In *FC 2009: 13th International Conference on Financial Cryptography and Data Security*.

[8] D. Boneh and M. Naor. Timed commitments. In *Advances in Cryptology – CRYPTO 2000*.

[9] R. Cleve. 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.

[10] I. Damgård 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*.

[11] J. R. Douceur. The sybil attack. In *First International Workshop on Peer-to-Peer Systems*, IPTPS '01, 2002.

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

[13] J. A. Garay and M. Jakobsson. Timed release of standard digital signatures. In *FC 2002: 6th International Conference on Financial Cryptography*.

[14] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In *19th Annual ACM Symposium on Theory of Computing*.

[15] R. Kumaresan and I. Bentov. How to use bitcoin to incentivize correct computations. In *ACM CCS 2014*, pages 30–41, 2014.

[16] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay - a secure two-party computation system. In *13th Conference on USENIX Security Symposium*, SSYM'04.

[17] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[18] T. W. Post. Cheating scandals raise new questions about honesty, security of internet gambling, November 30, 2008.

[19] P. Resnick, K. Kuwabara, R. Zeckhauser, and E. Friedman. Reputation systems. *Commun. ACM*, (12), Dec. 2000.

[20] A. C.-C. Yao. How to generate and exchange secrets (extended abstract). In *27th Annual Symposium on Foundations of Computer Science*.

PoW-Based Distributed Cryptography with no Trusted Setup

Marcin Andrychowicz* and Stefan Dziembowski**

University of Warsaw

Abstract. Motivated by the recent success of Bitcoin we study the question of constructing distributed cryptographic protocols in a fully peer-to-peer scenario under the assumption that the adversary has limited computing power and there is *no* trusted setup (like PKI, or an unpredictable beacon). We propose a formal model for this scenario and then we construct a broadcast protocol in it. This protocol is secure under the assumption that the honest parties have computing power that is some non-negligible fraction of computing power of the adversary (this fraction can be small, in particular it can be much less than $1/2$), and a (rough) total bound on the computing power in the system is known.
Using our broadcast protocol we construct a protocol for simulating any trusted functionality. A simple application of the broadcast protocol is also a scheme for generating an unpredictable beacon (that can later serve, e.g., as a genesis block for a new cryptocurrency).
Under a stronger assumption that the majority of computing power is controlled by the honest parties we construct a protocol for simulating any trusted functionality with guaranteed termination (i.e. that cannot be interrupted by the adversary). This could in principle be used as a provably-secure substitute of the blockchain technology used in the cryptocurrencies.
Our main tool for verifying the computing power of the parties are the Proofs of Work (Dwork and Naor, CRYPTO 92). Our broadcast protocol is built on top of the classical protocol of Dolev and Strong (SIAM J. on Comp. 1983).

1 Introduction

Distributed cryptography is a term that refers to cryptographic protocols executed by a number of mutually distrusting parties in order to achieve a common goal. One of the first primitives constructed in this area were the *broadcast protocols* [38,20] using which a party P can send a message over a point-to-point network in such a way that all the other parties will reach *consensus* about the value that was sent (even if P is malicious). Another standard example are the secure multiparty computations (MPCs) [48,30,13,8], where the goal of the parties is to simulate a trusted functionality. The MPCs turned out to be a very exciting theoretical topic. They have also found some applications in practice (in particular they are used to perform the secure on-line auctions [10]). Despite of this, the MPCs unfortunately still remain out of scope of interest

* m.andrychowicz@mimuw.edu.pl
** s.dziembowski@mimuw.edu.pl

for most of the security practitioners, who are generally more focused on more basic cryptographic tools such as encryption, authentication or the digital signature schemes.

One of very few examples of distributed cryptography techniques that attracted attention from general public are the *cryptographic currencies* (also dubbed the *cryptocurrencies*), a fascinating recent concept whose popularity exploded in the past 1-2 years. Historically the first, and the most prominent of them is the *Bitcoin*, introduced in 2008 by an anonymous developer using a pseudonym "Satoshi Nakamoto" [43]. Bitcoin works as a peer-to-peer network in which the participants jointly emulate the central server that controls the correctness of transactions, in particular: it ensures that there was no "double spending", i.e., a given coin was not spent twice by the same party. Although the idea of multiple users jointly "emulating a digital currency" sounds like a special case of the MPCs, the creators of Bitcoin did not directly use the tools developed in this area, and it is not clear even to which extend they were familiar with this literature (in particular, Nakamoto [43] did not cite any of MPC papers in his work). Nevertheless, at the first sight, there are some resemblances between these areas. In particular: the Bitcoin system works under the assumption that the majority of computing power in the system is under control of the honest users, while the classical results from the MPC literature state that in general constructing MPC protocols is possible when the majority of the users is honest.

At a closer look, however, it becomes clear that there are some important differences between both areas. In particular the main reason why the MPCs cannot be used directly to construct the cryptocurrencies is that the scenarios in which these protocols are used are fundamentally different. The MPCs are supposed to be executed by a fixed (and known in advance) set of parties, out of which some may be honestly following the protocol, and some other ones may be corrupt (i.e. controlled by the adversary). In the most standard case the number of misbehaving parties is bounded by some threshold parameter t. This can be generalized in several ways. Up to our knowledge, however, until now all these generalizations use a notion of a "party" as a separate and well-defined entity that is either corrupt or honest.

The model for the cryptocurrencies is very different, as they are supposed to work in a purely peer-to-peer environment, and hence the notion of a "party" becomes less clear. This is because they are constructed with a minimal trusted setup (as we explain below the only "trusted setup" in Bitcoin was the generation of an unpredictable "genesis block"), and in particular they do not rely on any Public Key Infrastructure (PKI), or any type of a trusted authority that would, e.g., "register" the users. Therefore the adversary can always launch a so-called *Sybil attack* [21] by creating a large number k of "virtual" parties that remain under his control. In this way, even if in reality he is just a single entity, from the point of view of the other participants he will control a large number of parties. In some sense the cryptocurrencies lift the "lack of trust" assumption to a whole new level, by considering the situation when it is not even clear who is a "party". The Bitcoin system overcomes this problem in the following way: the honest majority is defined in terms of the "majority of computing power". This is achieved by having all the honest participants to constantly prove that they devote certain computing power to the system, via the so-called "Proofs of Work" (PoWs) [22,23].

The high level goal for this work is to bridge the gap between these two areas. In particular, we propose a formal model for the peer-to-peer communication and the Proofs of Work concept used in Bitcoin. We also show how some standard primitives from the distributed computation, like broadcast and MPCs, can be implemented in this model. Our protocols do not require any trusted setup assumptions, unlike Bitcoin that assumes a trusted generation of an unpredictable "genesis block" (see below for more details). Besides of being of general interest, our work is motivated twofold.

Firstly, recently discovered weaknesses of Bitcoin [26,6] come, in our opinion, partially from the lack of a formal framework for this system. Our work can be viewed as a step towards better understanding of this model. We also believe that the "PoW-based distributed cryptography" can find several other applications in the peer-to-peer networks (we describe some of them). In particular, as the Bitcoin example shows, the "lack of trusted setup" can be very attractive to users[1]. In fact, there are already some ongoing efforts to use the Bitcoin paradigm for purposes other than the cryptocurrencies (see Appendix A.3 for more on this). We would like to stress however, that this is not the main purpose of our work, and that we do not provide a full description of a new currency. Our goal is also not the full analysis of the security of Bitcoin (which would be a very ambitious project that would also need to take into account the economical incentives of the participants).

Secondly, what may be considered unsatisfactory in Bitcoin is the fact that its security relies on the fact that the so-called *genesis block* B_0, announced by Satoshi Nakamoto on January 3, 2009, was generated using heuristic methods. More concretely, in order to prove that he did not know B_0 earlier, he included the text *The Times 03/Jan/2009 Chancellor on brink of second bailout for banks* in B_0 (taken from the front page of the London Times on that day). The *unpredictability* of B_0 is important for Bitcoin to work properly, as otherwise a "malicious Satoshi Nakamoto" $\mathcal{A}$ that knew B_0 beforehand could start the mining process much earlier, and publish an alternative block chain at some later point. Since he would have more time to work on his chain, it would be longer than the "official" chain, even if $\mathcal{A}$ controls only a small fraction of the total computing power. Admittedly, its now practically certain that no attack like this was performed, and that B_0 was generated honestly, as it is highly unlikely that any $\mathcal{A}$ invested more computing power in Bitcoin mining than all the other miners combined, even if $\mathcal{A}$ started the mining process long before January 3, 2009.

However, if we want to use the Bitcoin paradigm for some other purpose (including starting a new currency), it may be desirable to have an automatic and non-heuristic method of generating unpredictable strings of bits. The problem of generating such *random beacons* [44] has been studied in the literature for a long time. Informally: a random beacon scheme is a method (possibly involving a trusted party) of generating uniformly random (or indistinguishable from random) strings that are unknown before the moment of their generation. The beacons have found a number of applications in cryptography and information security, including the secure contract signing protocols [44,25], voting schemes [42], or zero-knowledge protocols [4,31]. Note that a random

[1] Actually, probably one of the reasons why the MPCs are not widely used in practice is that the typical users do not see a fundamental difference between assuming a trusted setup and delegating the whole computation to a trusted third party.

beacon is a stronger concept than the *common reference string* frequently used in cryptography, as it has to be unpredictable before it was generated (for every instance of the protocol using it). Notice also that for Bitcoin we actually need something weaker than uniformity of the B_0, namely it is enough that B_0 is hard to predict for the adversary.

Constructing random beacons is generally hard. Known practical solutions are usually based on a trusted third party (like the servers www.random.org and beacon.nist.gov). Since we do not want to base the security of our protocols on trusted third parties thus using such services is not an option for our applications. Another method is to use public data available on the Internet, e.g. the financial data [14] (the Bitcoin genesis block generation can also be viewed as an example of this method). Using publicly-available data makes more sense, but also this reduces the overall security of the constructed system. For example, in any automated solution the financial data would need to come from a trusted third party that would need to certify that the data was correct. The same problem applies to most of other data of this type (like using a sentence from a newspaper article). One could also consider using the Bitcoin blocks as such beacons (in fact recently some on-line lotteries started using them for this purpose). We discuss the problems with this approach in App. A.4.

Our contribution. Motivated by the cryptocurrencies we initiate a formal study of the distributed peer-to-peer cryptography based on the Proofs of Work. From the theory perspective the first most natural questions in this field is what is the right model for communication and computation in this scenario? And then, is it possible to construct in this model some basic primitives from the distributed cryptography area, like: (a) broadcast, (b) unpredictable beacon generation, or (c) general secure multiparty computations? We propose such a model (in Section 2). Our model does not assume any trusted setup (in particular: we do not assume any trusted beacon generation). Then, in Section 4 we answer the questions (a)-(c) positively. To describe our results in more detail let n denote the number of honest parties, let π be the computing power of each honest party (for simplicity we assume that all the honest parties have the same computing power), let π_{max} be the maximal computing power of all the participants of the protocol (the honest parties and the adversary), and let $\pi_{\mathcal{A}} \leq \pi_{\mathsf{max}} - n\pi$ be the actual computing power of the adversary. We allow the adversary to adaptively corrupt at most t parties, in which case he takes the full control over them (however, we do not allow him to use the computing power of the corrupt parties, or in other words: once he corrupts a party he is also responsible for computing the Proofs of Work for her). Of course in general it is better to have protocols depending on $\pi_{\mathcal{A}}$, not on π_{max}. On the other hand, sometimes the dependence from π_{max} is unavoidable, as the participants need to have some rough estimate on the power of the adversary (e.g. clearly it is hard to construct any protocol when π is negligible compared to π_{max}). Note that also Bitcoin started with some arbitrary assumption on the computing power of the participant (this was reflected by setting the initial "mining difficulty" to 2^{32} hash computations). Our contribution is as follows. First, we construct a broadcast protocol secure against any π_{max}, working in time linear in $\lceil \pi_{\mathsf{max}}/\pi \rceil$. Then, using this broadcast protocol, we argue how to construct a protocol for executing any functionality in our model. In case

the adversary controls the minority of the computing power (i.e. $n \geq \lceil \pi_{\mathcal{A}}/\pi \rceil + t$)[2] that were user ber our protocol cannot be aborted prematurely by her. This could in principle be used as a provably-secure substitute of the blockchain technology used in the cryptocurrencies. Using the broadcast protocol as a subroutine we later (in Section 5) construct a scheme for an unpredictable beacon generation.

One thing that needs to be stressed is that our protocols do not require an unpredictable trusted beacon to be executed (and actually, as described above, constructing a protocol that emulates such a beacon is one of our contributions). This poses a big technical challenge, since we have to prevent the adversary from launching a "pre-computation" attack, i.e., computing solutions to some puzzles before the execution of the protocol started.

The only thing that we assume is that the participating parties know a session identifier (sid), which can be known publicly long time before the protocol starts. Observe that some sort of mechanism of this type is always needed, as the parties need to know in advance, e.g., the time when the execution starts.

One technical problem that we need to address is that, since we work in a purely peer-to-peer model, an adversary can always launch a Denial of Service Attack, by "flooding" the honest parties with his messages, hence forcing them to work forever. Thus, in order for the protocols to terminate in a finite time we also need some mild upper bound θ on the number of messages that the adversary can send (much greater than what the honest parties will send). We write more on this in Section 2. Although our motivation is mostly theoretic, we believe that our ideas can lead to practical implementations (probably after some optimizations and simplifications). We discuss some possible applications of our protocols in Section 5.

Independent work. Recently an interesting paper by Katz, Miller and Shi [36] with a motivation similar to ours was published on the Eprint archive. While their high-level goal is similar to ours, there are some important technical differences. First of all, their solution essentially assumes existence of a trusted unpredictable beacon (technically: they assume that the parties have access to a random oracle that was not available to the adversary before the execution started). This simplifies the design of the protocols significantly, as it removes the need for every party to ensure that "her" challenge was used to compute the Proof-of-Work (that in our work we need to address to deal with the pre-computation attacks described above). Secondly, they assume that the proof verification takes zero time (we note that with such an assumption our protocols would be significantly simpler, and in particular we would not need an additional paramter θ that measures the number of messages sent by the adversary). Thirdly, unlike us, they assume that the number of parties executing the protocol is known from the beginning. Finally, they use strong intractability assumptions taken from the "time-lock puzzles" literature, namely that computing some hash functions has intrinsic "sequential cost".

[2] The reader migh be confused we in this inequality t appears on the righ hand side, as it may look like contradicting the assumption that the adversary does not take the control of the computing power of the corrupt parties. The reason for having this term is the adaptivity: the adversary can corrupt a party at the very end of the protocol, hence, in some sense taking advantage of her computing resources before she was corrupted.

2 Our model

In this section we present our model for reasoning about computing power and the peer-to-peer protocols. We first do it informally, and then formalize it using the *universal composability framework* of Canetti [11].

Modeling hashrate Since in general proving lower bounds on the computational hardness is very difficult, we make some simplifying assumptions about our model. In particular, following a long line of previous works both in theory and in the systems community (see e.g. [23,43,5]), we establish the lower bounds on computational difficulty by counting the number of times a given algorithm calls some random oracle H [7]. In our protocols the size of the input of H will be linear in the security parameter κ (usually it will be 2κ at most). Hence it is realistic to assume that each invocation of such a function takes some fixed unit of time.

Our protocols are executed in real time by a number of devices and attacked by an adversary $\mathcal{A}$. The exact way in which time is measured is not important, but it is useful to fix a unit of time Δ (think of it as 1 minute, say). Each device D that participates in our protocols will be able to perform some fixed number π_D of queries to H in time Δ. The parameter π_D is called the *hashrate of D (per time Δ)*. The hashrate of the adversary is denoted by $\pi_{\mathcal{A}}$. The other steps of the algorithms do not count as far as the hashrate is considered (they will count, however, when we measure the efficiency of our protocols, see paragraph *Computational complexity* below). Moreover we assume that the parties have access to a "cheap" random oracle, calls to this oracle do not count as far as the hashrate is considered. This assumption is made to keep the model as simple as possible. It should be straightforward that in our protocols we do not abuse this assumption, and in on any reasonable architecture the time needed for computing H's would be the dominating factor during the Proofs of Work. In particular: any other random oracles will be invoked a much smaller number of times than H. Note that, even if these numbers were comparable, one could still make H evaluate much longer than any other hash function F, e.g., by defining H to be equal to multiple iterations of F.

In this paper we will assume that every party (except of the adversary) has the same hashrate per time Δ (denoted π). This is done only to make the exposition simpler. Our protocols easily generalize to the case when each party has a device with hashrate π_i and the π_i's are distinct. Note that if a party has a hashrate $t\pi$ (for natural t) then we can as well think about her as of t parties of hashrate π each. Making it formal would require changing the definition of the "honest majority" in the MPCs to include also "weights" of the parties.

The communication model. Unlike in the traditional MPC settings, in our case the number of parties executing the protocol is not known in advance to the parties executing it. Because of this it makes no sense to specify a protocol by a finite sequence $(M_1, \ldots, M_n)$ of Turing machines. Instead, we will simply assume that there is *one* Turing machine M whose code will be executed by each party participating in the protocol (think of it as many independent executions of the same program). This, of course,

does no mean that these parties have identical behavior, since their actions depend also on their inputs, the party identifier (pid), and the random coins.

Since we do not assume any trusted set-up (like a PKI or shared private keys) modeling the communication between the parties is a bit tricky. We assume that the parties have access to a public channel which allows every party and the adversary to post a message on it. One can think of this channel as being implemented using some standard (cryptographically insecure) "network broadcast protocol" like the one in Bitcoin [47]. The contents of the communication channel is publicly available. The message m sent in time t by some P_i is guaranteed to arrive to P_j within time t' such that $t' - t \leq \Delta$. Note that some assumption of this type needs to be made, as if the messages can be delayed arbitrarily then there is little hope to measure the hashrate reliably. Also observe that we have to assume that the messages always reach their destinations, as otherwise an honest party could be "cut of" the network. Similar assumptions are made (implicitly) in Bitcoin. Obviously without assumptions like this, Bitcoin would be easy to attack (e.g. if the miners cannot send messages to each other reliably then it is easy to make a "fork" in the blockchain).

To keep the model simple we will assume that the parties have perfectly synchronized clocks. This assumption could be easily relaxed by assuming that clocks can differ by a small amount of time δ, and our model from Section 2.1 could be easily extended to cover also this case, using the techniques, e.g., from [35]. We decided not to do it to in order to keep the exposition as simple as possible.

We give to the adversary full access to the communication between the parties: he learns (without any delay) every message that is sent through the communication channel, and he can insert messages into it. The adversary may decide that the messages inserted into the channel by him arrive only to a certain subset of the parties (he also has a full control over the timing when they arrive). The only restriction is that he cannot erase or modify the messages that were sent by the other parties (but he can delay them for time at most Δ).

Resistance to the denial of service attacks As already mentioned in the introduction, in general a complete prevention of the denial of service attacks against fully distributed peer-to-peer protocols seems very hard. Since we do not assume any trusted set-up phase, hence from the theoretical point of view the adversary is indistinguishable from the honest users, and hence he can always initiate a connection with an honest user forcing it to perform some work. Even if this work can be done very efficiently, it still costs some effort (e.g. it requires the user to verify a PoW solution), and hence it allows a powerful (yet poly-time bounded) adversary to force each party to work for a very long amount of time, and in particular to exceed some given deadline for communicating with the other parties. Since any PoW-based protocol inherently needs to have such deadlines, thus we need to somehow restrict the power of adversary. We do it in the following way.

First of all, we assume that if a message m sent to P_i is longer than the protocols specifies then P_i can discard it without processing it.[3] Secondly, we assume that there

[3] Discarding incorrect messages is actually a standard assumption in the distributed cryptography. Here we want to state it explicitly to make it clear that the processing time of too long messages does not count into the computing steps of the users.

is a total bound θ on the number of messages that all the participants can send during each interval Δ. Since this includes also the messages sent by the honest parties, thus the bound on the number of messages that the adversary $\mathcal{A}$ sends will be slightly more restrictive, but from practical point of view (since the honest parties send very few messages) it is approximately equal to θ. This bound can be very generous, and, moreover it will be much larger than the number of messages sent by the honest users[4]. In practice such a bound could be enforced using some ad-hoc methods. For example each party could limit the number of messages it can receive from a given IP address. Although from the theoretical perspective no heuristic method is fully satisfactory, in practice they seem to work. For example Bitcoin seems to resist pretty well the DoS attacks thanks to over 30 ad-hoc methods of mitigating them (see [46]). Hence, we believe that some bound on θ is reasonable to assume (and, as argued above, seems necessary). We will use this bound in a weak way, in particular the number of messages sent by the honest parties will not depend on it, and the communication complexity will (for any practical choice of parameters) be linear in θ for every party (in other words: by sending θ messages the adversary can force an honest party to send one long message of length $O(\theta)$). The real time of the execution of the protocol can depend on θ. Formally it is a linear dependence (again: this seems to be unavoidable, since every message that is sent to an honest party P_i forces P_i to do some non-trivial work). Fortunately, the constant of this linear function will be really small. For example, in the $\mathsf{RankedKeys}$ (Figure 1, Page 16) the time each round takes (in the "key ranking phase") will be $\Delta + \theta \cdot \mathsf{time}_{\mathsf{V}}/\pi$, where $\mathsf{time}_{\mathsf{V}}$ is small. Observe that, e.q, $\theta/\pi = 1$ if the adversary can send the messages at the same speed as the honest party can compute the $\mathcal{H}^{\kappa}$ queries, hence it is reasonable to assume that $\theta/\pi < 1$.

Communication, message and computational complexity We define the communication complexity in the public channel model in App. E. We will also analyze the computational complexity of our protocols. This is defined in a straightforward way and appears in App. F. We can also extend our model to cover the case of non-authenticated bilateral channels. We describe this extension in App. G.

2.1 Formal definition

Formally, a *multiparty protocol (in the* $(\pi, \pi_{\mathcal{A}}, \theta)$*-model)* is an ITM (Interactive Turing Machine) M. It is executed together with an ITM $\mathcal{A}$ representing the *adversary*, and and ITM $\mathcal{E}$ representing the *environment*. The *real execution* of the system essentially follows that scheme from [11]. Every ITM gets as input a security parameter 1^{κ}. Initially, the environment takes some input $z \in \{0,1\}^*$ and then it activates an adversary $\mathcal{A}$ and a set $\mathcal{P}$ of parties. The adversary may (actively and adaptively) corrupt some parties. The environment is notified about these corruptions.

The set $\mathcal{P}$ (or even its size) will *not* be given as input to the honest parties. In other words: the protocol should work in the same way for any $\mathcal{P}$. On the other hand: each $P \in \mathcal{P}$ will get as input her own hashrate π and the upper bound π_{max} on the total combined hashrate of all the parties and the adversary (this will be the paramters of the protocol). The running time of $P \in \mathcal{P}$ can depend on these parameters. Note that

[4] This is important, since otherwise we could trivialize the problem by asking each user to prove that he is honest by sending a large number of messages.

$|\mathcal{P}| \cdot \pi + \pi_{\mathcal{A}} \leq \pi_{\mathsf{max}}$, but this inequality may be sharp, and even $|\mathcal{P}| \cdot \pi + \pi_{\mathcal{A}} \ll \pi_{\mathsf{max}}$ is possible, as, e.g., the adversary can use much less hashrate than the maximal amount that he is allowed to[5].

Each party $P \in \mathcal{P}$ runs the code of M. It gets as input its party identifier (pid) and some random input. We assume that all the pid's are distinct, which can be easily obtained by choosing them at random from a large domain ($\{0,1\}^\kappa$, say). Moreover the environment sends to each P some input $x_P \in \{0,1\}^*$, and at the end of its execution it outputs to $\mathcal{E}$ some value $y_P \in \{0,1\}^*$. We assume that at a given moment only one machine is active. For a detailed description on how the control is passed between the machines see [11], but let us only say that it is done via sending messages (if one party sends a message ot the other one then it "activates it"). The environment $\mathcal{E}$ can communicate with $\mathcal{A}$ and with the parties in $\mathcal{P}$. The adversary controls the network. However, we require that every message sent between two parties is always eventually delivered. Moreover, since the adversary is poly-time bounded, thus he always eventually terminates. If he does so without sending any message then the control passed to the environment (that chooses which party will be activated next).

We assume that all the parties have access to an ideal functionality $\mathcal{F}_{\mathsf{syn}}^{\pi,\pi_{\mathcal{A}},\theta}$ (depicted below) and possibly to some random oracles. The ideal functionality $\mathcal{F}_{\mathsf{syn}}^{\pi,\pi_{\mathcal{A}},\theta}$ is used to formally model the setting described informally above. Since we assumed that every message is delivered in time Δ we can think of the whole execution as divided into rounds (implicitly: of length Δ). This is essentially the "synchronous communication" model from [11] (see Section 6.5 of the Eprint version of that paper). As it is the case there, the notion of a "round" is controlled by a counter r, which is initially set to 1 and is increased each time all the honest parties send all their inputs for a given round to $\mathcal{F}_{\mathsf{syn}}^{\pi,\pi_{\mathcal{A}},\theta}$. The messages that are sent to P in a given round r are "collected" in a buffer denote L_P^r and delivered to P at the end of the rounds (on P's request). The fact that every message sent by an honest party has to arrive to another honest party within a given round is reflected as follows: the round progresses only if every honest party sent her message for a given round to the functionality (and this happens only if all of them received messages from the previous round). Recall also that sending "delayed output x to a P" means that the x is first received by $\mathcal{A}$ who can decide when x is delivered to P.

Compared to [11] there are some important differences though. First of all, since in our model the set $\mathcal{P}$ of the parties participating in the execution is known to the honest participants, thus we cannot assume that $\mathcal{P}$ is a part of the session identifier. We therefore give it directly to the functionality (note that this set is anyway known to $\mathcal{E}$, which can share it with $\mathcal{A}$).

Secondly, we do not assume that the parties can send messages directly to each other. The only communication that is allowed is through the "public channel". This is reflected by the fact that the "Send" messages produced by the parties do not specify the sender and the receiver (cf. Step 3), and are delivered to everybody . In contrast, the adversary can send messages to concrete parties (cf. Step 4)

[5] In particular it is important to stress that the assumption that the majority of the computing power is honest means that $n \cdot \pi > \pi_{\mathcal{A}}$, and *not*, as one might think, $n \cdot \pi > \pi_{\mathsf{max}}/2$ (assuming the number t of corrupt parties is zero).

Thirdly, and probably most importantly, the functionality $\mathcal{F}_{\mathsf{syn}}^{\pi,\pi_{\mathcal{A}},\theta}$ also keeps track on how much computational resources (in terms of access to the oracle H) were used by each participant in each round. To take into account the fact that the adversary may get access the oracle long before the honest parties started the execution we first allow him (in Step 1) to query this oracle adaptively (the number of these queries is bounded only by the running time of the adversary, and hence it has to be polynomial in the security parameter). Then, in each round every party $P \in \mathcal{P}$ can query H. The number of such queries is bounded by π.

Functionality $\mathcal{F}_{\mathsf{syn}}^{\pi,\pi_{\mathcal{A}},\theta}$

$\mathcal{F}_{\mathsf{syn}}^{\pi,\pi_{\mathcal{A}},\theta}$ receives a session ID $\mathsf{sid} \in \{0,1\}^*$. Moreover we assume that it obtains a list $\mathcal{P}$ of parties that were activated with sid, i.e., those parties among which synchronization is to be provided and that will issue the random oracle queries.

1. At the first activation, the functionality chooses at random a random oracle H. It then waits for queries from the adversary $\mathcal{A}$ of a form (Hash, w) (where w is from the domain of H). Each such a query is answered with $H(w)$. This phase ends when $\mathcal{A}$ sends a query Next or when it terminates its operation.
2. Initialize a round counter $r := 1$, for every party $P \in \mathcal{P}$ initialize variables $h_P := 0$ and $L_P^1 = \emptyset$. Initialize $h_{\mathcal{A}} := 0$. Send a public delayed output $(\mathsf{Init}, \mathsf{sid})$ to all parties in $\mathcal{P}$.
3. Upon receiving input $(\mathsf{Send}, \mathsf{sid}, m)$ from a party $P \in \mathcal{P}$, for every $P' \in \mathcal{P}$ set $L_{P'}^r := L_{P'}^r \cup \{m\}$ and output (sid, P, m, r) to the adversary.
4. Upon receiving input $(\mathsf{Send}, \mathsf{sid}, P', m)$ from $\mathcal{A}$ (where $P' \in \mathcal{P}$) set $L_{P'}^r := L_{P'}^r \cup \{m\}$.
5. Upon receiving (Hash, w) from $P' \in \mathcal{P} \cup \{\mathcal{A}\}$ (note that P' can either be a party or the adversary) do
   (a) if $P' \in \mathcal{P}$, where P' is not corrupt and $h_{P'} < \pi$ then reply with $H(w)$ and increment the counter: $h_{P'} := h_{P'} + 1$,
   (b) if $P' = \mathcal{A}$ and $h_{\mathcal{A}} < \pi_{\mathcal{A}}$ then reply with H and increment the counter: $h_{\mathcal{A}} := h_{\mathcal{A}} + 1$,
   (c) otherwise do nothing (since P' has already exceeded the number of allowed queries to $\mathcal{H}^{\kappa}$ in this round).
6. Upon receiving input $(\mathsf{Receive}, \mathsf{sid}, r')$ from a party $P \in \mathcal{P}$, do:
   (a) If $r' = r$ (i.e., r' is the current round), and you have received the Send message from every non-corrupt party in this round then:
      i. Increment the round number: $r := r + 1$.
      ii. For every $P' \in \mathcal{P} \cup \{\mathcal{A}\}$ reset the variable $h_{P'} := 0$.
      iii. If the size of L_P^{r-1} is at most θ then output $(\mathsf{Received}, \mathsf{sid}, L_P^{r-1})$ to P, otherwise output $\perp$ to P.
   (b) If $r' < r$ and the size of $L_P^{r'}$ is at most θ then output $(\mathsf{Received}, \mathsf{sid}, L_P^{r'})$ to P, otherwise output $\perp$ to P.
   (c) Else (i.e., $r' > r$ or not all parties in $\mathcal{P}$ have sent their messages for round r), output $\mathsf{Round\ Incomplete}$ to P.

We use a counter h_P (reset to 0 at the beginning of each new round) to track the number of times the user P queried H. The number of oracle queries that $\mathcal{A}$ can ask is bounded by $\pi_{\mathcal{A}}$ and controlled by the counter $h_{\mathcal{A}}$. Note that, once a party $P \in \mathcal{P}$ gets corrupted by $\mathcal{A}$ it looses access the oracle H. This reflects the fact that from this point the computing power of P does not count anymore as being controlled by the honest parties, and hence every call to H made by such a P has to be "performed" by the adversary (and consequently increase $h_{\mathcal{A}}$). The output of the environment on input z interacting with M, $\mathcal{A}$ and the ideal functionality $\mathcal{F}_{\mathsf{syn}}^{\pi,\pi_{\mathcal{A}},\theta}$ will be denoted $\mathsf{exec}_{M,\mathcal{A},\mathcal{E}}^{\pi,\pi_{\mathcal{A}},\theta}(z)$.

In order to define security of such execution we define an ideal functionality $\mathcal{F}$ that is also an ITM that can interact with the adversary. Its interaction with the parties is pretty simple: each party simply interacts with $\mathcal{F}$ directly (with no disturbance from the adversary). The adversary may corrupt some o the parties, in which case he learns their inputs and outputs. The functionality is notified about each corruption. At the end the environment outputs a value $\mathsf{exec}_{\mathcal{F},\mathcal{A},\mathcal{E}}(z)$.

Definition 1 *We say that a protocol M* securely implements a functionality $\mathcal{F}$ in the $(\pi, \pi_{\mathcal{A}}, \theta)$-model *if for every polynomial-time adversary $\mathcal{A}$ there exists a polynomial-time simulator $\mathcal{S}$ such that for every environment $\mathcal{Z}$ the distribution ensemble $\mathsf{exec}_{M,\mathcal{A},\mathcal{E}}^{\pi,\pi_{\mathcal{A}},\theta}$ and the distribution ensemble $\mathsf{exec}_{\mathcal{F},\mathcal{A},\mathcal{E}}$ are computationally indistinguishable (see [11] for the definition of the distribution ensembles and the computational indistinguishability).*

3 The security definition of broadcast

In this section we present the security definitions of our main construction, i.e., the broadcast protocol. We first describe its informal properties and then specify it as an ideal functionality. Let $\mathcal{P}$ be the set of parties executing Π, each of them having a device with hashrate $\pi > 0$ per time Δ. Each $P \in \mathcal{P}$ takes as input $x_P \in \{0,1\}^\kappa$, and it produces as output a multiset $\mathcal{Y}_P \subset \{0,1\}^\kappa$. The protocol is called a π_{max}-*secure broadcast protocol* if it terminates is some finite time and for any poly-time adversary $\mathcal{A}$ whose device has hashrate $\pi_{\mathcal{A}} < \pi_{\mathsf{max}}$ and who attacks this protocol the following conditions hold except with probability negligible in κ (let $\mathcal{H}$ denote the set of parties in $\mathcal{P}$ that were not corrupted by the adversary): (1) *Consistency:* All the sets $\mathcal{Y}_P$ are equal for all non-corrupt P's, i.e.: there exists a set $\mathcal{Y}$ such that for every $P \in \mathcal{H}$ we have $\mathcal{Y}_P = \mathcal{Y}$, (2) *Validity:* For every $P \in \mathcal{H}$ we have $x_i \in \mathcal{Y}$, and (3) *Bounded creation of inputs:* The number of elements in $\mathcal{Y}$ that do not come from the honest parties (i.e.:$|\mathcal{Y} \setminus \{x_P\}_{P\in\mathcal{P}}|$) is at most $\lceil \pi_{\mathcal{A}}/\pi \rceil$. This is formally defined by specifying an ideal functionality $\mathcal{F}_{\mathsf{bc}}^{\lceil \pi_{\mathcal{A}}/\pi \rceil}$ (see below). The formal definition is given below.

Definition 2 *An ITM M is a* $(\pi_{\mathsf{max}}, \theta)$-secure broadcast protocol *if for any π and $\pi_{\mathcal{A}}$ it securely implements the functionality $\mathcal{F}_{\mathsf{bc}}^{\lceil \pi_{\mathcal{A}}/\pi \rceil}$ in the $(\pi, \pi_{\mathcal{A}}, \theta)$-model (see Def. 1 from Sect. 2.1), as long as the number $|\mathcal{P}|$ of parties running the protocol (i.e. invoked by the environment) is such that $|\mathcal{P}| \cdot \pi + \pi_{\mathcal{A}} \leq \pi_{\mathsf{max}}$.*

Note that we do not require any lower bound on π other than 0. In practice, however, running this protocol will make sense only for π being a noticeable fraction of π_{max}, since the running time of our protocol is linear in π_{max}/π. This protocol is implemented in the next section.

4 The construction of the broadcast protocol

We are now ready to present the constructions of the protocols specified in Section 3. In our protocols the computational effort will be verified using so-called Proofs of Work. A *Proof-of-Work (PoW)* scheme [22], for a fixed security parameter κ is a pair of randomized algorithms: a *prover* P and a verifier V, having access to a random oracle H (in our constructions the typical input to H will be of size 2κ). The algorithm P takes as input a *challenge* $c \in \{0,1\}^\kappa$ and produces as output a *solution* $s \in \{0,1\}^*$. The algorithm V takes as input (c, s) and outputs true or false. We require that for every $c \in \{0,1\}^*$ it is that case that $\mathsf{V}(c, \mathsf{P}(c)) = \mathsf{true}$.

Functionality $\mathcal{F}_{\mathsf{bc}}^{T\,a}$

$\mathcal{F}_{\mathsf{syn}}^{T}$ receives a session ID $\mathsf{sid} \in \{0,1\}^*$. Moreover it obtains a list $\mathcal{P}$ of parties that were activated with sid.

1. At the first activation initialize the variables $\mathcal{X} := \emptyset$ and $\mathcal{X}_{\mathcal{S}} := \emptyset$, where $\mathcal{X}$ and $\mathcal{X}_{\mathcal{S}}$ are multisets. Send a public delayed output $(\mathsf{Init}, \mathsf{sid})$ to all parties in $\mathcal{P}$.
2. Upon receiving input $(\mathsf{Broadcast}, \mathsf{sid}, x)$ (where $x \in \{0,1\}^*$) from $P \in \mathcal{P}$ (with PID pid) do the following:
   (a) add x to $\mathcal{X}$, i.e., let $\mathcal{X} := \mathcal{X} \cup \{x\}$, moreover send $(\mathsf{Broadcast}, \mathsf{sid}, \mathsf{pid}, x)$ to $\mathcal{S}$,
   (b) otherwise do nothing.
3. Upon receiving $(\mathsf{Broadcast}, \mathsf{sid}, x)$ from $\mathcal{S}$:
   (a) if $|\mathcal{X}_{\mathcal{S}}| < T$ then let $\mathcal{X}_{\mathcal{S}} := \mathcal{X}_{\mathcal{S}} \cup \{x\}$,
   (b) otherwise do nothing.
4. Upon receiving $(\mathsf{Remove}, \mathsf{sid}, \mathsf{pid})$ from $\mathcal{S}$: if P with PID pid is not corrupt or such a message has already been received before then ignore it.
   Otherwise look for a string x that was added by a party with PID pid to $\mathcal{X}$ in Step 2. If no such string exists do nothing. Otherwise: remove x from the multiset $\mathcal{X}$.
5. Upon receiving $(\mathsf{Receive}, \mathsf{sid})$ from some $P \in \mathcal{P}$:
   (a) If there is some non-corrupt party $P \in \mathcal{P}$ from which no message $(\mathsf{Broadcast}, \mathsf{sid}, x)$ has been received yet then ignore this message.
   (b) Otherwise:
      i. If it is the first message $(\mathsf{Receive}, \mathsf{sid})$ received then set $\mathcal{Y} := \mathcal{X} \cup \mathcal{X}_{\mathcal{S}}$ and send $\mathcal{Y}$ to the adversary.
      ii. Output $(\mathsf{Received}, \mathsf{sid}, \mathcal{Y})$ to P.

[a] T is the bound on the number of "fake identities" that the adversary can create. Our security definition requires that $T = \lceil \pi_{\mathcal{A}}/\pi \rceil$.

We say that a PoW (P, V) has *prover complexity* t if on every input $c \in \{0,1\}^*$ the prover P makes at most t queries to the oracle H. We say that (P, V) has *verifier complexity* t' if for every $c \in \{0,1\}^\kappa$ and $s \in \{0,1\}^*$ the verifier V makes at most t' queries to the oracle H. Defining security is a little bit tricky, since we need to consider also the malicious provers that can spend considerable amount of computational effort *before* they get the challenge c. We will therefore have two parameters: $\hat{t}_0, \hat{t}_1 \in \mathbf{N}$, where $\hat{t}_0$ will be the bound on the *total time* that a malicious prover has, and $\hat{t}_1 \leq \hat{t}_0$

will be the bound on the time that a malicious prover got after he learned c. Consider the following game between a malicious prover $\hat{\mathsf{P}}$ and a verifier V: (1) $\hat{\mathsf{P}}$ adaptively queries the oracles H on the inputs of his choice, (2) $\hat{\mathsf{P}}$ receives $c \leftarrow \{0,1\}^\kappa$, (3) $\hat{\mathsf{P}}$ again adaptively queries the oracles H on the inputs of his choice, (4) $\hat{\mathsf{P}}$ sends a value $s \in \{0,1\}^*$ to V. We say that $\hat{\mathsf{P}}$ *won* if $\mathsf{V}(c,s) = \mathsf{true}$. We say that (P,V) is $(\hat{t}_0,\hat{t}_1)$-*secure with ϵ-error (in the H-model)* if for a uniformly random $c \leftarrow \{0,1\}^*$ and every malicious prover $\hat{\mathsf{P}}$ that makes in total at most $\hat{t}_0$ queries to H in the game above, and at most $\hat{t}_1$ queries after receiving c we have that $\mathbb{P}\left(\hat{\mathsf{P}}(c) \text{ wins the game}\right) \leq \epsilon$. It will also be useful to use the asymptotic variant of this notion (where κ is the security parameter). Consider a family $\{(\mathsf{P}^\kappa,\mathsf{V}^\kappa)\}_{\kappa=1}^\infty$. We will say that it is $\hat{t}_1$-*secure* if for every polynomial $\hat{t}_0$ there exists a negligible ϵ such that $(\mathsf{P}^\kappa,\mathsf{V}^\kappa)$ is $(\hat{t}_0(\kappa),\hat{t}_1)$-secure with error $\epsilon(\kappa)$. Our protocols will be based on the PoW based on the Merkle trees combined with the Fiat-Shamir transform. The following lemma is proved in App. C.

Lemma 1. *For every function $t : \mathbf{N} \to \mathbf{N}$ s.t. $t(\kappa) \geq \kappa$ there exists a family of PoWs $(\mathsf{PTree}^\kappa_{t(\kappa)},\mathsf{VTree}^\kappa_{t(\kappa)})$ has prover complexity t and verifier complexity $\lceil \kappa \log^2 t \rceil$. Moreover the family $\{(\mathsf{PTree}^\kappa_{t(\kappa)},\mathsf{VTree}^\kappa_{t(\kappa)})\}_{\kappa=1}^\infty$ is ξt-secure for every constant $\xi \in [0,1)$.*

One of the main challenges will be to prevent the adversary from precomputing the solutions to PoW, as given enough time every puzzle can be solved even by a device with a very small hashrate. Hence, each honest party P_i can accept a PoW proof only if it is computed on some string that contains a freshly generated challenge c. Since we work in a completely distributed scenario, and in particular we do not want to assume existence of a trusted beacon, thus the only way a P_i can be sure that a challenge c was fresh is that she generated it herself at some recent moment in the past (and, say, sent it to all the other parties).

This problem was already considered in [2], where the following solution was proposed. At the beginning of the protocol each party P_i creates a fresh (public key, secret key) pair $(\mathsf{pk}_i,\mathsf{sk}_i)$ (we will call the public keys *identities*) and sends to all other parties a random challenge c_i. Then, each party computes a Proof of Work on her public key and all received challenges. Finally, each party sends her public key with a Proof of Work to all other parties. Moreover, whenever a party receives a message with a given key for the first time, than it forwards it to all other parties. An honest party P_i accepts only these public keys which: (1) she received before some agreed deadline, and (2) are accompanied with a Proof of Work containing her challenge c_i. It is clear that each honest party accepts a public key of each other honest party and that after this process an adversary can not control a higher fraction of all identities that his fraction of the computational power. Hence, it may seem that the parties can later execute protocols assuming channels that are authenticated with the secret keys corresponding to these identities.

Unfortunately there is a problem with this solution. Namely it is easy to see that the adversary can cause a situation where some of his identities will be accepted by some

honest parties and not accepted by some other honest parties [6]. We present a solution to this problem in the next sections.

4.1 Ranked key sets

The main idea behind our protocol is that parties assign *ranks* to the keys they have received. If a key was received before the deadline and the corresponding proof contains the appropriate challenge, then the key is assigned a rank 0. In particular, keys belonging to honest parties are always assigned a rank 0. The rank bigger than 0 means that the key was received with some discrepancy from the protocol (e.g. it was received slightly after the deadline) and the bigger the rank is, the bigger this discrepancy was. More precisely each party P_i computes a function rank_i from the set of keys she knows $\mathcal{K}_i$ into the set $\{0, \ldots, \ell\}$ for some parameter ℓ. Note that this primitive bares some similarities with the "proxcast" protocol of Considine et al [17]. Since we will use this protocol only as a subroutine for our broadcast protocol, to save space, we present its definition without using the "ideal functionality" paradigm.

Let $\Sigma = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ be a signature scheme and let $\ell \in \mathbf{N}$ be an arbitrary parameter. Consider a multi-party protocol Π in the model from Section 2, i.e., having access to an ideal functionality $\mathcal{F}_{\mathsf{syn}}^{\pi,\pi_{\mathcal{A}}}$, where π is interpreted as the hashrate of each of the parties, and $\pi_{\mathcal{A}}$ as the hashrate of the adversary.

Each party P takes as input a security parameter 1^{κ}, and it produces as output a tuple $(\mathsf{sk}_P, \mathsf{pk}_P, \mathcal{K}_P, \mathsf{rank}_P)$, where $(\mathsf{sk}_P, \mathsf{pk}_P) \in \{0,1\}^* \times \{0,1\}^*$ is called a *(private key, public key) pair of* P, the finite set $\mathcal{K}_P \subset \{0,1\}^*$ will be some set of public keys, and $\mathsf{rank}_P : \mathcal{K}_P \to \{0, \ldots, \ell\}$ will be called a *key-ranking function (of* P*)*. We will say that an *identity* pk *was created during the execution* Π if $\mathsf{pk} \in \mathcal{K}_P$ for at least one honest P (regardless of the value of $\mathsf{rank}_P(\mathsf{pk})$). The protohancol Π is called a $\pi_{\mathcal{A}}$*-secure* ℓ*-ranked* Σ*-key generation protocol* if for any poly-time adversary $\mathcal{A}$ who attacks this protocol (in the model from Section. 2) the following conditions hold: (1) *Key-generation:* Π is a key-generation algorithm for every P, by which we mean the following. First of all, for every $i = 1, \ldots, n$ and every $m \in \{0,1\}^*$ we have that $\mathsf{Vrfy}(\mathsf{pk}_P, \mathsf{Sign}(\mathsf{sk}_P, m)) = \mathsf{true}$. Moreover sk_P can be securely used for signing messages in the following sense. Suppose the adversary $\mathcal{A}$ learns the entire information received by all the parties except of some P, and later $\mathcal{A}$ engages in the "chosen message attack" (see Sect. B) against an oracle that signs messages with key sk_P. Then any such $\mathcal{A}$ has negligible (in κ) probability of forging a valid (under key pk_P) signature on a fresh message. (2) *Bounded creation of identities:* We require that the number of created identities is at most $n + \lceil \pi_{\mathcal{A}}/\pi \rceil$ except with probability negligible in κ. (3) *Validity:* For every two honest parties P and P' we have that $\mathsf{rank}_P(P') = 0$. (4) *Consistency:* For every two honest parties P and P' and every key $\mathsf{pk} \in \mathcal{K}_P$ such that $\mathsf{rank}_P(\mathsf{pk}) < \ell$ we have that $\mathsf{pk} \in \mathcal{K}_{P'}$ and moreover $\mathsf{rank}_{P'}(\mathsf{pk}) \leq \mathsf{rank}_P(\mathsf{pk}) + 1$.

Our construction of a ranked key generation protocol RankedKeys is presented on Figure 1. The protocol RankedKeys uses a Proof of Work scheme (P, V) with prover time $\mathsf{time}_{\mathsf{P}}$ and verifier time $\mathsf{time}_{\mathsf{V}}$. Note that the algorithms P and V query the oracle

[6] This discrepancy can come from two reasons: (1) some messages could be received by some honest parties before deadline and by some other after it, and (2) a Proof of Work can containing challenges of some of the honest parties, but not all.

H. Technically this is done by sending Hash queries to the $\mathcal{F}_{\mathsf{syn}}^{\pi,\pi_{\mathcal{A}}}$ oracle, in the $\mathcal{H}^{\kappa}$-model (it also uses another hash function $F : \{0,1\}^* \to \{0,1\}^{\kappa}$ that is modeled as a random oracle, but its computation does not count into the hashrate). We can instantiate this PoW scheme with the scheme $(\mathsf{PTree}, \mathsf{VTree})$ described in App. C. The parameter ℓ will be equal to $\lceil \pi_{\mathsf{max}}/\pi \rceil$. The notation $\prec$ is described below.

Let us present some intuitions behind our protocol. First, recall that the problem with the protocol from [2] (described at the beginning of this section) was that some public keys could be recognized only by a subset of the honest parties. A key could be dropped because: (1) it was received too late; or (2) the corresponding proof did not contained the appropriate challenge. Informally, the idea behind the RankedKeys protocol is to make these conditions more granular. If we forget about the PoWs, and look only at the time constrains then our protocol could be described as follows: keys received in the first round are assigned rank 0, keys received in the second round are assigned rank 1, and so on. Since we instruct every honest party to forward to everybody all the keys that she receives, hence if a key receives rank k from some honest party, then it receives rank at most $k+1$ from all the other honest parties.

If we also consider the PoWs then the description of the protocol becomes a bit more complicated. The RankedKeys protocol consists of 3 phases. We now sketch them informally. The "challenges phase" is divided into $\ell+2$ rounds. At the beginning of the first round each P generates his challenge c_P^0 randomly and sends it to all the other parties. Then, in each k-th round each P collects the messages $a_1, \ldots, a_m$ sent in the previous round, concatenates then into $A_P^k = (a_1, \ldots, a_m)$, hashes them, and sends the result $c_P^k = F(A_P^k)$ to all the other parties.

Let $a \prec (b_1, \ldots, b_m)$ denote the fact that $a = b_i$ for some i. We say that the string b *dependents* on a if there exists a sequence $a = v_1, \ldots, v_m = b$, such that for every $1 \leq i < m$, it holds that $F(v_i) \prec v_{i+1}$.The idea behind this notion is that b could not have been predicted before a was revealed, because b is created using a series of concatenations and queries to the random oracle starting from the string a. Note that in particular c_P^k depends on $c_{P'}^{k-1}$ for any honest P, P' [7] and $1 \leq k \leq \ell$ and hence c_P^k depends on $c_{P'}^0$ for any honest P, P' and an arbitrary $1 \leq k \leq \ell+1$.

Then, during the "Proof of Work" phase each honest party P draws a random key pair $(\mathsf{sk}_P, \mathsf{pk}_P)$ and creates a proof of work[8] $\mathsf{P}(F(\mathsf{pk}_P, A_P^{\ell+1}))$. Then, she sends her public key together with the proof to all the other parties.

Later, during the "key ranking phase" the parties receive the public keys of the other parties and assign them ranks. To assign the public key pk rank k the party P requires that she receives it in the k-th round in this phase and that it is accompanied with a proof $\mathsf{P}(F(\mathsf{pk}_P, s))$ for some string s, which depends on $c_P^{\ell-k}$. Such a proof could not have been precomputed, because $c_P^{\ell-k}$ depends on c_P^0, which was drawn randomly by P at the beginning of the protocol and hence could not been predicted before the execution of the protocol. If those conditions are met, than P forwards the message with the key to the other parties. This message will be accepted by other parties, because it will be received by them in the $(k+1)$-st round of this phase and because s depends on $c_P^{\ell-k}$,

[7] This is because $c_{P'}^{k-1} \prec A_P^k$ and $F(A_P^k) = c_P^k$.

[8] The reason why we hash the input before computing a PoW is that the PoW definition requires that the challenges are random.

The **challenges phase** consists of $\ell + 2$ rounds:

- *Round* 0*:* Each party P draws a random challenge $c_P \leftarrow \{0,1\}^\kappa$ and sends his *challenge message of level* 0 equal to $(\mathsf{Challenge}^0, c_P^0)$ to all parties (including herself).
- For $k = 1$ to $\ell + 1$ in *round* k each party P does the following. It waits for the messages of a form $(\mathsf{Challenge}^{k-1}, a)$ that were sent in the previous round (note that some of them might have already arrived earlier, but, by our assumptions they are all guaranteed to arrive before round k ends). Of course if the adversary does not perform any attack then there will be exactly n such messages (one from every party), but in general there can be much more of them. Let $(\mathsf{Challenge}^{k-1}, a_1), \ldots, (\mathsf{Challenge}^{k-1}, a_m)$ be all messages received by P. Denote $A_P^k = (a_1, \ldots, a_m)$. Then P computes her challenge in round k as $c_P^k = F(A_P^k)$ and sends $(\mathsf{Challenge}^k, c_P^k)$ to all parties (this is not needed in the last rounds, i.e., when $k = \ell + 1$).

In the **Proof of Work phase** each party P performs the following.

1. Generate a fresh key pair $(\mathsf{sk}_P, \mathsf{pk}_P) \leftarrow \mathsf{Gen}(1^k)$ and compute $\mathsf{Sol}_P = \mathsf{P}(F(\mathsf{pk}_P, A_P^{\ell+1}))$ (recall that $A_P^{\ell+1}$ contains all the challenges that P received in the last round of the "challenges phase"). Note that this phase takes $\lceil \mathsf{time}_\mathsf{P}/(\pi \cdot \Delta) \rceil$ rounds.
2. Send to all the other parties a message $(\mathsf{Key}^0, \mathsf{pk}_P, A_P^{\ell+1}, \mathsf{Sol}_P)$. This message contains P's public key pk_P, the sequence $A_P^{\ell+1}$ of challenges that he received in the last round of the "challenges phase", and a Proof of Work Sol_P. The reason why she sends the entire $A_P^{\ell+1}$, instead of $F(\mathsf{pk}_P, A_P^{\ell+1})$, is that in this way every other party will be able check if her challenge was used as an input to F when $F(\mathsf{pk}_P, A_P^{\ell+1})$ was computed (this check will be performed in the next phase).

The **key ranking phase** consists of $\ell + 1$ steps, each lasting $1 + \lceil (\theta \cdot \mathsf{time}_\mathsf{V})/(\pi \cdot \Delta) \rceil$ rounds. During these steps each party P constructs a set $\mathcal{K}_P$ of ranked keys, together with a ranking function $\mathsf{rank}_P : \mathcal{K}_P \to \{0, \ldots, \ell\}$ (the later a key is added to $\mathcal{K}_P$ the higher will be its rank). Initially all $\mathcal{K}_P$'s are empty.

- *Step 0*: Each party P waits for one round for the messages of the form $(\mathsf{Key}^0, \mathsf{pk}, B^{\ell+1}, \mathsf{Sol})$ sent in the PoW phase. Then, for each such message she checks the following conditions:
  - Sol is a correct PoW solution for the challenge $F(\mathsf{pk}, B^{\ell+1})$, i.e., if $\mathsf{V}(F(\mathsf{pk}, B^{\ell+1}), \mathsf{Sol}) = \mathsf{true}$,
  - c_P^ℓ appears in $B^{\ell+1}$, i.e., $c_P^\ell \prec B^{\ell+1}$.

  If both of these conditions hold then P accepts the key pk with rank 0, i.e., P adds pk to the set $\mathcal{K}_P$ and sets $\mathsf{rank}_P(\mathsf{pk}) := 0$. Moreover P notifies all the other parties about this fact by sending to every other party a message $(\mathsf{Key}^1, \mathsf{pk}, A_P^\ell, B^{\ell+1}, \mathsf{Sol})$.
- For $k = 1$ to ℓ in *step* k each party P does the following. She waits for one round for the messages of a form $(\mathsf{Key}^k, \mathsf{pk}, B^{\ell+1-k}, \ldots, B^{\ell+1}, \mathsf{Sol})$. Then she stops listening and for each received message she checks the following conditions:
  - the key pk has not been yet added to $\mathcal{K}_P$, i.e.: $\mathsf{pk} \notin \mathcal{K}_P$,
  - Sol is a correct PoW solution for the challenge $F(\mathsf{pk}, B^{\ell+1})$, i.e., if $\mathsf{V}(F(\mathsf{pk}, B^{\ell+1}), \mathsf{Sol}) = \mathsf{true}$,
  - $c_P^{\ell-k} \prec B^{\ell+1-k}$ and for every $i = \ell + 1 - k$ to ℓ it holds that $F(B^i) \prec B^{i+1}$.

  If all of these conditions hold then P accepts the key pk with rank k, i.e., P adds pk to the set $\mathcal{K}_P$ and sets $\mathsf{rank}_P(\mathsf{pk}) := k$. Moreover if $k < \ell$ then P notifies all the other parties about this fact by sending at the end of the round to every other party a message $(\mathsf{Key}^{k+1}, \mathsf{pk}, A_P^{\ell-k}, B^{\ell+1-k}, \ldots, B^{\ell+1}, \mathsf{Sol})$ (recall that A_P^k is equal to the set of challenges received by P in the k-th round of the "challenges phase", and $F(A_P^k) = c_P^k$).

At the end of the protocol each party P outputs $(\mathsf{sk}_P, \mathsf{pk}_P, \mathcal{K}_P, \mathsf{rank}_P)$.

Fig. 1. The RankedKeys protocol.

which depends on $c_{P'}^{\ell-(k+1)}$ for any honest P'. In the effect, all other honest parties, which have not yet assigned pk a rank will assign it a rank $k+1$.

Let $\mathsf{RankedKeys}_{\mathsf{PTree}}$ denote the RankedKeys scheme instantiated with the PoW scheme $(\mathsf{PTree}^{\kappa}_{\mathsf{time_P}}, \mathsf{VTree}^{\kappa}_{\mathsf{time_P}})$ (from App. C), where $\mathsf{time_P} := \kappa^2 \cdot (\ell+2)\Delta \cdot \pi$ and $\mathsf{time_V} := \kappa \lceil \log_2 \mathsf{time_P} \rceil$. We have the following fact (its proof appears in App. D).

Lemma 2. *Assume the total hashrate of all the participants is at most π_{max}, the hashrate of each honest party if π, and the adversary can not send more than $(\theta - \lceil \pi_{\mathsf{max}}/\pi \rceil)$ messages in every round. Then the $\mathsf{RankedKeys}_{\mathsf{PTree}}$ protocol is a $\pi_{\mathcal{A}}$-secure ℓ-ranked key generation protocol, for $\ell = \lceil \pi_{\mathsf{max}}/\pi \rceil$, whose total execution takes $(2\ell+3) + \lceil \mathsf{time_P}/(\pi \cdot \Delta) \rceil + \lceil (\ell+1)(\theta \cdot \mathsf{time_V})/(\pi \cdot \Delta) \rceil$ rounds.*

The communication and message complexity of the RankedKeys protocol are analysed in App. H.

The Broadcast protocol

1. Each party P takes as input $(\mathsf{Broadcast}, \mathsf{sid}, x_P)$.
2. The parties run the RankedKeys protocol (attaching sid to every message). Let $(\mathsf{sk}_P, \mathsf{pk}_P, \mathcal{K}_P, \mathsf{rank}_P)$ be the output of each $P \in \mathcal{P}$.
3. Each party P initializes, for every $\mathsf{pk} \in \mathcal{K}_P$, a variable $\mathcal{Z}_P^{\mathsf{pk}} = \emptyset$.
4. Each $D \in \mathcal{P}$ performs the following procedure that consists of $\ell+1$ rounds (this can be executed in parallel for every D):
   - *Round* 0*:* D (we will call him the *Dealer*) sends to every other party a message $(\mathsf{sid}, x_D, \mathsf{pk}_D, \mathsf{Sign}_{\mathsf{pk}_D}(x_D, \mathsf{pk}_D))$.
   - *Round* k*, for* $1 \leq k \leq \ell$*:* Each party P except of the dealer D waits for the messages of the form $(\mathsf{sid}, v, \mathsf{pk}_D, \mathsf{Sign}_{\mathsf{sk}_{a_1}}(v, \mathsf{pk}_D), \ldots, \mathsf{Sign}_{\mathsf{sk}_{a_k}}(v, \mathsf{pk}_D))$. Such a message is accepted by P if:
     (1) all signatures are valid and are corresponding to different public keys,
     (2) $\mathsf{pk}_{a_1} = \mathsf{pk}_D$,
     (3) $\mathsf{pk}_{a_j} \in \mathcal{K}_P$ and $\mathsf{rank}_P(\mathsf{pk}_{a_j}) \leq k$ for $1 \leq j \leq k$, and
     (4) $v \notin \mathcal{Z}_P^{\mathsf{pk}_D}$ and $|\mathcal{Z}_P^{\mathsf{pk}_D}| < 2$.
     If a message is accepted then P adds v to her set $\mathcal{Z}_P^{\mathsf{pk}_D}$ and if moreover $k < \ell$, than she sends a message $(\mathsf{sid}, v, \mathsf{pk}_D, \mathsf{Sign}_{\mathsf{pk}_{a_1}}(v, \mathsf{pk}_D), \ldots, \mathsf{Sign}_{\mathsf{pk}_{a_k}}(v, \mathsf{pk}_D), \mathsf{Sign}_{\mathsf{pk}_P}(v, \mathsf{pk}_D))$ to all other parties.
5. Each party P determines the set $\mathcal{Y}_P$ as the union over all $\mathcal{Z}_P^{\mathsf{pk}}$'s that are of size 1, i.e.: $\mathcal{Y}_P = \bigcup_{\mathsf{pk}:|\mathcal{Z}_P^{\mathsf{pk}}|=1} \mathcal{Z}_P^{\mathsf{pk}}$. It outputs $(\mathsf{Received}, \mathsf{sid}, \mathcal{Y}_P)$.

The Broadcast protocol. The reason why ranked key sets are useful is that they allow to construct a reliable broadcast protocol, which is secure against an adversary that has an arbitrary hashrate. The only assumption that we need to make is that the total hashrate in the system is bounded by some π_{max} and the adversary cannot send more than $\theta - n$ messages in one interval (for some parameter θ). Our protocol, denoted Broadcast, works in time that is linear in $\ell = \lceil \pi_{\mathsf{max}}/\pi \rceil$ plus the execution time of RankedKeys. It is based on a classical authenticated Byzantine agreement by Dolev and Strong [20]

(and is similar to the technique used to construct broadcast from a proxcast protocol [17]). The protocol is depicted on the figure above and it works as follows. First the parties execute the $\mathsf{RankedKeys}$ protocol with parameters π, π_{max} and θ, built on top of a signature scheme $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ — $\mathsf{Sign}_{\mathsf{pk}}$ denotes a signatures computed using a *private* key corresponding to a public key pk. For convenience assume that every signature σ contains information identifying the public key that was used to compute it. Let $(\mathsf{sk}_P, \mathsf{pk}_P, \mathcal{K}_P, \mathsf{rank}_P)$ be the output of each P after this protocol ends (recall that $(\mathsf{sk}_P, \mathsf{pk}_P)$ is her key pair, $\mathcal{K}_P$ is the set of public keys that she accepted, rank_P is the key ranking function). Then, each party $D \in \mathcal{P}$ executes in parallel the procedure from Step 4. During the execution each party P maintains a set $\mathcal{Z}_P^{\mathsf{pk}_D}$ initialized with $\emptyset$. The output of each party is equal to the only elements of this set (if $\mathcal{Z}_P^{\mathsf{pk}_D}$ is a singleton) or $\perp$ otherwise.

The following lemma is proven in Appendix I.

Lemma 3. *The* $\mathsf{Broadcast}$ *protocol is a* $(\pi_{\mathsf{max}}, \theta)$*-secure broadcast protocol.*

5 Applications

Multiparty computations. As already mentioned before, the $\mathsf{Broadcast}$ protocol can be used to establish a group of parties that can later perform the MPC protocols. For the lack of space we only sketch this method here. The main idea is as follows. First, each party P generates its key pair $(\mathsf{sk}_P, \mathsf{pk}_P)$. Then, it uses the broadcast protocol to send to all the other parties the public key pk_P. Let $\pi_{\mathcal{A}}$ be the computing power of the adversary, and let t be the number of parties that he corrupted. From the properties of the broadcast protocol he can make the honest parties accept at most $\lceil \pi_{\mathcal{A}}/\pi \rceil$ keys pk_P chosen by the adversary. Additionally, the adversary knowns up to t secret keys of the parties that she corrupted. Therefore altogether there are at most $\lceil \pi_{\mathcal{A}}/\pi \rceil + t$ keys pk_P such that the adversary knows the corresponding secret keys sk_P. The total number of keys is $\lceil \pi_{\mathcal{A}}/\pi \rceil + n$ (where n is the number of the honest parties).

Given such a setup the parties can now simulate the secure channels, even if initially they did not know each others identities (i.e. in the model from Section 2), by treating the public keys as the identities. More precisely: whenever a party P wants to send a message to P' (known to P by her public key $\mathsf{pk}_{P'}$) she would use the standard method of encryption (note that in the adaptive case this is secure only if the encryption scheme is non-committing) and digital signatures to establish a secure channel (via the insecure broadcast channel available in the model) with P'. Hence the situation is exactly as in the standard MPC settings with the private channels between $\lceil \pi_{\mathcal{A}}/\pi \rceil + n$ parties. We can now use well-known fact that simulating any functionality is possible in this case [12]. In case we require that the protocol has guaranteed termination we need an assumption that the majority of the participants is honest [30], i.e, that $\lceil \pi_{\mathcal{A}}/\pi \rceil + t < (n - t)$. Suppose we ignore the rounding up (observe that in practice we can make $\lceil \pi_{\mathcal{A}}/\pi \rceil$ arbitrarily close to $\pi_{\mathcal{A}}/\pi$ by making π small). Then we obtain the condition $\pi_{\mathcal{A}} + t\pi < (n-t)\pi$. The left hand side of this inequality can be interpreted as the "total computing power of the adversary" (including his own computing power and the one of corrupt parties), and the right hand side can be interpreted as the total computing power of the honest parties. Therefore we get that every functionality can he simulated (with guaranteed termination) as long as the majority of the computing power is controlled by the honest parties. This argument will be formalized in the full version of this paper.

Unpredictable beacon generation. The Broadcast protocols can also be used to produce unpredictable beacons even if there is no honest majority of computing power in the system by letting every party broadcast a random nonce and then hashing the result. This is described in more detail is App. J. In App. K we discuss also the possibility of creating provable secure currencies using our techniques.

References

1. M. Andrychowicz, S. Dziembowski, D. Malinowski, and Ł. Mazurek. Secure Multiparty Computations on Bitcoin, 2014. 35th IEEE Symposium on Security and Privacy (Oakland). ACM.
2. James Aspnes, Collin Jackson, and Arvind Krishnamurthy. Exposing computationally-challenged byzantine impostors. *Department of Computer Science, Yale University, New Haven, CT, Tech. Rep*, 2005.
3. Giuseppe Ateniese, Ilario Bonacina, Antonio Faonio, and Nicola Galesi. Proofs of space: When space is of the essence. Cryptology ePrint Archive, Report 2013/805, 2013. `http://eprint.iacr.org/2013/805`.
4. L Babai. Trading group theory for randomness. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, STOC '85, pages 421–429, New York, NY, USA, 1985. ACM.
5. Adam Back. Hashcash - a denial of service counter-measure, 2002. technical report.
6. Lear Bahack. Theoretical bitcoin attacks with less than half of the computational power (draft). *arXiv preprint arXiv:1312.7013*, 2013.
7. Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In V. Ashby, editor, *ACM CCS 93*, pages 62–73, Fairfax, Virginia, USA, November 3–5, 1993. ACM.
8. Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88, pages 1–10, New York, NY, USA, 1988. ACM.
9. Bitcoin. Wiki. en.bitcoin.it/wiki/.
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*, volume 5628 of *LNCS*, pages 325–343, Accra Beach, Barbados, February 23–26, 2009. Springer.
11. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145, Las Vegas, Nevada, USA, October 14–17, 2001. IEEE Computer Society Press.
12. Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. Universally composable two-party and multi-party secure computation. In *34th ACM STOC*, pages 494–503, Montréal, Québec, Canada, May 19–21, 2002. ACM.
13. David Chaum, Claude Crépeau, and Ivan Damgard. Multiparty unconditionally secure protocols. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88, pages 11–19, New York, NY, USA, 1988. ACM.
14. Jeremy Clark and Urs Hengartner. On the use of financial data as a random beacon. Cryptology ePrint Archive, Report 2010/361, 2010. `http://eprint.iacr.org/2010/361`.

15. Fabien Coelho. An (almost) constant-effort solution-verification proof-of-work protocol based on merkle trees. In *Progress in Cryptology - AFRICACRYPT 2008, First International Conference on Cryptology in Africa, Casablanca, Morocco, June 11-14, 2008. Proceedings*, volume 5023, pages 80–93. Springer, 2008.
16. Sophia Yakoubov Conner Fromknecht, Dragos Velicanu. A decentralized public key infrastructure with identity retention. Cryptology ePrint Archive, Report 2014/803, 2014. `http://eprint.iacr.org/`.
17. Jeffrey Considine, Matthias Fitzi, Matthew K. Franklin, Leonid A. Levin, Ueli M. Maurer, and David Metcalf. Byzantine agreement given partial broadcast. *Journal of Cryptology*, 18(3):191–217, July 2005.
18. Nicolas T. Courtois. On the longest chain rule and programmed self-destruction of crypto currencies. *CoRR*, abs/1405.0534, 2014.
19. Nicolas T Courtois and Lear Bahack. On subversive miner strategies and block withholding attack in bitcoin digital currency. *arXiv preprint arXiv:1402.1718*, 2014.
20. Danny Dolev and H. Raymond Strong. Authenticated algorithms for byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983.
21. 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.
22. Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In Ernest F. Brickell, editor, *CRYPTO'92*, volume 740 of *LNCS*, pages 139–147, Santa Barbara, CA, USA, August 16–20, 1992. Springer.
23. Cynthia Dwork, Moni Naor, and Hoeteck Wee. Pebbling and proofs of work. In Victor Shoup, editor, *CRYPTO 2005*, volume 3621 of *LNCS*, pages 37–54, Santa Barbara, CA, USA, August 14–18, 2005. Springer.
24. Stefan Dziembowski, Sebastian Faust, Vladimir Kolmogorov, and Krzysztof Pietrzak. Proofs of space. Cryptology ePrint Archive, Report 2013/796, 2013. `http://eprint.iacr.org/2013/796`.
25. Shimon Even, Oded Goldreich, and Abraham Lempel. A randomized protocol for signing contracts. In David Chaum, Ronald L. Rivest, and Alan T. Sherman, editors, *CRYPTO'82*, pages 205–210, Santa Barbara, CA, USA, 1982. Plenum Press, New York, USA.
26. Ittay Eyal and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. *arXiv preprint arXiv:1311.0243*, 2013.
27. Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *CRYPTO'86*, volume 263 of *LNCS*, pages 186–194, Santa Barbara, CA, USA, August 1986. Springer.
28. Juan A. Garay, Jonathan Katz, Ueli Maurer, Björn Tackmann, and Vassilis Zikas. Rational protocol design: Cryptography against incentive-driven adversaries. In *54th FOCS*, pages 648–657, Berkeley, CA, USA, October 26–29, 2013. IEEE Computer Society Press.
29. Christina Garman, Matthew Green, and Ian Miers. Decentralized anonymous credentials. Cryptology ePrint Archive, Report 2013/622, 2013. `http://eprint.iacr.org/2013/622`.
30. O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. STOC, 1987.
31. S Goldwasser and M Sipser. Private coins versus public coins in interactive proof systems. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, STOC '86, pages 59–68, New York, NY, USA, 1986. ACM.
32. S. Dov Gordon and Jonathan Katz. Rational secret sharing, revisited. In Roberto De Prisco and Moti Yung, editors, *SCN 06*, volume 4116 of *LNCS*, pages 229–241, Maiori, Italy, September 6–8, 2006. Springer.
33. Nermin Hajdarbegovic. Ibm sees role for block chain in internet of things. *Coindesk Magazine*, September 2014. www.coindesk.com/ibm-sees-role-block-chain-internet-things.

34. Sergei Izmalkov, Silvio Micali, and Matt Lepinski. Rational secure computation and ideal mechanism design. In *46th FOCS*, pages 585–595, Pittsburgh, PA, USA, October 23–25, 2005. IEEE Computer Society Press.
35. Jonathan Katz, Ueli Maurer, Björn Tackmann, and Vassilis Zikas. Universally composable synchronous computation. In Amit Sahai, editor, *TCC 2013*, volume 7785 of *LNCS*, pages 477–498, Tokyo, Japan, March 3–6, 2013. Springer.
36. Jonathan Katz, Andrew Miller, and Elaine Shi. Pseudonymous secure computation from time-lock puzzles. Cryptology ePrint Archive, Report 2014/857, 2014. `http://eprint.iacr.org/`.
37. Sunny King and Scott Nadal. Ppcoin: Peer-to-peer crypto-currency with proof-of-stake. .
38. Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.
39. Ralph C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *CRYPTO'87*, volume 293 of *LNCS*, pages 369–378, Santa Barbara, CA, USA, August 16–20, 1987. Springer.
40. Andrew Miller, Ari Juels, Elaine Shi, Bryan Parno, and Jonathan Katz. Permacoin: Repurposing bitcoin work for data preservation. *online full version: http://cs. umd. edu/ amiller/permacoin full. pdf*, 2014.
41. Andrew Miller, Elaine Shi, and Jonathan Katz. Non-outsourceable scratch-off puzzles to discourage bitcoin mining coalitions, 2014.
42. Tal Moran and Moni Naor. Split-ballot voting: everlasting privacy with distributed trust. In Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson, editors, *ACM CCS 07*, pages 246–255, Alexandria, Virginia, USA, October 28–31, 2007. ACM.
43. Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Consulted*, 1(2012):28, 2008.
44. Michael O. Rabin. Transaction protection by beacons. *Journal of Computer and System Sciences*, 27(2):256 – 267, 1983.
45. Bitcoin Talk. [ann] litecoin - a lite version of bitcoin. launched! bitcointalk.org/index.php?topic=47417.0, Accessed on 27.05.2014.
46. Bitcoin Wiki. Denial of service (dos) attacks. en.bitcoin.it/wiki/Weaknesses , Accessed on 26.09.2014.
47. Bitcoin Wiki. Network. en.bitcoin.it/wiki/Network, Accessed on 26.09.2014.
48. Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *23rd FOCS*, pages 160–164, Chicago, Illinois, November 3–5, 1982. IEEE Computer Society Press.

A Description of Bitcoin

A.1 A short introduction to Bitcoin.

The PoWs that Bitcoin uses are based on computing the value of a hash function H (more concretely: H is the SHA256 function) on multiple inputs. Therefore the "computing power" is measured in terms of the speed at which a given party can compute a certain hash function. This speed is called a *hashrate*. Currently almost all of the computing power in Bitcoin comes from dedicated hardware, as computing SHA256 in software is too inefficient. Bitcoin contains incentives for the users to contribute their computing power to the system. We do not describe them here (the reader may find a description of this incentive system, e.g., in [43,9,1]). The same idea is used in several

other cryptocurrencies like Litecoin [45] (that, instead of SHA256, uses a hash function *Script*, which is designed in such a way, that it is difficult to implement it in hardware efficiently), or Peercoin [37] (in combination with the so-called "Proof of Stake" which we will not describe here).

The "trusted functionality" that the parties emulate is simply a public ledger on which the parties can post their transactions. From the security perspective the Bitcoin ledger is very similar to a broadcast channel: every party should be able to broadcast some value to all the other parties (i.e.: post it on the ledger) and in case a malicious party posts several different values, the honest participants should be able to reach a consensus about which of them is accepted as a valid one. One additional property (compared to the standard broadcast definition) that the Bitcoin ledger has is the *public verifiability*, which in particular means that the parties that joined the system long time after a given value v was posted can verify that v appeared on the ledger.

We do not describe here the exact syntax of the Bitcoin transactions, as it is not relevant to this paper. The Bitcoin ledger is implemented 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 (this is the only "trusted setup" that is required in Bitcoin, however, as we describe later, some heuristic methods were applied to prove that B_0 was generated honestly). Each new block B_i contains a list T_i of new transactions, the 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 on average each 10 minutes.

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 computing 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 post a transaction on the network, he sends it to other nodes. The receivers validate this transaction and add it to the block they are mining. When some node solves the mining problem, it broadcasts the new block B_i to the network. Nodes obtain a new block, check if the transactions are correct, that it contains the hash of the previous block B_{i-1} and that $H(B_i)$ starts with an appropriate number of zeros. If yes, then they accept it and start 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.

In [43] it was claimed that this system is secure as long as the majority of computing power is controlled by the honest users. 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. Unfortunately, no proof of this statement, or even a formal security definition was provided. In our opinion, this is one of the main weaknesses of Bitcoin. We discuss it in the next section.

A.2 Lack of security proof and the dishonest minority attacks on Bitcoin.

While the hard-core cryptography part (like the choice of the signature schemes and the hash functions) in the most popular cryptocurrency systems looks secure, what seems much less understood is the system of maintaining the trusted ledger. This is not just a theoretical weakness. In fact, recently in a very interesting paper Ittay Eyal and Emin Gun Sirer [26] have shown that Nakamoto's claim that no dishonest majority can break the system is false. We will not present their attack (called the "selfish mining") in detail here, as it depends on Bitcoin incentive mechanism that we do not describe in this paper. Let us only say that from a very high level view their strategy for the dishonest minority is to keep the newly mined blocks secret, and to send them over the network only if certain conditions are satisfied.

One may argue, that performing such attacks by miners is financially irrational, because such attacks can be easily noticed, what would cause a collapse in the Bitcoin price and subsequently would make mining less profitable. Even if this argument is sound, it shows that we need some additional assumptions to make Bitcoin secure, other than "the majority is honest", what was claimed in the original Nakamoto's paper.

Another claim from the original work of Nakamoto, which turned out not to be completely true is that a probability of reverting a transaction in a block on top of which there are b other blocks decreases exponentially with b. Surprisingly, Lear Bahack [6] has recently shown that this claim is no longer true if we consider the *difficulty adjustment* algorithm, which is used in Bitcoin to gradually make mining new blocks more difficult as the total computational power of all miners grows. In his paper Bahack shows than an adversary can discard a block on any depth with a probability 1 regardless of his computational power if he is willing to wait long enough. An interesting survey of the known strategies for dishonest miners and their discussion can be found in [19].

In our opinion all of these weakness could have been avoided (or at least they could be known in advance) if Bitcoin came with a formal model and mathematically proven security. Unfortunately, it was not the case. This was probably partly due to the fact that designing a complete model for cryptocurrencies is a challenging and ambitious project. For example such a model should take into account the incentive system for mining, and hence should include elements of the *rational cryptography* framework [34,32,28].

A.3 Other applications of the blockchain technology

Garman et al. [29] propose to use the Bitcoin's PoW-based distributed consensus in order to create a decentralized system for anonymouns credentials, and Fromknecht et al. [16] present an idea of applying the same technology to create a decentralized PKI, also IBM recently announced an "Adept" system [33], where the blockchain technology is used for the "Internet of Things" applications. Many of these applications require only some kind of a "distributed consensus mechanism", and hence one can use our protocols there (as a replacement of the blockchain). We also believe that our protocols can potentially lead to improved constructions of new cryptocurrencies.

A.4 Bitcoin as a source of unpredictable beacons?

Treating Bitcoin blocks as a source of randomness can make sense for some applications, even for running new cryptocurrencies, e.g., a genesis block for a new currency can be based on some Bitcoin block. This solution is not fully satisfactory from a theoretical point of view since it suffers from a "chicken or egg problem": to create a cryptocurrency one needs to assume that another cryptocurrency is already running. Also, from the practical point of view it has some weaknesses. In particular, as described above, Bitcoin is not fully secure, and moreover one of the attacks described in the literature [26] is based on the strategy of withholding blocks. Associating some external (possibly financial) incentive for publishing only blocks that satisfy certain properties, can additionally change the economical model of Bitcoin, and needs to be taken into account when the security of the whole system is considered[9].

B Signature schemes

We recall here the definition of the signature schemes. This is done for the reference, as some of our protocol definitions can be viewed as extensions of this standard definitions. A *signature scheme* is a tuple of poly-time randomized algorithms $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ where Gen is a *key generation algorithm* that takes as input a security parameter $1^\kappa \in \mathbf{N}$ and produces as output a key pair $(\mathsf{pk}, \mathsf{sk}) \in \{0,1\}^* \times \{0,1\}^*$. The *signing algorithm* Sign takes as input the *private key* sk and a message $m \in \{0,1\}^*$ and produces as output a *signature* $\sigma = \mathsf{Sign}(\mathsf{sk}, m)$, and the *verification algorithm* Vrfy takes as input the *public key* pk, a message m and a signature $\sigma \in \{0,1\}^*$ and produces as output a bit $\mathsf{Vrfy}(\mathsf{pk}, m, \sigma) \in \{\mathsf{true}, \mathsf{false}\}$. We require that always $\mathsf{Vrfy}(\mathsf{pk}, \mathsf{Sign}(\mathsf{sk}, m), m) = \mathsf{true}$. The security of the signature scheme is defined by the following game played be a poly-time adversary $\mathcal{A}$ (for some fixed 1^κ): (1) let $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Gen}(1^\kappa)$, (2) the adversary $\mathcal{A}$ learns 1^κ and pk, (2) the adversary can apply a *chosen-message attack*, i.e., he can adaptively specify a sequence of messages $m_1, \ldots, m_a$ and learn $\mathsf{Sign}(\mathsf{sk}, m_i)$ for each m_i, and (3) the adversary produces a pair $(\hat{m}, \hat{\sigma})$. We say that $\mathcal{A}$ *won* if $\mathsf{Vrfy}(\mathsf{pk}, \hat{m}, \hat{\sigma}) = \mathsf{true}$ and $\hat{m} \notin \{m_1, \ldots, m_a\}$. We say

[9] Actually, this is one of the reasons why modeling and proving Bitcoin's security is so hard: one would need to also consider these type of "environmental conditions" to model the whole economic system accurately.

that $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ is *secure* if for every $\mathcal{A}$ the probability that $\mathcal{A}$ wins is negligible in κ.

C Proofs of Work based on the Merkle trees

The goal of this section is to prove Lemma 1 that states that for every function $t : \mathbf{N} \to \mathbf{N}$ s.t. $t(\kappa) \geq \kappa$ there exists a family of PoWs $(\mathsf{PTree}^{\kappa}_{t(\kappa)}, \mathsf{VTree}^{\kappa}_{t(\kappa)})$ has prover complexity t and verifier complexity $\lceil \kappa \log^2 t \rceil$. Moreover it states the family $\{(\mathsf{PTree}^{\kappa}_{t(\kappa)}, \mathsf{VTree}^{\kappa}_{t(\kappa)})\}_{\kappa=1}^{\infty}$ is ξt-secure for every constant $\xi \in [0, 1)$. This lemma will be proven at the every end of this section. We first need to introduce some auxiliary machinery.

Binary trees. A *(binary) tree* is a finite non-empty set $T \subset \{0,1\}^*$ that is prefix-closed (i.e.: for every $x \in T$ every prefix of x is also in T). We say that *a tree T' is a sub-tree of* T if $T' \subseteq T$. Every element $x \in T$ is called a *node* and its length $|x|$ is called its *depth*. The *depth of the tree* T is equal to the maximal depth of its nodes. The *size of the tree* T is equal to $|T|$. The empty string ϵ is called *the root of* T. For every $x \in T$ the elements $x||0 \in T$ and $x||1 \in T$ will be called the *left (resp.: right) child of* x (where "||" denotes concatenation). Moreover $x||0$ and $x||1$ will be called *siblings* (of each other). A node without children in T will be called a *leaf*. A *path* is a set of nodes $v_1, \ldots, v_i$ such that each v_{i+1} is a child of v_i. Sometimes it will be useful we to fix an ordering $\preccurlyeq$ on the nodes of a binary tree. We will assume that if v_0, v_1 are nodes of the same depth then $\preccurlyeq$ compares the nodes accordingly to the lexicographic order, and otherwise $v_0 \preccurlyeq v_1$ if and only if the depth of v_0 is smaller or equal to the depth of v_1. A tree T is called *complete* if every leaf $x \in T$ has the same depth d. It is easy to see that in this case d has to be equal to $\log_2(|T|+1)-1$. A tree is *almost complete* if every leaf $x \in T$ has depth either $\lfloor \log_2(|T|+1) - 1 \rfloor$ or $\lceil \log_2(|T|+1) - 1 \rceil$ (hence every almost complete tree of size $2^i - 1$, for a natural i, is complete). It is easy to see that every almost complete tree of size t has exactly $\lceil t/2 \rceil$ leaves. To make the almost complete tree unique for every tree size t we assume that all the leafs of length $\lceil \log_2(|T|+1) \rceil$ are always "shifted to the left", i.e., for every node $\lambda \in T$ all the other strings that are smaller according to the $\preccurlyeq$ ordering are also in T. A *labelled binary tree* is a pair (T, f), where T is a binary tree and f is a *labelling function* of a type $T \to \mathcal{X}$ (for some set $\mathcal{X}$ of *labels*). In this case $f(\lambda)$ (for $\lambda \in T$) will be called a *label* of the node λ. We will often abuse the notation and use T also as the labelling function (i.e. $T(\lambda)$ will denote the label of λ).

Random oracle model. As already mentioned above we model the hash functions as random oracles [7]. It will be convenient to assume that our algorithms have access to a family $\mathcal{H} = \{H_\lambda\}_{\lambda \in \Lambda}$ of random oracles, where the finite set Λ will be fixed for every input size of a given algorithm (and, in particular $|\Lambda|$ will never be larger than the running time of the algorithm). Clearly one random oracle is enough to simulate the existence of such a family (as λ can be treated simply as an additional argument). Without loss of generality assume that every algorithm $\mathcal{A}$ that we consider never queries each random oracle H_λ on the same input more than once. The hash functions that

we use often take inputs from the set $\{0,1\}^\kappa \times \{0,1\}^\kappa \cup \{0,1\}^\kappa$ (for some natural parameter κ). In this case we will denote each individual hash function as H^κ_λ, and the family $\{H^\kappa_\lambda\}_{\lambda\in\Lambda}$ of such functions as $\mathcal{H}^\kappa$. Some additional machinery needed for analyzing the random oracles appears in Appendix C.

Consider an algorithm $\mathcal{A}$ running in time $\hat{t}$ and look at his calls to the random oracles in $\mathcal{H}^\kappa$ during his execution. Call an execution *canonical* if it never happened that a malicious prover "guessed" an output of any random oracle. More formally, an execution is canonical if for every $\lambda \in \Lambda$ and every call q to H^κ_λ equal to $(w, v) \in \{0,1\}^\kappa \times \{0,1\}^\kappa$ or $w \in \{0,1\}^\kappa$ it is never the case that $\mathcal{A}$ receives v or w from $H^\kappa_{\lambda'}$ (for some possibly different $\lambda' \in \Lambda$) *after* he issues q . We now have the following.

Lemma 4. *For every $\mathcal{A}$ running in time $\hat{t}$ the probability that the execution of $\mathcal{A}$ is* not *canonical is at most $2\hat{t}^2 \cdot 2^{-\kappa}$.*

Proof. Consider a value $u \in \{0,1\}^\kappa$ that $\mathcal{A}$ received from any $H_{\lambda'}$ on some query q. Since we assumed that $\mathcal{A}$ never queries $H_{\lambda'}$ more than once on q, thus before $\mathcal{A}$ received $H(q)$, it appeared uniform to him. Therefore the probability that $\mathcal{A}$ *earlier* queried H on $u, (w, u)$ or (u, w) (for some $w \in \{0,1\}^\kappa$) is at most $2\hat{t} \cdot 2^{-\kappa}$. Since $\mathcal{A}$ issues at most $\hat{t}$ oracle queries hence (by the union bound) we get that the probability that the execution of $\mathcal{A}$ is *not* canonical is at most $2\hat{t}^2 \cdot 2^{-\kappa}$. □

Suppose every H_λ is of a type $H : \{0,1\}^* \to \{0,1\}^\kappa$. Call an execution *collision-free* if it never happened that there were two different calls q and q' to H_λ (for some $\lambda \in \Lambda$) such that $H_\lambda(q) = H_\lambda(q')$.

Lemma 5. *For every $\mathcal{A}$ running in time $\hat{t}$ the probability that the execution of $\mathcal{A}$ is* not *collision-free is at most $\hat{t}^2 \cdot 2^{-\kappa}$.*

Proof. For every output w of H_λ the probability that it was output already to some earlier call is at most $\hat{t} \cdot 2^{-\kappa}$. Thus, the probability that this happens for *some* output w is at most $\hat{t}^2 \cdot 2^{-\kappa}$. □

Merkle trees. We use a standard cryptographic tool called the *Merkle trees* [39]. Take any $\kappa \in \mathbf{N}, c \in \{0,1\}^\kappa$ and $\beta \in \mathbf{N}$. Let Λ_t be an almost complete binary tree of size $t = 2\beta$. Let $\mathcal{H}^\kappa$ be a family of hash functions, i.e.: $\{H^\kappa_\lambda\}_{\lambda\in\Lambda_t}$, where each H^κ_λ is of a type $\{0,1\}^* \times (\{0,1\}^* \cup \{0,1\}^\kappa \times \{0,1\}^\kappa) \to \{0,1\}^\kappa$. Define a function $\mathsf{MHash}^{\mathcal{H}^\kappa} : (\{0,1\}^\kappa)^\beta \to \{0,1\}^\kappa$ as on Figure 2.

The Merkle trees are useful since they allow for very efficient proofs that a given value v_i was used to calculate $r = \mathsf{MHash}^{\mathcal{H}^\kappa}(v_1, \ldots, v_\beta)$. To be more precise, there exists a procedure $\mathsf{MProof}^{\mathcal{H}}$ (described on Figure 2) that on input $(v_1, \ldots, v_\beta) \in (\{0,1\}^\kappa)^\beta$ outputs a *certificate* $\mathcal{M}'$ that proves that v_i was used to calculate $r = \mathsf{MHash}(v_1, \ldots, v_\beta)$. The certificate $\mathcal{M}'$ is a labelled subtree of $\mathcal{M}$ *induced by* i, i.e. consisting of all nodes on a path from the leaf λ_i to the root and their siblings. Such a proof can be verified by a procedure $\mathsf{MVrfy}^{\mathcal{H}^\kappa}$ (described on Figure 2) that takes as input $v_i \in \{0,1\}^\kappa, i \in \{1, \ldots, \beta\}, r \in \{0,1\}^\kappa$ and a labelled tree $\mathcal{M}'$ and outputs true if the labeling of $\mathcal{M}'$ is correct. We clearly have that $\mathsf{MVrfy}^{\mathcal{H}^\kappa}(v_i, i, r, \mathcal{M}')$ is equal to

true if $r = \mathsf{MHash}(v_1, \ldots, v_\beta)$ and $\mathcal{M}' = \mathsf{MProof}((v_1, \ldots, v_\beta), i)$. It is easy to see that the following holds.

Function $\mathsf{MHash}^{\mathcal{H}^\kappa}$**:**

On input $(v_1, \ldots, v_\beta)$ the function $\mathsf{MHash}^{\mathcal{H}^\kappa}$ first recursively labels the tree Λ_t (denote the resulting *labelled* almost complete binary tree as $\mathcal{M}_c^{\mathcal{H}^\kappa}$) and then outputs the root of this tree. This is done as follows.

1. Let $\lambda_1 \preccurlyeq \cdots \preccurlyeq \lambda_\beta$ be the leaves of $\mathcal{M}_c^{\mathcal{H}^\kappa}$. Each leaf λ_i is labelled with v_i.
2. If a node λ has one child, and this child is labelled with v then let $\mathcal{M}_c^{\mathcal{H}^\kappa}(\lambda) := H_\lambda(v)$.
3. If a node λ has two children $\lambda||0$ and $\lambda||1$ and their respective labels are v and w then let $\mathcal{M}_c^{\mathcal{H}^\kappa}(\lambda) := H_\lambda(v, w)$.
4. Output the label of the root of $\mathcal{M}_c^{\mathcal{H}^\kappa}$.

Function $\mathsf{MProof}^{\mathcal{H}^\kappa}$**:**

On input $(v_1, \ldots, v_\beta) \in (\{0,1\}^\kappa)^\beta$ and $i \in \{1, \ldots, \beta\}$ outputs a labelled binary tree $\mathcal{M}'$ that is a sub-tree of $\mathcal{M}_c^{\mathcal{H}^\kappa}$ constructed as follows:

1. Run $\mathsf{MHash}^{\mathcal{H}^\kappa}(v_1, \ldots, v_\beta)$. Let $\mathcal{M}_c^{\mathcal{H}^\kappa}$ be the output of this function (in many cases $\mathcal{M}_c^{\mathcal{H}^\kappa}$ will be already stored in user's memory, and hence there will be no need to compute it),
2. Define $\Lambda' := \{(\lambda||j) \in \mathcal{M}_c^{\mathcal{H}^\kappa} : \lambda \text{ is a prefix of } \lambda_i, \lambda \neq \lambda_i \text{ and } j \in \{0,1\}\} \cup \{\epsilon\}$ (recall that λ_i is an i-th leaf of Λ accordingly to the $\preccurlyeq$ ordering). In other words, Λ' is a subtree of Λ *induced by* λ_i (i.e. containing a path from λ_i to the root, together with all the siblings of the nodes on this path).
3. Let $\mathcal{M}'$ be the tree labelled like $\mathcal{M}_c^{\mathcal{H}^\kappa}$, but containing only nodes from Λ'.

Function $\mathsf{MVrfy}^{\mathcal{H}^\kappa}$

Take as input $v_i \in \{0,1\}^\kappa, i \in \{1, \ldots, \beta\}, r \in \{0,1\}^\kappa$ and a labelled tree $\mathcal{M}'$, and output true if the labeling of $\mathcal{M}'$ is correct. More precisely, it outputs true if the following conditions hold:

1. the leaf λ_i has label v_i in $\mathcal{M}'$,
2. every non-leaf node λ in $\mathcal{M}'$ is labelled by the appropriate hash of its children,
3. the root of $\mathcal{M}'$ is labelled with r.

Otherwise it outputs false.

Fig. 2. The $\mathsf{MHash}^{\mathcal{H}^\kappa}$, $\mathsf{MProof}^{\mathcal{H}^\kappa}$ and $\mathsf{MVrfy}^{\mathcal{H}^\kappa}$ functions.

Lemma 6. *Consider any algorithm $\mathcal{A}$ running in time at most $\hat{t}$. The algorithm $\mathcal{A}$ gets as input $(v_1, \ldots, v_\beta) \in (\{0,1\}^\kappa)^\beta$ and then it outputs $r \in \{0,1\}^\kappa$, $i \in \{1, \ldots, \beta\}$, and a tree $\mathcal{M}'$. Suppose that it happened that $\mathsf{MVrfy}^{\mathcal{H}^\kappa}(v_i, i, r, \mathcal{M}') = \mathsf{true}$. Then with probability at least $3\hat{t}^2 \cdot 2^{-\kappa}$* before *the algorithm $\mathcal{A}$ produced $\mathcal{M}'$ he made all the queries to the $\mathcal{H}^\kappa$ oracles that are needed by the verification algorithm $\mathsf{MVrfy}^{\mathcal{H}^\kappa}(v_i, i, p, \mathcal{M}')$ (in particular: he queried $H^\kappa_{\lambda_i}$ on v_i).*

Proof. Without loss of generality assume that $\mathcal{A}$ also queries the oracle on r. Now, suppose his execution was canonical and collision-free. This means that he also had to query the oracle on both children of the root of $\mathcal{M}_c^{\mathcal{H}^\kappa}$ (i.e. on $\mathcal{M}_c^{\mathcal{H}^\kappa}(0), \mathcal{M}_c^{\mathcal{H}^\kappa}(1)$, and then, recursively, on children of every node on the path from the root to λ_i. Hence in this case $\mathcal{A}$ made all the queries that are needed by the verification algorithm $\mathsf{MVrfy}^{\mathcal{H}^\kappa}(v_i, i, p, w)$. This finishes the proof, since, by Lemmas 4 and 5 the probability that the execution was either not canonical or not collision-free is at most $3\hat{t}^2 \cdot 2^{-\kappa}$. □

C.1 Proofs-of-Work

An example of a PoW scheme The PoW scheme used in Bitcoin is based on finding inputs for a hash function that produce an output starting with a certain number of zeros. We cannot use this PoW here, since the variance of the computational effort needed to solve it is to high: a lucky solver can solve the Bitcoin PoW much quicker than an unlucky one. Instead, we use a PoW based on the Merkle trees and a variant of the Fiat-Shamir transform [27]: a prover first constructs a Merkle tree with leaves depending on the challenge c, and then hashes (using some hash function G) the value r in the root of this tree to obtain indices of α leaves $\mu_1, \ldots, \mu_\alpha$ in the tree (for some parameter α). Finally, he sends r, the labels on the leaves $\mu_1, \ldots, \mu_\alpha$ together with the Merkle Proofs that the these leaves are correct. We later prove that a malicious prover that did not compute sufficiently large part of the Merkle tree cannot reply to all of these queries correctly with non-negligible probability. Similar techniques were already used in [15,24,3,40,41]

Prover $\mathsf{P}_t^{\kappa,\alpha}$:

On input $c \in \{0,1\}^\kappa$ the prover $\mathsf{P}_t^{\kappa,\alpha}$ executes the following steps.

1. Compute $r = \mathsf{MHash}^{\mathcal{H}^\kappa}(H_{\lambda_1}^\kappa(c), \ldots, H_{\lambda_\beta}^\kappa(c))$.
2. Let $(\mu_1, \ldots, \mu_\alpha) := G(r)$.
3. For every $i = 1$ to α compute $\mathcal{M}'_i := \mathsf{MProof}^{\mathcal{H}^\kappa}((\mu_1, \ldots, \mu_\alpha), i)$. For efficiency reasons we assume that the tree $\mathcal{M}_c^{\mathcal{H}^\kappa}$ was not erased after Step 1, and hence each invocation of $\mathsf{MProof}^{\mathcal{H}^\kappa}$ does not need to recompute it.
4. Output $(\mathcal{M}'_1, \ldots, \mathcal{M}'_\alpha, r)$.

Verifier $\mathsf{V}_t^{\kappa,\alpha}$:

On input $(c, (\mathcal{M}_1, \ldots, \mathcal{M}_\alpha, r))$ the verifier $\mathsf{V}_t^{\kappa,\alpha}$ does the following:

1. Let $(\mu_1, \ldots, \mu_\alpha) := G(r)$.
2. For every $i = 1$ to α run $\mathsf{MVrfy}(H_{\mu_i}^\kappa(c))$. If in all the cases the output is true then output true. Otherwise output false.

Fig. 3. A PoW scheme $(\mathsf{P}_t^{\kappa,\alpha}, \mathsf{V}_t^{\kappa,\alpha})$.

We now define our PoW scheme $(\mathsf{P}_t^{\kappa,\alpha}, \mathsf{V}_t^{\kappa,\alpha})$ that is secure in the $\mathcal{H}^\kappa$ model, for any natural parameters κ, α and t. Let $\beta := \lceil t/2 \rceil$ and let $\lambda_1, \ldots, \lambda_\beta$ be the leaves of an almost complete binary tree of size t. Suppose $G : \{0,1\}^\kappa \to \{1, \ldots, \beta\}^\alpha$ is a hash function modeled as a random oracle. Note that we do not count the calls to G in the hashrate of the devices. This is ok since the calls to $\mathcal{H}^\kappa$ will dominate. Our PoW scheme is presented on Figure 3. It is easy to see that the following holds.

Lemma 7. *The prover complexity of* $(\mathsf{P}_t^{\kappa,\alpha}, \mathsf{V}_t^{\kappa,\alpha})$ *is* t *and its verifier complexity is* $\alpha \cdot \lceil \log_2 t \rceil$.

Proof. The prover complexity of $(\mathsf{P}_t^{\kappa,\alpha}, \mathsf{V}_t^{\kappa,\alpha})$ is equal to the number of nodes of $\mathsf{Merkle}_{t,c}^\kappa$, and hence it is equal to t. The verifier complexity is equal to the number of nodes in each $\mathcal{M}_i'$ times α, and hence it is equal to $\alpha \cdot \lceil \log_2 t \rceil$. □

The security of $(\mathsf{P}_t^{\kappa,\alpha}, \mathsf{V}_t^{\kappa,\alpha})$ is proven in the following lemma.

Lemma 8. *The PoW scheme* $(\mathsf{P}_t^{\kappa,\alpha}, \mathsf{V}_t^{\kappa,\alpha})$ *is* $(\hat{t}_0, \hat{t}_1)$*-secure with error* $\hat{t}_1((\hat{t}_1+1)/t)^\alpha + (3\hat{t}_0^2 + 1) \cdot 2^{-\kappa}$.

Proof. Consider a malicious prover $\hat{\mathsf{P}}$ running in total time $\hat{t}_0$ and in time $\hat{t}_1$ after he received some $c \in \{0,1\}^\kappa$, and look at his calls to the random oracles during his execution. Consider the labeled tree M that $\hat{\mathsf{P}}$ sends to the verifier. Of course, every "reasonable" $\hat{\mathsf{P}}$ would check himself the conditions that the verifier checks in Step 2, as otherwise his probability of guessing the correct labels are very small. In the proof, however, we need to consider all possible strategies of $\hat{\mathsf{P}}$, and hence we also need to take into account the case when he behaves in an unreasonable way. To make it formal, we say that an execution of $\hat{\mathsf{P}}$ is *normal* if the tree T that he sends to V is such that during the execution $\hat{\mathsf{P}}$ issued all the oracle queries that V issues in Step 2. We now have the following.

Lemma 9. *If an execution of* $\hat{\mathsf{P}}$ *is not normal then the probability that* V *outputs* `true` *is at most* $2^{-\kappa}$.

Proof. If $\hat{\mathsf{P}}$ never issued a query that will be issued during the verification process, then the only thing he can do is try to guess it. Since the outputs of the random oracle are distributed uniformly, thus the probability that he guesses correctly is at most $2^{-\kappa}$. Hence Lemma 9 is proven. □

Lemma 10. *Assume the execution of* $\hat{\mathsf{P}}$ *on some* $c \in \{0,1\}^k$ *was canonical and collision-free (cf. Appendix C) and normal. Then the probability that* $\hat{\mathsf{P}}(c)$ *convinces the verifier* V *is at most* $\hat{t}_1((\hat{t}_1+1)/t)^\alpha$.

Proof. Let $\mathcal{Q}$ denote the set of queries of a type (L, R) that $\hat{\mathsf{P}}(c)$ ever made to the random oracle H_ϵ^κ (where ϵ is the root of the tree). Clearly $|\mathcal{Q}| \leq \hat{t}_1$. Fix some $q \in \mathcal{Q}$ and define recursively a labeled binary tree U (of depth at most $\lceil \log_2 t \rceil$) as follows:

- the root ϵ to U has a label $H^\kappa(q)$,
- for every node of U with a label w:

- if during the execution there was a call (L, R) to H^κ_λ whose outcome was w then add to U nodes $(\lambda||0)$ and $(\lambda||1)$ with labels L and R (resp.),
- if during the execution there was a call L to H^κ_λ whose outcome was w then add to U a nodes $(\lambda||0)$ with labels L.

Since we assumed that the execution is canonical and collision-free thus each value w appears at most once as an output of an oracle. Therefore the binary tree U is defined uniquely for every q. Let $\Sigma = \{\sigma_1, \ldots, \sigma_m\}$ denote the leaves of U. Since U is a binary tree, hence its total number of nodes is at least equal to $2 \cdot m - 1$. It is also easy to see, that, since the execution is canonical, thus all the calls to H^κ that were issued during the construction of U were made *after* $\hat{\mathsf{P}}$ learned c). Therefore the total number of nodes in U is at most $\hat{t}_1$, and thus we obtain:

$$m \leq (\hat{t}_1 + 1)/2 \tag{1}$$

The fixed query $q \in \mathcal{Q}$ also determines the input $H^\kappa(q)$ to the oracle G, and, in turn, G's output $(\lambda_1, \ldots, \lambda_\alpha) := G(H^\kappa(q))$. For $i = 1, \ldots, d$ let $\mathcal{X}_i$ denote the event that $\lambda_i \in \Sigma$. Since the execution is canonical Σ had to be chosen *before* the query q was sent to H^κ, and thus the choice of Σ is independent from $(\lambda_1, \ldots, \lambda_\alpha)$. Therefore the events $\mathcal{X}^q_1, \ldots, \mathcal{X}^q_\alpha$ are independent. Moreover, since the outputs of the random oracle are uniform, hence for every i the probability of $\mathcal{X}^q_i$ is equal to the cardinality m of $|\Sigma|$ divided by the total number $\lceil t/2 \rceil \geq t + 1$ of leaves in $\mathsf{Merkle}^\kappa_{t,c}$. Therefore, by (1), it is at most $(\hat{t}_1 + 1)/t$. Let $\mathcal{X}^q$ denote the event that for *every* i we have $\lambda_i \in \Sigma$, i.e. $\mathcal{X}^q := \wedge^\alpha_{i=1} \mathcal{X}^q_i$. From the independence of $\mathcal{X}^q_i$'s we get that

$$\mathbb{P}\left(\mathcal{X}^q\right) \leq ((\hat{t}_1 + 1)/t)^\alpha$$

Let $\mathcal{X}$ denote the sum of events $\mathcal{X}^q$ over all $q \in \mathcal{Q}$. There are clearly at most $\hat{t}_1$ such q's. Therefore, from the union bound we have $\mathbb{P}\left(\mathcal{X}\right) = \hat{t}_1((\hat{t}_1+1)/t)^\alpha$. If $\mathcal{X}$ did not happen then for every $q \in \mathcal{Q}$ the malicious prover does not know the label of at least one leaf $\lambda_i \in H^\kappa(q)$. Thus, since we assumed that the execution is normal, he cannot send any tree T to V that would convince him. Hence the total probability of $\hat{\mathsf{P}}$ succeeding is at most $\hat{t}_1((\hat{t}_1 + 1)/t)^\alpha$. This finishes the proof of Lemma 10. □

We now go back to the proof of Lemma 8. Since, by Lemmas 4 and 5 (in Appendix C) and Lemma 9 the total probability that an execution is either not normal or not collision-free or not cannonism is at most $(3\hat{t}^2+1) \cdot 2^{-\kappa}$. If this did not happen, then by Lemma 10 the malicious prover convinces V with probability at most $\hat{t}_1((\hat{t}_1 + 1)/t)^\alpha$. Altogether, this probability is bounded by $(3\hat{t}^2 + 1) \cdot 2^{-\kappa} + \hat{t}_1((\hat{t}_1 + 1)/t)^\alpha$. □

We now have the following.

Proof (of Lemma 1). Denote $(\mathsf{PTree}^\kappa_t, \mathsf{VTree}^\kappa_t) := (\mathsf{P}^{\kappa,\kappa}_t, \mathsf{V}^{\kappa,\kappa}_t)$ and use Lemma 8.

D Proof of Lemma 2

Proof. First observe that the honest parties have enough time to perform all the computations that the protocol requires them to perform. Clearly, this is true for the "challenges" and the "Proof of Work" phases. To see why it holds for the "key ranking

phase" assume for a while (it will be proved later), that each honest party sends at most $\lceil \pi_{\mathsf{max}}/\pi \rceil$ messages to each other party in this phase and therefore altogether there are at most θ messages delivered to each party in every round. Therefore each party has to compute at most θ times the V function, which altogether takes $\theta \cdot \mathsf{time}_{\mathsf{V}}$ steps, which take real time $(\theta \cdot \mathsf{time}_{\mathsf{V}})/\pi$. This is exactly the time we have given to every P to compute it.

The *key generation* property is satisfied trivially since the secret keys sk_P of the honest parties are never used in the protocol. It is also easy to see that if two parties P and P' are honest then the message $(\mathsf{Key}^0, \mathsf{pk}_P, A_P^{\ell+1}, \mathsf{Sol}_P)$ will always be accepted in round 0 of the "key ranking" phase, and hence the *validity* property holds.

To see why the *consistency* property holds assume that $k = \mathsf{rank}_P(\mathsf{pk}) < \ell$ for some pk. This means that P received a message $(\mathsf{Key}^k, \mathsf{pk}, B^{\ell+1-k}, \ldots, B^{\ell+1}, \mathsf{Sol})$ in k-th round of the "key ranking" phase, and she accepted this pk. Hence, she sent the message $(\mathsf{Key}^{k+1}, \mathsf{pk}, A_P^{\ell-k}, B^{\ell+1-k}, \ldots, B^{\ell+1}, \mathsf{Sol})$ to P', and it was received by P' in the $(k+1)$-st round of the key ranking phase. If P' already accepted pk in some earlier round then $\mathsf{rank}_{P'}(\mathsf{pk}) \leq k$, and hence we are done. Suppose it was not the case. Then P', in order to accept pk checks exactly the same conditions as P plus the condition that $c_{P'}^{\ell-(k+1)} \prec A_P^{\ell-\kappa}$. It is easy to see that this condition has to hold, since an honest P always adds $c_{P'}^{\ell-(\kappa+1)}$ (sent to him by an honest P' in the "challenges phase") to $A_P^{\ell-\kappa}$.

What remains is to prove the *bounded creation of identities* property. To simplify the calculations assume that $\mathsf{time}_{\mathsf{P}}/(\pi \cdot \Delta)$ and $(\theta \cdot \mathsf{time}_{\mathsf{V}})/(\pi \cdot \Delta)$ are integers, and hence producing the proof of work takes time $\mathsf{time}_{\mathsf{P}}/\pi$ and each step of the key ranking phase takes time $\Delta + \theta \cdot \mathsf{time}_{\mathsf{V}}/\pi$. Observe that each party P accepts a public key pk if it comes with a proof of work computed on $F(\mathsf{pk}, B^{\ell+1})$, and a sequence $B^{\ell+1-k}, \ldots, B^{\ell+1}$, such that for every $\ell + 1 - k \leq i \leq \ell$ we have $F(B^i) \prec B^{i+1}$. Because of this, clearly, $F(\mathsf{pk}, B^{\ell+1})$ is uniformly random to everybody (including the adversary) before a query $F(B^{\ell+1-k})$ is made to F. Party P also checks if $c_P^{\ell-k} \prec B^{\ell+1-k}$, where $c_P^{\ell-k}$ is P's challenge from the $(\ell-k)$th round. Since this challenge needs to be a function of P's' challenge c_P^0 from the 0th round, hence before the protocol started it was uniform from the point of view of the adversary. Hence, altogether, $F(\mathsf{pk}, B^{\ell+1})$ was uniform from $\mathcal{A}$'s point of view before the protocol started. Hence, for each pk the adversary had to invest some number of computing steps, let I denote the set of those pk's where he worked for at least $\xi \mathsf{time}_P$ steps, where

$$\xi := \frac{1 + \pi/(2\pi_{\mathcal{A}})}{1 + \pi/\pi_{\mathcal{A}}}.$$

Since ξ is a constant and is smaller than 1, thus (by Corollary 1) with overwhelming probability he will only manage to create to identities from the set I. Hence, what remains is to give a bound on $|I|$. Let us look at the total time T that the execution of the protocol takes. The "challenges phase" takes $(\ell+2)\Delta$ time. The "proof-of-work" phase takes $\mathsf{time}_{\mathsf{P}}/\pi$ time, and the "key ranking phase" takes $(\ell+1)(\Delta + (\theta \cdot \mathsf{time}_{\mathsf{V}})/\pi)$

time. Summing it up we obtain

$$
\begin{aligned}
T &\le (\ell+2)\Delta + \mathsf{time}_\mathsf{P}/\pi + (\ell+1)(\Delta + (\theta\cdot\mathsf{time}_\mathsf{V})/\pi) \\
&\le \mathsf{time}_\mathsf{P}/(\kappa^2\pi) + \mathsf{time}_\mathsf{P}/\pi + \mathsf{time}_\mathsf{P}/(\kappa^2\pi) + (\ell+1)\cdot\theta\cdot\mathsf{time}_\mathsf{V}/\pi && (2)\\
&= (1+2/\kappa^2)\cdot\mathsf{time}_\mathsf{P}/\pi + (\ell+2)\cdot\theta\cdot\mathsf{time}_\mathsf{V}/\pi \\
&\le (1+2/\kappa^2)\cdot\mathsf{time}_\mathsf{P}/\pi + (\mathsf{time}_\mathsf{P}/(\kappa^2\cdot\Delta\cdot\pi))\cdot\theta\cdot\mathsf{time}_\mathsf{V}/\pi && (3)\\
&= \frac{\mathsf{time}_\mathsf{P}}{\pi}\cdot\left(1+\frac{2+\theta\cdot\mathsf{time}_\mathsf{V}/(\Delta\cdot\pi)}{\kappa^2}\right) \\
&= \frac{\mathsf{time}_\mathsf{P}}{\pi}\cdot\Bigg(1+\underbrace{\frac{2}{\kappa^2}+\frac{\theta\cdot\lceil\log_2\mathsf{time}_\mathsf{P}\rceil}{\kappa\cdot\Delta\cdot\pi}}_{\epsilon(\kappa):=}\Bigg) && (4)\\
&= \frac{\mathsf{time}_\mathsf{P}}{\pi}\cdot(1+\epsilon(\kappa)), && (5)
\end{aligned}
$$

where (2) and (3) come from the fact that $\mathsf{time}_\mathsf{P} = \kappa^2\cdot(\ell+2)\cdot\Delta\cdot\pi$, and (4) comes from the fact that $\mathsf{time}_\mathsf{V} = \kappa\lceil\log_2\mathsf{time}_\mathsf{P}\rceil$. We get that

$$
\begin{aligned}
|I| &\le \frac{\mathsf{time}_\mathsf{P}\cdot\pi_\mathcal{A}}{\pi}\cdot(1+\epsilon(\kappa))/(\xi\cdot\mathsf{time}_\mathsf{P}) && (6)\\
&= \frac{\pi_\mathcal{A}}{\pi}\cdot(1+\epsilon(\kappa))\cdot\frac{1+\pi/\pi_\mathcal{A}}{1+\pi/(2\pi_\mathcal{A})} && (7)\\
&= \left(\frac{\pi_\mathcal{A}}{\pi}+1\right)\cdot\frac{1+\epsilon(\kappa)}{1+\pi/(2\pi_\mathcal{A})}. && (8)
\end{aligned}
$$

Since $\log\mathsf{time}_\mathsf{P} = 2\log\kappa + \log(\ell+2) + \log\pi$ thus $\epsilon(\kappa)\to 0$. Hence for sufficiently large κ the value of (8) is smaller than $\pi_\mathcal{A}/\pi+1$, and hence, since it is an integer, it has to be at most $\lceil\pi_\mathcal{A}/\pi\rceil$. □

E Communication and message complexity

Our main measure of communication complexity is based on the public channel. More precisely, we say that *an execution of a protocol Π has communication complexity γ* for a party P_i if the total length of the messages that the party P_i sends is γ. Similarly, the *communication complexity of the adversary $\mathcal{A}$ attacking a protocol Π* is the total length of the messages that $\mathcal{A}$ sends. We also define the *message complexity of an execution of a protocol Π for a party P_i* as the total number of messages the party P_i sends, and analogously the *message complexity of the adversary $\mathcal{A}$ attacking a protocol Π* as the number of messages that the adversary sends. The notion of a message complexity will be useful when we will be reasoning about the denial of service attacks, since arguably in many cases it is easier for an adversary to send say 1 message of size 1MB than 1 million messages of 1 byte size, e.g. if each message requires starting a new IP session. We explain this in more detail below.

F Computational complexity

We will also measure the running time of our algorithms in the standard complexity-theoretic way. In order to avoid confusion with the notion of the real time, we always use the term "time complexity" in this context. More precisely we say that a *an execution of an algorithm has time complexity* n if a Turing machine executes it in n steps. Each random oracle or ideal functionality call counts as one step. It will be important to assume that the adversary's computational complexity is poly-time, since otherwise he could query the random oracle H on all possible inputs before the protocol starts or break the underlying cryptographic primitives (like the signature schemes that we use frequently in our protocols).

G The bilateral communication model.

The reader may object that this way of measuring the communication complexity ignores the fact that sending messages over the public channel may be expensive, as the messages sent through it have to arrive to every party in the system. What sometimes might be more realistic is to measure the communication complexity by looking at messages sent directly between the parties. Such an approach would take into account differences between the costs of sending a message to one party and sending it to a large number of parties. We propose the following model for this. Each party P_i is able to send messages either through the public channel, or directly to some other party P_j. Since the set $\mathcal{P}$ of parties is unknown to P_i, hence in principle P_i is not able to address a message to any other party before it received a message from it . Therefore in our protocols we will always assume that P_i replies to a message of P_j that was earlier sent via the public channel. The security properties of the direct channel between P_i and P_j are exactly like the public channel, i.e., $\mathcal{A}$ can listen to it, insert his own messages and delay the messages by time at most Δ. Moreover $\mathcal{A}$ can create a "fake identity" P_k and hence provoke P_i to send messages to a non-existing P_k. It should be straightforward how the functionality $\mathcal{F}_{\mathsf{syn}}^{\pi,\pi_{\mathcal{A}},\theta}$ can be extended to cover this case.

We say that *an execution of a protocol Π has communication complexity γ for a party P_i in the bilateral model* if the total number of bits that the party P_i sends is γ. The *message complexity of an execution of a protocol Π for a party P_i* is the total number of messages that the party P_i sends. In all the cases above the messages sent over the public channel count $|\mathcal{P}|$ times. These notions extend naturally to the communication and message complexities of the adversary in the bilateral model.

H Communication and message complexity of the RankedKeys protocol

Communication and message complexity in the public channel model. Before we provide an efficiency analysis of the RankedKeys scheme let us optimize it a bit[10]. First

[10] The reason why the first version of the protocol was presented without these optimizations was that we think that would make the protocol harder to understand.

of all, observe that the only reason why $(\mathsf{Key}^k, \mathsf{pk}, A_P^{\ell-k}, B^{\ell+1-k}, \ldots, B^{\ell+1}, \mathsf{Sol})$ is sent by P at the end of the kth round of the "key ranking phase" is that each other party P' needs to be able to check in the the $(k+1)$st round that $cP'^{\ell-(k+1)} \prec A_P^{\ell-k}$ and $F(A_P^{\ell-k}) \prec B^{\ell+1-k}$, and $F(B^\ell) \prec B^{\ell+1}$ (for $i = k-1$ down to 0). An obvious way to optimize it is to use the Merkle Trees (see Section **??**) in the following way. Instead of using a hash function F we use MHash, and then instead of sending $A_P^{\ell-k}, B^{\ell+1-k}, \ldots, B^{\ell+1}$ we send $A_P^{\ell-\kappa}, \mathsf{MHash}\,(B^{\ell+1-k}), \ldots, \mathsf{MHash}(B^{\ell+1})$ together with the Merkle proofs that $F(A_P^{\ell-k})$ was used to compute the hash $\mathsf{MHash}(B^{\ell+1-k})$ and that $\mathsf{MHash}(B^{\ell-i})$ was used to compute $\mathsf{MHash}(B^{\ell+1-i})$ (for $i = k-1$ down to 0). These proofs can be checked efficiently using the MVrfy procedure. The security of such improved protocol easily follows from Lemma 6 (that states that no poly-time adversary can "fake" a Merkle proof with non-negligible probability). Since the length of each Merkle proof is at most logarithmic in the length of its input, hence the total length of every message sent during the "key ranking phase" is $O(\kappa(\theta + \ell \log \theta) + p)$, where p is the size of the proofs generated by Sol algorithm [11]. There are at most $\ell = \lceil \pi_{\mathsf{max}}/\pi \rceil$ identities created in the execution of the protocol (except a negligible probability), so each honest party sends at most ℓ messages in the „key ranking phase" and hence the communication complexity of each party in this phase is $O(\ell\kappa(\theta + \ell \log \theta) + \ell p)$. It is also easy to see that the communication complexity of each party in the "challenges phase" is $O(\ell\kappa)$, and in the "Proof of Work" phase it is $O(\theta\kappa + p)$. Therefore the total communication complexity of each party is $O(\ell\kappa(\theta + \ell \log \theta) + \ell p)$. Each honest party sends $\ell + 2$ messages in the „challenge phase", exactly one message in the „Proof of Work" phase and at most ℓ messages in the „key ranking phase", so the total message complexity of every party is equal to $2\ell + 3$.

Communication and message complexity in the bilateral channels model. Recall that in the bilateral channels model we assume unreliable channels between the parties and measure the communication and message complexities by counting the total number of respectively bits and messages sent over all channels. Of course, every protocol secure in the public key model can be run also in the bilateral model, by telling each party to send through the bilateral channels all the messages that normally she would send over $\mathcal{C}$. This, however, would result in the communication complexity multiplied by n (since each channel counts now separately). Fortunately, in case of our protocol we can do something more clever. Note that in the "key ranking phase" the goal of sending the $A_P^{\ell-k}$ vectors is to allow each party to check that here challenge (that is $c_{P'}^{\ell-(k+1)}$) was used to compute $c_P^{\ell-k} = F(A_P^{\ell-k})$. In the bilateral channels model we can modify this by using once again the Merkle trees: each P sends to each P' a Merkle hash $\mathsf{MHash}(A_P^{\ell-k})$ together with a proof that P''s challenge $c_{P'}^{\ell-(k+1)}$ was used to compute it. Later, P' can easily verify it using the MVrfy procedure. Hence, the communication complexity becomes $O(\kappa\ell^2 \log \theta + \ell p)$. Obviously, the message complexity is $O(n\ell)$.

[11] $A_P^{\ell-k}$ has a length $O(\kappa\theta)$ and we have $O(\ell)$ Merkle proofs of length $O(\kappa \log \theta)$ each

I Proof of Lemma 3

We start by showing that the informal properties of the protocol (i.e, "consistency", "validity" and the "bounded creation of inputs"), described in Section 3, hold.

The "validity" property follows simply from the description of the protocol. Each honest party P receives in the first round the message $(\mathsf{sid}, v, \mathsf{pk}_D, \mathsf{Sign}_{\mathsf{pk}_D}(v, \mathsf{pk}_D))$, so she sets $\mathcal{Z}_P^{\mathsf{pk}_D} = \{v\}$, where v is the value the dealer wants to broadcast. Note that an honest party will never accept a message with with the same pk_D and different v, because the value has to be signed by the dealer, what implies that at the end of the protocol $\mathcal{Z}_P^{\mathsf{pk}} = \{v\}$ for each honest P and hence for each honest P we have $v \in \mathcal{Y}_P$.

To prove the "consistency" consider for a moment a slightly simplified version of the Broadcast protocol, namely one with the condition $|\mathcal{Z}_P^{\mathsf{pk}_D}| < 2$ omitted. We will now prove that in this simplified version at the end of the protocol it holds that $\mathcal{Z}_P^{\mathsf{pk}_D} = \mathcal{Z}_{P'}^{\mathsf{pk}_D}$ for any two honest P and P' at the end of the protocol. To this end consider an arbitrary $v \in \mathcal{Z}_P^{\mathsf{pk}_D}$. We will prove that $v \in \mathcal{Z}_{P'}^{\mathsf{pk}_D}$. Suppose that P added v to $\mathcal{Z}_P^{\mathsf{pk}_D}$ during kth round. Consider two cases:

- $k < \ell$. It means that P sent a message

  $$(\mathsf{sid}, v, \mathsf{pk}_D, \mathsf{Sign}_{\mathsf{pk}_{a_1}}(v, \mathsf{pk}_D), \ldots, \mathsf{Sign}_{\mathsf{pk}_{a_k}}(v, \mathsf{pk}_D), \mathsf{Sign}_{\mathsf{pk}_i}(v, \mathsf{pk}_D))$$

  to P' at the end of k-th round and it was received by P' in the $(k+1)$-st round. P' will accept this message as all conditions are satisfied (or already $v \in \mathcal{Z}_{P'}^{\mathsf{pk}_D}$), namely:
  - Conditions (41) and (42) are trivially satisfied.
  - It holds that $\mathsf{rank}_P(a_q) \leq k$ for any $1 \leq q \leq k$, because otherwise P would have not accepted the message containing the value v in the k-th round. Therefore, from the definition of ranked key sets we have that $\mathsf{rank}_P(a_q) \leq k+1$ for any $1 \leq q \leq k$. Moreover, $\mathsf{rank}_{P'}(\mathsf{pk}_P) = 0$, because P is honest. Hence, condition (43) is satisfied.
  - If the condition (44), namely $v \notin \mathcal{Z}_{P'}^{\mathsf{pk}_D}$ is not satisfied than we already have that $v \in \mathcal{Z}_{P'}^{\mathsf{pk}_D}$.

  Hence, either P' accepted this message and added v to $\mathcal{Z}_{P'}^{\mathsf{pk}_D}$ at $(k+1)$-st round or v was already in $\mathcal{Z}_{P'}^{\mathsf{pk}_D}$
- $k = \ell$. It means that P received a message with a signatures of ℓ different parties on the value v. Obviously we can assume that at least one party is honest (as otherwise $n = 0$ and the protocol is secure trivially). Since there are at most $\ell = \lceil \pi_{\mathsf{max}}/\pi \rceil$ identities created by the RankedKeys protocol, thus at least one of these signatures comes from an honest party. Therefore, this honest party added this value to her set in one of the previous rounds and sent a message with the value v to all other parties. Note that messages sent by honest parties are always accepted by other honest parties (cf. analysis from the previous point), so the message with this value was accepted by all honest parties.

Hence, in both cases it is true that $v \in P'$ and therefore at the end of the protocol each party P has the same value of the set $\mathcal{Z}_P^{\mathsf{pk}_D}$ and outputs the same value.

The condition $v \in \mathcal{Z}_P^{\mathsf{pk}_D}$ is added to the Broadcast protocol merely for the efficiency reasons, as it puts down the number of the messages exchanges by the parties down to $O(n^2)$. Assume that at some point of the protocol it holds that $\mathcal{Z}_P^{\mathsf{pk}_D} = \{a, b\}$ for some honest P and some a, b. Hence, P_i does not need to broadcast any messages with values other than a and b, because it is already certain that during the execution of the protocol each honest P' will accept the message with values a and b unless it already had accepted some other two values. Therefore, in this case for each honest party P we have $|\mathcal{Z}_P^{\mathsf{pk}_D}| \neq 1$.

The "bounded creation of inputs" property comes from the "bounded creation of identities" property of the RankedKeys protocol (as clearly the size of each $\mathcal{Y}_P$ cannot be larger than P's set $\mathcal{K}_P$).

Description of the simulator. The description of the simulator $\mathcal{S}$ is fairly simple. It starts the adversary $\mathcal{A}$ and forwards all the communication from $\mathcal{E}$ to $\mathcal{A}$ and backwards. Since he receives from the $\mathsf{Broadcast}^{\kappa}$ functionality the list of inputs for the parties he can simulate the execution of the protocol himself. He observes the calls from $\mathcal{A}$ and the parties to the ideal functionality $\mathcal{F}_{\mathsf{syn}}^{\pi,\pi_{\mathcal{A}},\theta}$ and to the other random oracles used but the Proof-of-Work scheme. Thanks to this, he can easily "predict" what will be the output of the honest parties, or, in other words which inputs x will be accepted (on top of the inputs provided by the honest parties). If he realizes that for some corrupt party P (with PID pid) her input x_P will not be accepted by the honest party he issues a request $(\mathsf{Remove}, \mathsf{sid}, \mathsf{pid})$. He also determines the set of inputs x that do not come from the honest parties but will be accepted by them. For each such x he issues a $(\mathsf{Broadcast}, \mathsf{sid}, x)$ query. He then outputs what $\mathcal{A}$ outputs. Now, from the "validity" we obtain that the inputs of the honest parties will be included in $\mathcal{Y}$, from "consistency" we get that the outputs are the same for every party, and from the "bounded creation of inputs" it follows that the number of $(\mathsf{Broadcast}, \mathsf{sid}, x)$ queries is at most $\lceil \pi_{\mathcal{A}}/\pi \rceil$. □

J Unpredictable beacon generation

Generating an unpredictable beacon using the Broadcast protcool is done in the following straightforward way. First, each party P_i draws at random a string $s_i \leftarrow \{0,1\}^{\kappa}$. Then, the parties $P_1, \ldots, P_n$ execute the $\mathsf{Broadcast}^{\kappa}$ protocol with inputs $s_1, \ldots, s_n$ respectively. Let $s'_1, \ldots, s'_{m'}$ be the result. The parties compute the value of the beacon as a hash $H(s'_1, \ldots, s'_{m'})$. Note that obviously, the adversary can influence the result of this computation by adding some s_i that he controls. He can even do it after he learns the inputs of all the other honest parties. However, it is easy to see that if H is modeled as a random oracle, then a poly-time limited adversary cannot make $H(s'_1, \ldots, s'_{m'})$ equal to some value chosen by him in advance (except with negligible probability). Hence, the value of the beacon is unpredictable. This is, of course, a weaker property than the uniformity, but it still suffices for several applications (e.g. it is ok to use this value as a genesis block for a new cryptocurrency).

K Provably secure cryptocurrencies?

It might be tempting to say that the honest majority created for the MPC protocols can be used to construct a new cryptocurrency. In the simplest case, the parties selected by this procedure would simply emulate a trusted ledger, but also more general functionalities could be emulated. This would have the following advantages over the blockchain-based systems: (1) possibly quicker transaction confirmation times (no need to wait for new blocks), (2) security proof (which is especially important given the recent attacks on Bitcoin described in the introduction). We think it is an interesting option to explore, but we leave it as future work, since to fully solve this problem a good economic model for the cryptocurrencies is needed (and we are not aware of such a model). Below we only state some simple ideas and observations that can be used in such constructions.

One way of implementing a new currency using our methods would be as follows. Assume that the "honest majority group" is selected once a day. The parties that constitute this group are responsible for recording all the transactions. Independent of this, they also participate in a process of selecting of a new group for the next day. Once such a group is selected it identifies itself with a public key (whose corresponding private key is shared over all parties). Then the parties from the previous group sign a statement that the "passes" the control over the currency to the new group.

One thing that we need to consider here is the public verifiability of the history of transactions. Currently Bitcoin is designed in such a way that every new user can decide himself which chain is the proper one (assuming he knows the genesis block). The protocols that we propose in this paper provide assurance of correctness only to the players that were active during the execution of these protocols. This problem could be dealt with in the following way: a new user of the system waits for 1 day before deciding which group to trust (so that he can be sure that the "majority group" was selected honestly). The reader may object that in this case the adversary can break the system if he gets control over large computing power for a short period of time only. We want to stress that Bitcoin also suffers from this problem (as pointed out in [18]).