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* [24,14] 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) [30,20,11,7], 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 [8]). Despite of this, the MPCs unfortunately still remain out of scope of interest

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

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" [26]. 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 [26] 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* [15] 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) [16,17].

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 [19,5] 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 full version of this paper [1] 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* [27] 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 [27,18], voting schemes [25], or zero-knowledge protocols [3,21]. 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 [12] (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 the full version of this paper [1].

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 [23] 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. On the other hand, their work covers also the "sequential puzzles" (see [23]), while in this work we focus on parallelizable puzzles.

[2] The reader might 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 [9].

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. [17,26,4]), we establish the lower bounds on computational difficulty by counting the number of times a given algorithm calls some random oracle H [6]. 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 [29]. 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 [22]. 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 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 [28]). 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 RankedKeys (Figure 3, 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 In the full version of this paper [1] we define and analyze the communication complexity of our protocols. We also analyze their computational complexity. We also extend our model to cover the case of non-authenticated bilateral channels.

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 [9]. 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

[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.

[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.

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 $|\mathcal{P}| \cdot \pi + \pi_{\mathcal{A}} \leq \pi_{\mathsf{max}}$, 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 [9], but let us only say that it is done via sending messages (if one party sends a message to 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 on Fig. 1) 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 [9] (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 [9] 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

[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).

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.

Fig. 1. Functionality $\mathcal{F}_{\mathsf{syn}}^{\pi,\pi_{\mathcal{A}},\theta}$.

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)

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.

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 of 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 [9] 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 Fig. 2. The formal definition is given below.

$\mathcal{F}_{\mathsf{syn}}^{T}$ 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.

Fig. 2. Functionality $\mathcal{F}_{\mathsf{bc}}^{T}$, where T is the bound on the number of "fake identities" that the adversary can create. Our security definition requires that $T = \lceil \pi_{\mathcal{A}}/\pi \rceil$.

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 Sect. 3. In our protocols the computational effort will be verified using so-called Proofs of Work. A *Proof-of-Work (PoW)* scheme [16], 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 $\mathsf{V}(c, \mathsf{P}(c)) = \mathsf{true}$.

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 the full version of this paper [1].

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 [13]. 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

[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.

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" 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 3. 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 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 the full version of this paper [1]. 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

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. 3. The RankedKeys protocol.

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, then 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 $c_P^{\ell-k}$, 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 Lemma 1), 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 the full version of this paper [1]).

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 the full version of this paper [1].

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 [14] (and is similar to the technique used to construct broadcast from a proxcast protocol [13]). The protocol is depicted on Figure 4 and it works as follows. First the parties execute the RankedKeys protocol with parameters π, π_{max} and θ, built

[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.

1. Each party P takes as input $(\mathsf{Broadcast}, \mathsf{sid}, x_P)$.
2. The parties run the $\mathsf{RankedKeys}$ 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 $(\mathsf{Received}, \mathsf{sid}, \mathcal{Y}_P)$.

Fig. 4. The Broadcast protocol.

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 the full version of this paper [1].

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

5 Applications

Multiparty computations. As already mentioned before, the 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 [10]. In case we require that the protocol has guaranteed termination we need an assumption that the majority of the participants is honest [20], 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 in the full version of this paper [1], where we also discuss also the possibility of creating provable secure currencies using our techniques.

References

1. M. Andrychowicz and S. Dziembowski. Distributed cryptography based on the proofs of work. Cryptology ePrint Archive, Report 2014/796, 2014.
2. J. Aspnes, C. Jackson, and A. Krishnamurthy. Exposing computationally-challenged byzantine impostors. *Department of CS, Yale University, Tech. Rep*, 2005.
3. L. Babai. Trading group theory for randomness. In *STOC*, 1985.
4. A. Back. Hashcash - a denial of service counter-measure, 2002. technical report.
5. L. Bahack. Theoretical bitcoin attacks with less than half of the computational power (draft). *arXiv preprint arXiv:1312.7013*, 2013.
6. M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *ACM CCS*, 1993.
7. M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *STOC*, 1988.
8. P. Bogetoft, D. Lund Christensen, I. Damgard, M. Geisler, T. Jakobsen, M. Krøigaard, J. Dam Nielsen, J. Buus Nielsen, K. Nielsen, J. Pagter, M. Schwartzbach, and T. Toft. Secure multiparty computation goes live. In *Financial Cryptography*, 2009.

9. R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, 2001.
10. R. Canetti, Y. Lindell, R. Ostrovsky, and A. Sahai. Universally composable two-party and multi-party secure computation. In *STOC*, 2002.
11. D. Chaum, C. Crépeau, and I. Damgård. Multiparty unconditionally secure protocols (extended abstract). In *STOC*, 1988.
12. J. Clark and U. Hengartner. On the use of financial data as a random beacon. In *the International Conference on Electronic Voting Technology/Workshop on Trustworthy Elections*, 2010.
13. J. Considine, M. Fitzi, M. K. Franklin, L. A. Levin, U. M. Maurer, and D. Metcalf. Byzantine agreement given partial broadcast. *Journal of Cryptology*, 18(3):191–217, July 2005.
14. D. Dolev and H. R. Strong. Authenticated algorithms for byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983.
15. J. R. Douceur. The sybil attack. In *the First International Workshop on Peer-to-Peer Systems*, 2002.
16. C. Dwork and M. Naor. Pricing via processing or combatting junk mail. In *CRYPTO*, 1992.
17. C. Dwork, M. Naor, and H. Wee. Pebbling and proofs of work. In *CRYPTO*, 2005.
18. S. Even, O. Goldreich, and A. Lempel. A randomized protocol for signing contracts. In *CRYPTO*, 1982.
19. I. Eyal and E. G. Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *Financial Cryptography*. 2014.
20. O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. STOC, 1987.
21. S. Goldwasser and M. Sipser. Private coins versus public coins in interactive proof systems. In *STOC*, 1986.
22. J. Katz, U. Maurer, B. Tackmann, and V. Zikas. Universally composable synchronous computation. In *TCC*, 2013.
23. J. Katz, A. Miller, and E. Shi. Pseudonymous secure computation from time-lock puzzles. Cryptology ePrint Archive, Report 2014/857, 2014.
24. L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.
25. T. Moran and M. Naor. Split-ballot voting: everlasting privacy with distributed trust. In *ACM CCS*, 2007.
26. S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008. Available at bitcoin.org/bitcoin.pdf.
27. M. O. Rabin. Transaction protection by beacons. *Journal of Computer and System Sciences*, 27(2):256 – 267, 1983.
28. Bitcoin Wiki. Denial of service (dos) attacks. en.bitcoin.it/wiki/Weaknesses , Accessed on 26.09.2014.
29. Bitcoin Wiki. Network. en.bitcoin.it/wiki/Network, Accessed on 26.09.2014.
30. A. Chi-Chih Yao. Protocols for secure computations (extended abstract). In *FOCS*, 1982.