Distributed Cryptography Based on the Proofs of Work*

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 (without any trusted setup) under the assumption that the adversary has limited computing power. We propose a formal model for this scenario and then we construct the following protocols working in it:

(i) a broadcast protocol 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 $1/2$),
(ii) a protocol for identifying a set of parties such that the majority of them is honest, and every honest party belongs to this set (this protocol works under the assumption that the majority of computing power is controlled by the honest parties).

Our broadcast protocol can be used to generate an unpredictable beacon (that can later serve, e.g., as a genesis block for a new cryptocurrency). The protocol from Point (ii) can be used to construct arbitrary multiparty computation protocols. Our main tool for checking 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). 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 at the end of the paper. We stress however that the goal of this paper is not to propose new cryptocurrencies or to analyze the existing ones.

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* [43,23] 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) [53,34,14,9], 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 [12]). Despite of this, the MPCs unfortunately still remain out of scope of interest 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" [48]. Other examples include the *Litecoin* [50], Peercoin [42], and dozens of other so-called "Altcoins" (see, e.g., [1] for a list of them). Although initially these currencies were used mostly by a limited group enthusiasts, they quickly gained noticeable attention among the general public, and their economic importance has been rapidly growing — the current capitalization of Bitcoin is around 5 billion USD, and the average number

* 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.

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

of transactions per day is well above 50.000. Admittedly, this currency is not yet widely accepted by the merchants, but this situation is likely to change in close future. Indeed, recently some major US companies like Amazon [38], Dish Network [31], eBay and PayPal [37] expressed their interest in adopting Bitcoin.

The cryptocurrencies, unlike the cryptographic payment systems (e.g. [15]), are "independent" currencies whose exchange rate fluctuates freely. They owe their popularity mostly to the fact that they have no central authority, and hence it is infeasible for anyone to take control over the system, "print" the money (to generate inflation), or shut the entire system down. The money is transferred directly between the parties — they do not have to trust any third party for this. 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 [48] 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 operated by the honest users (we write more on this below), 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 always 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. For example, instead of assuming a bound on the *number* of malicious parties one can specify a family of sets of potentially malicious parties (the so-called *adversary structures* [40]). 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.[1]

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 in a moment 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* [24] 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) [25,26].

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

[1] A mixed case when an honest party can leak some information to the adversary was also considered in a sequence of works on the leakage-resilient MPCs [13,22,10].

that assumes a trusted generation of an unpredictable "genesis block" (See Appendix A for more details). Besides of being of general interest, our work is motivated twofold. Firstly, recently discovered weaknesses of Bitcoin [29,7] 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. Secondly, we 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[2]. In fact, there are already some ongoing efforts to use the Bitcoin paradigm for purposes other than the cryptocurrencies. For example Garman et al. [33] propose to use the Bitcoin's PoW-based distributed consensus in order to create a decentralized system for anonymouns credentials, and Fromknecht et al. [18] present an idea of applying the same technology to create a decentralized PKI, also IBM recently announced an "Adept" system [39], 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. 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). We discuss this problems further in Section 7.3. The technical description of Bitcoin and some recent attacks on its PoW scheme appears in Appendix A.

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 3). Our model does not assume any trusted setup (in particular: we do not assume any trusted beacon generation). Then, in Section 6 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. 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.

1. We construct a broadcast protocol secure against any π_{max}, working in time linear in $\lceil \pi_{\mathsf{max}}/\pi \rceil$ (in Section 6.2). Using this protocol as a subroutine we later (in Section 7.2) construct a scheme for an unpredictable beacon generation.
2. Using the broadcast protocol from the previous point, we construct (in Section 6.3) a protocol for identifying a set $\mathcal{K}$ of parties such that the majority of them is honest, and every honest party belongs to this set. The protocol also works in time $\lceil \pi_{\mathsf{max}}/\pi \rceil$. It requires an assumption that $n \geq \lceil \pi_{\mathcal{A}}/\pi \rceil$. Moreover the parties in $\mathcal{K}$ know each other public keys. This allows them to execute any standard MPC protocol that works under the assumption that the majority of the participants in honest (we argue about it in Section 7.1).

[2] 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.

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 3.4. 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 7.

2 Preliminaries

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

Random oracle model. We model the hash functions as random oracles [8]. 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 B.

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).

3 Model

In this section we present our model for reasoning about computing power and the peer-to-peer protocols.

3.1 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. [26,48,6]), we establish the lower bounds on computational difficulty by counting the number of times a given algorithm calls some random oracle. We will use the random oracle family $\mathcal{H}^\kappa = \{H^\kappa_\lambda\}_{\lambda \in \Lambda}$ defined in Section 2, where κ is a security parameter. This will be called an $\mathcal{H}^\kappa$*-model*. Note that the input and output size of each H^κ_λ is fixed for every value of the parameter κ. Hence assuming that each invocation of such a function takes some fixed unit of time is realistic.

Our protocols are executed by a number of devices and attacked by one device controlled by an adversary $\mathcal{A}$, each running some code represented as a Turing machine. Everything happens in real time (see Section 3.2). 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 of queries to $\mathcal{H}^\kappa$ in time Δ. The parameter π is called the *hashrate of D (per time Δ in the $\mathcal{H}^k$-model)*. 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* in Section 3.2). 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 $\mathcal{H}^\kappa$'s would be the dominating factor during the Proofs of Work (Sec. 5.1). In particular: any other random oracles will be invoked a much smaller number of times than $\mathcal{H}^\kappa$. Note that, even if this number were comparable, one could still make $\mathcal{H}^\kappa$ evaluate much longer than any other hash function F, e.g., by defining $\mathcal{H}^\kappa$ to be equal to multiple iterations of F.

3.2 Multiparty protocols

Unlike in the traditional MPC settings, in our case the number of parties executing the protocol is not known in advance (even 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 whose code Π 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 and random coins. Another unusual property of our model is that there is no concept of "corrupting a party". Since we are in the peer-to-peer scenario without any trusted setup, thus the parties have no way to check the integrity of the messages. Hence, we can simply assume that the adversary is an external entity and give him full access to the communication channels (as we discuss below, in order to prevent the adversary from stopping the whole communication we will assume that he cannot block the messages sent between the honest parties). This is clearly as powerful as assuming

that some parties may be corrupt, but it makes the model cleaner. Let us stress that whenever we use the term "party" it means an *honest* party, since there are no "corrupt" parties in the system.

Formally, a *multiparty protocol (in the $\mathcal{H}^\kappa$-model)* is an a randomized algorithm described as an interactive Turing machine Π with access to the random oracles $\mathcal{H}^\kappa$ (see Sect. 3.1), and possibly some other random oracles. The algorithm Π will be run in n copies (for some parameter $n \in \mathbf{N}$) on devices $P_1, \ldots, P_n$ called the *(honest) parties*. Each device will have a hashrate π per time Δ in the $\mathcal{H}^\kappa$-model.[3] Each P_i gets as input enceher own identifier i and the security parameter 1^κ. The assumption that every party knows her identifier is used only in the "bilateral communication model" (see Section 3.3), where the parties need to attach identifiers to messages that they send. Equivalently we could simply assume that at the beginning of the protocol each party generates her identifier randomly (say: by drawing it uniformly from $\{0,1\}^\kappa$). Assumption that these identifiers are the consecutive natural number is stated only to simplify the exposition.

Moreover P_i can take an input $x_i \in \{0,1\}^*$ and produce an output $y_i \in \{0,1\}^*$. The protocol is attacked by an adversary $\mathcal{A}$ which is also a Turing machine that can query the same random oracles as the parties and is run on a device with hashrate $\pi_\mathcal{A}$. The number n will *not* be given as input to the honest parties, but it will be known by the adversary. In other words: the protocol should work in the same way for any n. On the other hand: each P_i 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. The running time of P_i can depend on these parameters. Note that $n \cdot \pi + \pi_\mathcal{A} \leq \pi_{\mathsf{max}}$, but this inequality may be sharp, and even $n \cdot \pi + \pi_\mathcal{A} \ll \pi_{\mathsf{max}}$ is possible, as, e.g., the adversary can use much less hashrate than the maximal amount that he is allowed to[4].

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 $\mathcal{C}$ which allows every party and the adversary to post a message on it. One can think of $\mathcal{C}$ as being implemented using some standard (cryptographically insecure) "network broadcast protocol" like the one in Bitcoin [52]. The contents of $\mathcal{C}$ 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).

We give to the adversary full access to $\mathcal{C}$: he learns (without any delay) every message that is sent through $\mathcal{C}$, and he can insert messages into it. The adversary may decide that the messages inserted into $\mathcal{C}$ 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). For simplicity we assume that every P_i that posts a message through $\mathcal{C}$ attaches his identifier i to it. Of course it should not be understood as any type of a cryptographically-strong message authentication (in particular: $\mathcal{A}$ can also post messages "in the name of P_i").

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 protocols would also be secure in this model[5]

[3] Note that we assume that all the honest devices have identical hashrate. 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 also 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 (e.g. in Section 6.3) to include also "weights" of the parties.

[4] 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, $n \cdot \pi > \pi_{\mathsf{max}}/2$.

[5] This is because one can think of δ as being "counted into" the time Δ that it takes for the messages to arrive to all receivers.

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 (introduced in Section 3.1), 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 call (to $\mathcal{H}^\kappa$ or to some other oracle) 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 oracles $\mathcal{H}^\kappa$ 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).

Communication and message complexity Our main measure of communication complexity is based on the public channel $\mathcal{C}$. We say that *an execution of a protocol Π has communication complexity γ* for a party P_i if the total number of bits that the party P_i sends is γ. Similarly, the *communication complexity of the adversary $\mathcal{A}$ attacking a protocol Π* is the total number bits that $\mathcal{A}$ sends over the channel. 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 in Section 3.4.

3.3 The bilateral communication model

The reader may object that this way of measuring the communication complexity ignores the fact that sending messages over $\mathcal{C}$ may be expensive, as the messages on $\mathcal{C}$ 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 $\mathcal{C}$, or directly to some other party P_j. Since the number n of parties is unknown to P_i, hence in principle j can be any natural number. Therefore in our protocols we will always assume that P_i replies to a message of P_j that was earlier sent through $\mathcal{C}$ (remember that we assumed that P_j's identifier j is attached to every such message, and hence it will be known to P_i). The security properties of the direct channel between P_i and P_j are exactly like $\mathcal{C}$, i.e., $\mathcal{A}$ can listen to it, insert his own messages and delay the messages by time at most Δ. Of course $\mathcal{A}$ can create a "fake identity" P_k and hence provoke P_i to send messages to a non-existing P_k.

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 $\mathcal{C}$ count n times (where n is the number of honest parties). These notions extend naturally to the communication and message complexities of the adversary in the bilateral model.

3.4 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.[6] 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[7]. 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 [51]). 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 14) 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 $\theta/\pi < 1$.

4 Security definitions

In this section we present the security definitions of our main constructions. We start with the broadcast protocol, which is defined as follows.

Definition 1. *Consider a multi-party protocol Π in the $\mathcal{H}^\kappa$-model. Let $(P_1, \ldots, P_n)$ denote the honest parties executing Π, each of them having a device with hashrate $\pi > 0$ per time Δ in the $\mathcal{H}^\kappa$-model. Each P_i takes as input $x_i \in \{0,1\}^\kappa$, and it produces as output a set $\mathcal{Y}_i \subset \{0,1\}^\kappa$. The protocol Π 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 (in the model from Section. 3.2) the following conditions hold (except with probability negligible in κ):*

Consistency: *All sets $\mathcal{Y}_i$ are equal, i.e.: $\mathcal{Y}_1 = \cdots = \mathcal{Y}_n$. Call this set $\mathcal{Y}$.*
Validity: *For every $i \in \{1, \ldots, n\}$ we have $x_i \in \mathcal{Y}$.*
Bounded creation of inputs: *The size of $\mathcal{Y}$ is at most $n + \lceil \pi_{\mathcal{A}}/\pi \rceil$.*

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 Section 6.2 (it is denoted RankedBroadcast).

The second main primitive that we construct is a protocol allowing the parties to agree on a set of parties, such that it is guaranteed that the majority of the parties in this set is honest. Formally, this will mean that the

[6] 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.

[7] 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.

parties output a set $\mathcal{K}$ of public keys such that n of these keys "belong" to the honest party (pk_i belongs to P_i if she knows the secret key sk_i corresponding to pk_i), and the set of the remaining keys is of size less than n. Obviously, this is possible to construct only if the majority of the computing power is honest, which will be formally expressed as $\lceil \pi_{\mathcal{A}}/\pi \rceil < n$. Each secret key sk_i corresponding to pk_i's needs to be secret after executing the protocol (also with respect to all the parties other than P_i). This is formalized by requiring the protocol is a secure key-generation protocol for some signature scheme Σ.

Definition 2. *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 $\mathcal{H}^\kappa$-model. Let $(P_1, \ldots, P_n)$ denote the honest parties executing Π, each of them having a device with hashrate π per time Δ in the $\mathcal{H}^\kappa$-model. Each P_i takes as input a security parameter 1^κ, and it produces as output a tuple $(\mathsf{sk}_i, \mathsf{pk}_i, \mathcal{K}_i)$, where $(\mathsf{sk}_i, \mathsf{pk}_i) \in \{0,1\}^* \times \{0,1\}^*$ is called a* (private key, public key) pair of P_i*, and the finite set $\mathcal{K}_i \subset \{0,1\}^*$ will be some set of public keys.*

The protocol Π is called an honest majority Σ-key generation protocol for Σ *if it terminates in finite time and for any poly-time adversary $\mathcal{A}$ whose device has hashrate $\pi_{\mathcal{A}}$ and who attacks this protocol (in the model from Section. 3.2) the following conditions hold, provided that $\lceil \pi_{\mathcal{A}}/\pi \rceil < n$:*

Key-generation: *Π is a key-generation algorithm for every P_i, by which we mean the following. First of all, for every $i = 1, \ldots, n$ and every $m \in \{0,1\}^*$ we have that $\mathsf{Vrfy}(\mathsf{pk}_i, \mathsf{Sign}(\mathsf{sk}_i, m)) = \mathsf{true}$. Moreover sk_i 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_i, and later $\mathcal{A}$ engages in the "chosen message attack" (see Sect. 2) against an oracle that signs messages with key sk_i. Then any such $\mathcal{A}$ has negligible (in κ) probability of forging a valid (under key pk_i) signature on a fresh message.*

Consistency *The sets $\mathcal{K}_i$ that were produced by the honest parties are identical. Let $\mathcal{K} := \mathcal{K}_1 (= K_2 = \cdots = \mathcal{K}_n)$.*

Validity: *For every i it is the case that $\mathsf{pk}_i \in \mathcal{K}$.*

Bounded generation of identities: *The size of $\mathcal{K}$ is at most $n + \lceil \pi_{\mathcal{A}}/\pi \rceil$ (hence the adversary "controls" at most $\lceil \pi_{\mathcal{A}}/\pi \rceil$ identities).*

5 Ingredients

Merkle trees. We use a standard cryptographic tool called the *Merkle trees* [44]. 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 $t = 2\beta$ (cf. Section 2). Let $\mathcal{H}^\kappa$ be a family of hash functions defined in Section 2, 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 5.1.

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 5.1) 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 5.1) 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 (the proof of this lemma appears in Appendix C).

Lemma 1. *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).*

5.1 Proofs-of-Work

A natural way to prevent the adversary from launching Sybil attacks is to require some computational work from each party in order to establish an identity. This is verified using so-called Proofs of Work. A *Proof-of-Work (PoW)* scheme [25] is a pair of randomized algorithms: a *prover* P and a verifier V, working in the $\mathcal{H}^\kappa$-model (cf. Section 3.1), where κ is a security parameter. 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 $\mathsf{V}(c, \mathsf{P}(c)) = \mathsf{true}$.

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

We say that a PoW (P, V) has *prover complexity* t *(in the $\mathcal{H}^\kappa$-model)* if on every input $c \in \{0,1\}^*$ the prover P makes at most t queries to the oracles in $\mathcal{H}^\kappa$. We say that (P, V) has *verifier complexity* t' *(in the $\mathcal{H}^\kappa$-model)* if for every $c, s \in \{0,1\}^*$ the verifier V makes at most t' queries to the oracles in $\mathcal{H}^\kappa$. 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 $\mathcal{H}^\kappa$ 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 $\mathcal{H}^\kappa$ 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 $\mathcal{H}^\kappa$-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 $\mathcal{H}^\kappa$ 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 in the $\mathcal{H}^\kappa$ model with error $\epsilon(\kappa)$.

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 [30]: 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 [17,27,4,45,46]

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 $r = \mathsf{MHash}^{\mathcal{H}^\kappa}(H^\kappa_{\lambda_1}(c), \ldots, H^\kappa_{\lambda_\beta}(c))$.
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 $\mathsf{MVrfy}(H^\kappa_{\mu_i}(c))$. If in all the cases the output is true then output true. Otherwise output false.

Fig. 2. 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 (cf. Section 3.1). Our PoW scheme is presented on Figure 2. It is easy to see that the following holds (the proof appears in Appendix D).

Lemma 2. *The prover complexity of $(\mathsf{P}_t^{\kappa,\alpha}, \mathsf{V}_t^{\kappa,\alpha})$ is t and its verifier complexity is $\alpha \cdot \lceil \log_2 t \rceil$.*

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

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

It will be sometimes convenient to use one security parameter κ instead of κ and α. Denote $(\mathsf{PTree}_t^\kappa, \mathsf{VTree}_t^\kappa) := (\mathsf{P}_t^{\kappa,\kappa}, \mathsf{V}_t^{\kappa,\kappa})$.The following fact can be easily derived from Lemma 3.

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

6 Constructions

We are now ready to present the constructions of the protocols specified in Section 4. Our protocols will be based on the PoW described in Section 5.1. 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 [3], 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. 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.

It is relatively easy to see that nevertheless the above protocol can be used to achieve Byzantine agreement (and hence the broadcast protocol), under the assumption that the honest parties computing power is more than $(2/3) \cdot \pi_{\mathsf{max}}$, where π_{max} is a total bound on the computing power of all the participants (honest parties and the adversary). This is because in this case it is guaranteed that there exists a set $\mathcal{P}'$ of identities such that all of them are controlled by the honest parties, and $|\mathcal{P}|$ is of size at least $2/3$ of the total size of identities accepted by the honest parties. Therefore the parties could execute a classical Byzantine agreement protocol, which is secure if more than $2/3$ of the parties are honest (each party will simply ignore the messages signed by keys that she does not recognize).

It is a natural question whether we can achieve Byzantine agreement and broadcast without this assumption. In the rest of this section we answer this question affirmatively. Our presentation is organized as follows. First, in Section 6.1 we show a protocol that we call the "ranked key generation protocol". Then we show how to use it to construct a broadcast protocol (in Section 6.2). Finally, in Section 6.3 we show how to use these protocols to construct a protocol for identifying a group of parties with honest majority.

6.1 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 protocol bares some similarities with the "proxcast" protocol of Considine et al [19]. The formal definition follows.

Definition 3. *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 $\mathcal{H}^\kappa$-model. Let $(P_1, \ldots, P_n)$ denote the honest parties executing Π, each of them having a device with hashrate π per time Δ in the $\mathcal{H}^\kappa$-model. Each P_i takes as input a security parameter 1^κ, and it produces as output a tuple $(\mathsf{sk}_i, \mathsf{pk}_i, \mathcal{K}_i, \mathsf{rank}_i)$, where $(\mathsf{sk}_i, \mathsf{pk}_i) \in \{0,1\}^* \times \{0,1\}^*$ is called a* (private key, public key) pair of P_i*, the finite set $\mathcal{K}_i \subset \{0,1\}^*$ will be some set of public keys, and $\mathsf{rank}_i : \mathcal{K}_i \to \{0, \ldots, \ell\}$ will be called a* key-ranking function (of P_i)*. We will say that an* identity pk was created during the execution Π *if $\mathsf{pk} \in \mathcal{K}_i$ for at least one honest P_i (regardless of the value of $\mathsf{rank}_i(\mathsf{pk})$).*

*The protocol Π is called a $\pi_{\mathcal{A}}$-*secure ℓ-ranked Σ-key generation protocol *if for any poly-time adversary $\mathcal{A}$ whose device has hashrate $\pi_{\mathcal{A}}$ and who attacks this protocol (in the model from Section. 3.2) the following conditions hold:*

Key-generation: *Π is a key-generation algorithm for every P_i in the same sense as in Definition 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.*

Validity: *for every i it is the case that $\{\mathsf{pk}_1, \ldots, \mathsf{pk}_n\} \subseteq \mathcal{K}_i$ and for every $j \in \{1, \ldots, n\}$ we have that $\mathsf{rank}_i(\mathsf{pk}_j) = 0$,*

Consistency: *for every $i \in \{1, \ldots, n\}$ and every $k \in \mathcal{K}_i$ if $\mathsf{rank}_i(k) < \ell$ then for every $j \in \{1, \ldots, n\}$ we have that $k \in \mathcal{K}_j$ and $\mathsf{rank}_j(k) \leq \mathsf{rank}_i(k) + 1$.*

Our construction of a ranked key generation protocol RankedKeys is presented on Figure 3. The protocol RankedKeys work 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. It uses a Proof of Work scheme (P, V) with prover time time_P and verifier time time_V. We will later instantiate this PoW scheme with the scheme $(\mathsf{PTree}, \mathsf{VTree})$ from Section 5.1. The parameter ℓ will be equal to $\lceil \pi_{\mathsf{max}}/\pi \rceil$.

Let us present some intuitions behind our protocol. First, recall that the problem with the protocol from [3] (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 with a delay at most Δ are assigned rank 0, keys received in time at most 2Δ 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 he 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, each of them taking time Δ. At the beginning of the first round each P_i generates his challenge c_i^0 randomly and sends it to all the other parties. Then, in each kth round each P_i collects the messages $a_1, \ldots, a_m$ sent to him in the previous round, concatenates then into $A_i^k = (a_1, \ldots, a_m)$, hashes them, and sends the result $c_i^k = F(A_i^k)$ to all the other parties.

The challenges phase

This phase consists of $\ell + 2$ rounds, each lasting exactly one interval Δ of real time:

- *Round* 0*:* Each party P_i draws a random challenge $c_i \leftarrow \{0,1\}^\kappa$ and sends his *challenge message of level* 0 equal to $(\mathsf{Challenge}^0, c_i^0)$ to all parties (including herself).
- For $k = 1$ to $\ell + 1$ in *round* k each party P_i 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_i. Denote $A_i^k = (a_1, \ldots, a_m)$. Then P_i computes her challenge in round k as $c_i^k = F(A_i^k)$ and sends $(\mathsf{Challenge}^k, c_i^k)$ to all parties (this is not needed in the last rounds, i.e., when $k = \ell + 1$).

The Proof of Work phase

This phase takes real time $\mathsf{time}_\mathsf{P}/\pi$. Each party P_i performs the following.

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

The key ranking phase

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

- *Round 0*: Each party P_i waits for time Δ for message of the form $(\mathsf{Key}^0, \mathsf{pk}, B^{\ell+1}, \mathsf{Sol})$ received in the "proof-of-work". 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_i^ℓ appears in $B^{\ell+1}$, i.e., $c_i^\ell \prec B^{\ell+1}$.

  If both of these conditions hold then P_i accepts the key pk with rank 0, i.e., P_i adds pk to the set $\mathcal{K}_i$ and sets $\mathsf{rank}_i(\mathsf{pk}) := 0$. Moreover P_i notifies all the other parties about this fact by sending to every other party a message $(\mathsf{Key}^1, \mathsf{pk}, A_i^\ell, B^{\ell+1}, \mathsf{Sol})$.
- For $k = 1$ to ℓ in *round* k each party P_i does the following. It waits for messages of a form $(\mathsf{Key}^k, \mathsf{pk}, B^{\ell+1-k}, \ldots, B^{\ell+1}, \mathsf{Sol})$. After time Δ passed P_i stops listening and for each received message she checks the following conditions:
  - the key pk has not been yet added to $\mathcal{K}_i$, i.e.: $\mathsf{pk} \notin \mathcal{K}_i$
  - Sol is a correct PoW solution for the challenge $F(\mathsf{pk}, B^{\ell+1})$, i.e., if $\mathsf{V}(F(\mathsf{pk}, B^{\ell+1}), \mathsf{Sol}) = \mathsf{true}$,
  - $c_i^{\ell-k} \prec B^{\ell+1-k}$ 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_i accepts the key pk with rank k, i.e., P_i adds pk to the set $\mathcal{K}_i$ and sets $\mathsf{rank}_i(\mathsf{pk}) := k$. Moreover if $k < \ell$ then P_i notifies all the other parties about this fact by sending at the end of the round to every other party a message $(\mathsf{Key}^{k+1}, \mathsf{pk}, A_i^{\ell-k}, B^{\ell+1-k}, \ldots, B^{\ell+1}, \mathsf{Sol})$ (recall that A_i^k is equal to the set of challenges received by P_i in the kth round of the "challenges phase", and $F(A_i^k) = c_i^k$).

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

Fig. 3. The RankedKeys protocol.

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_i^k depends on c_j^{k-1} for any honest P_i, P_j [8] and $1 \leq k \leq \ell$ and hence $F(A_j^k)$ depends on c_i for any honest P_i, P_j and an arbitrary $1 \leq k \leq \ell + 1$.

Then, during the "Proof of Work" phase each honest party P_i draws a random key pair $(\mathsf{sk}_i, \mathsf{pk}_i)$ and creates a proof of work[9] $\mathsf{P}(F(\mathsf{pk}_i, A_i^{\ell+1}))$. 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_i requires that she receives it in the kth round in this phase and that it is accompanied with a proof $\mathsf{P}(F(\mathsf{pk}_i||s))$ for some string s, which depends on $c_i^{\ell-k}$. Such a proof could not been precomputed, because $c_i^{\ell-k}$ depends on c_i^0, which was drawn randomly by P_i 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_i 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 $c_i^{\ell-k}$, which depends on $c_j^{\ell-(k+1)}$ for any honest P_j. In the effect, all other honest parties, which have not yet assigned pk a rank will assign it a rank $k+1$.

Let $\mathsf{RankedKeys}^\kappa$ denote the $\mathsf{RankedKeys}$ scheme instantiated with the PoW scheme $(\mathsf{PTree}^\kappa_{\mathsf{time_P}}, \mathsf{VTree}^\kappa_{\mathsf{time_P}})$ (from Section 5.1), where $\mathsf{time_P} := \kappa^2 \cdot (\ell+2)\Delta \cdot \pi$. Note that therefore $\mathsf{time_V} := \kappa \lceil \log_2 \mathsf{time_P} \rceil$. We now have the following fact (its proof appears in Appendix F).

Lemma 4. *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 interval Δ (where n is the number of honest parties). Then the $\mathsf{RankedKeys}^\kappa$ protocol is a $\pi_{\mathcal{A}}$-secure ℓ-ranked key generation protocol, for $\ell = \lceil \pi_{\mathsf{max}}/\pi \rceil$, whose total execution takes real time $\Delta(2\ell+3) + \mathsf{time_P}/\pi + (\ell+1)(\theta \cdot \mathsf{time_V})/\pi$.*

Communication and message complexity of the $\mathsf{RankedKeys}^\kappa$ protocol is analysed in Appendix G.

6.2 The RankedBroadcast 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 $\mathsf{RankedBroadcast}^\kappa$, works in time that is linear in $\ell = \lceil \pi_{\mathsf{max}}/\pi \rceil$ plus the execution time of $\mathsf{RankedKeys}^\kappa$. It is based on a classical authenticated Byzantine agreement by Dolev and Strong [23] (and is similar to the technique used to construct broadcast from a proxcast protocol [19]). The protocol works as follows. First the parties execute the $\mathsf{RankedKeys}^\kappa$ protocol with parameters π, π_{max} and θ, built on top of a signature scheme $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$. For convenience assume that every signature σ contains information identifying the public key that was used to compute it. Let $(\mathsf{sk}_i, \mathsf{pk}_i, \mathcal{K}_i, \mathsf{rank}_i)$ be the output of each P_i after this protocol ends (recall that $(\mathsf{sk}_i, \mathsf{pk}_i)$ is her key pair, $\mathcal{K}_i$ is the set of public keys that she accepted, rank_i is the key ranking function). Then, each party P_d executes the procedure $\mathsf{RankedBroadcast}^\kappa_d$ depicted on Fig. 4 (this happens in parallel for every d). During the execution each party P_i maintains a set $\mathcal{Z}_i^d$ initialized with $\emptyset$. The output of each party is equal to the only elements of this set (if $\mathcal{Z}_i^d$ is a singleton) or $\perp$ otherwise.
The security of this protocol is proven in the following lemma.

Lemma 5. *The $\mathsf{RankedBroadcast}^\kappa$ protocol is a π_{max}-secure broadcast protocol.*

Proof. We consider the execution of each $\mathsf{RankedBroadcast}^\kappa_d$ separately. The "validity" property follows simply from the description of the protocol. Each honest party P_i receives in the first round the message

[8] This is because $c_j^{k-1} \prec A_i^k$ and $F(A_i^k) \prec F(A_i^k) = c_i^k$.

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

The protocol consists of $\ell + 1$ rounds, each lasting one interval Δ:

- *Round 0:* The dealer P_d sends to every other party the message $(x_d, \mathsf{Sign}_{\mathsf{sk}_d}(x_d, \mathsf{pk}_d))$, where x_d is his input that he wants to broadcast.
- *Round k, for $1 \leq k \leq \ell$:* Each party except of the dealer P_d waits for the messages of the form $(v, \mathsf{Sign}_{\mathsf{pk}_{a_1}}(v, \mathsf{pk}_d), \ldots, \mathsf{Sign}_{\mathsf{pk}_{a_k}}(v, \mathsf{pk}_d))$. Such a message is accepted by P_i if:
  (1) all signatures are valid and are corresponding to different public keys,
  (2) $\mathsf{pk}_{a_1} = \mathsf{pk}_d$,
  (3) $\mathsf{pk}_{a_j} \in K_i$ and $\mathsf{rank}_i(\mathsf{pk}_{a_j}) \leq k$ for $1 \leq j \leq k$, and
  (4) $v \notin \mathcal{Z}_i^d$ and $|\mathcal{Z}_i^d| < 2$.
  If a message is accepted then P_i adds v to her set $\mathcal{Z}_i^d$ and if $k < \ell$ she sends a message $(v, \mathsf{Sign}_{\mathsf{pk}_{a_1}}(v, \mathsf{pk}_d), \ldots, \mathsf{Sign}_{a_k}(v, \mathsf{pk}_d), \mathsf{Sign}_{\mathsf{pk}_i}(v, \mathsf{pk}_d))$ to all other parties.

At the end of the protocol P_i outputs $v \in \mathcal{Z}_i^d$ if $|\mathcal{Z}_i^d| = 1$ and $\perp$ otherwise.

Fig. 4. The $\mathsf{RankedBroadcast}_d^\kappa$ protocol.

$(v, \mathsf{Sign}_{\mathsf{pk}_d}(v, \mathsf{pk}_d))$, so she sets $\mathcal{Z}_i^d = \{v\}$, where v is the value the dealer wants to broadcast. Note that an honest party will never accept a message with a different value, because the value has to be signed by the dealer, what implies that at the end of the protocol $\mathcal{Z}_i^d = \{v\}$ for each honest P_i and hence each honest party outputs v.

To prove the "consistency" consider for a moment a slightly simplified version of the RankedBroadcast protocol, namely one with the condition $|\mathcal{Z}_i^d| < 2$ omitted. We will now prove that in this simplified version at the end of the protocol it holds that $\mathcal{Z}_i^d = \mathcal{Z}_j^d$ for any two honest P_i, P_j at the end of the protocol. To this end consider an arbitrary $v \in \mathcal{Z}_i^d$. We will prove that $v \in \mathcal{Z}_j^d$. Suppose that P_i added v to $\mathcal{Z}_i^d$ during kth round. Consider two cases:

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

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

  to P_j at the end of k-th round and it was received by P_j in the $(k+1)$-st round. P_j will accept this message as all conditions are satisfied (or it already had been that $v \in \mathcal{Z}_j^d$), namely:
  - Conditions (1) and (2) are trivially satisfied.
  - It holds that $\mathsf{rank}_i(a_q) \leq k$ for any $1 \leq q \leq k$, because otherwise P_i 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 $\mathsf{rank}_j(a_q) \leq k+1$ for any $1 \leq q \leq k$. Moreover, $\mathsf{rank}_j(\mathsf{pk}_i) = 0$, because P_i is honest. Hence, condition (3) is satisfied.
  - If the condition (4), namely $v \notin \mathcal{Z}_j^d$ is not satisfied than we already have that $v \in \mathcal{Z}_j^d$.

  Hence, either P_j accepted this message and added v to $\mathcal{Z}_j^d$ at $(k+1)$-st round or it was already true that $v \in \mathcal{Z}_j^d$.
- $k = \ell$. It means that P_i 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 $v \in P_j$ and therefore at the end of the protocol each party has the same value of the set $\mathcal{Z}_i^d$ and outputs the same value.

The condition $v \in \mathcal{Z}_i^d$ is added to the RankedBroadcast 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 $\mathcal{Z}_i^d = \{a, b\}$ for some honest P_i and some a, b. Hence, P_i does not have to broadcast any messages with values other than a and b, because already we know that during the execution of the protocol each honest P_j will accept the message with values a and b unless it already had accepted some other two values. Therefore, in this case each honest party outputs $\perp$.

The "bounded creation of inputs" property comes from the "bounded creation of identities" property of the RankedKeys protocol (as clearly the size of each $\mathcal{Y}_i$ that is output by P_i cannot be larger than his set $\mathcal{K}_i$). □

Note that resistance of $\mathsf{RankedBroadcast}_d^\kappa$ to the DoS attacks comes from the fact that the parties are only accepting one message per round for each recognized identity (and there are at most ℓ such identities). It is also easy to see that (both in the public channel model and in the bilateral channels model) that the communication and message complexities of the $\mathsf{RankedBroadcast}^\kappa$ protocol are dominated by the execution of $\mathsf{RankedKeys}^\kappa$, and hence asymptotically these complexities are as in case of the $\mathsf{RankedKeys}^\kappa$.

6.3 A group with an honest majority

We now show how to use the protocols from the previous sections to construct an honest majority generation protocol. The protocol is depicted on Figure 5.

Lemma 6. *The* $\mathsf{HonestMaj}^\kappa$ *protocol is an honest majority* Σ*-key generation algorithm.*

Proof. The "key-generation" property holds trivially since the secret keys sk_i are generated using Gen and are never used for signing messages. The "consistency" property holds since the output of each P_i is a deterministic function of $\mathcal{K}$ (that has the same value for every P_i).

Suppose n, π and $\pi_{\mathcal{A}}$ are such that $\lceil \pi_{\mathcal{A}}/\pi \rceil < n$. Since the broadcast protocol has the "bounded creation of inputs" property, thus $m \leq n + \lceil \pi_{\mathcal{A}}/\pi \rceil$. Hence $n > m/2$, and therefore if $k \in \mathcal{K}$ then for at least one P_i we have $\mathsf{rank}(k) = 0$. By the "bounded creation of identities" property of the RankedKeys protocol the set of such k's is at most $n + \lceil \pi_{\mathcal{A}}/\pi \rceil$ thus the $\mathsf{HonestMaj}^\kappa$ protocol satisfies "bounded generation of identities" property.

It is also easy to that every pk_i belongs to at least n sets in the family $\mathcal{K}$ (since $\mathsf{rank}_j(k) = 0$ for every P_j). Since $n > m/2$ thus such $\mathsf{pk}_i \in \mathcal{K}$. This shows the "validity" property. □

1. The party P_i execute the $\mathsf{RankedKeys}^\kappa$ protocol. Let $(\mathsf{sk}_i, \mathsf{pk}_i, \mathcal{K}_i, \mathsf{rank}_i)$ be the output of each P_i. Let $\mathcal{K}_i^0$ denote the set of keys k such that $\mathsf{rank}_i(k) = 0$.
2. The parties execute the $\mathsf{RankedBroadcast}^\kappa$ protocol with the input of each P_i being $\mathcal{K}_i^0$. Note that from the properties of the broadcast protocol the output of every party is identical. Let $\{\mathcal{X}_1, \ldots, \mathcal{X}_m\}$ be the family of sets that the parties receive as the output of this protocol.
3. Define $\mathcal{K} := \{k : k \text{ belongs to more than } m/2 \text{ of the sets } \mathcal{X}_1, \ldots, \mathcal{X}_m\}$
4. Output $\mathcal{K}$

Fig. 5. The $\mathsf{HonestMaj}^\kappa$ protocol.

7 Applications

7.1 Multiparty computation protocols with honest majority of computing power

As already mentioned before, we can use the HonestMaj protocol from Section 6.3 to establish a group of parties that can later perform the MPC protocols. This is possible, since the HonestMaj protocol identifies

the parties by the set of their public keys. It is well-known, that given such a trusted set-up the parties can emulate any trusted functionality [34], provided that the majority of them is honest (which is the case here). Such a trusted functionality can be anything that the parties find useful. For example it can be a procedure for generating a uniformly random beacon, or a system for maintaining a trusted server (a discussion board, say) in the peer-to-peer network. It could also potentially be used for the "Internet of Things" applications, e.g., to replace the blockchain paradigm in the "Adept" technology [39]. In principle it can also lead to creations a new digital currencies (we discuss it in Section 7.3).

7.2 Unpredictable beacon generation without honest majority of computing power

The RankedBroadcast protocols can also be used to produce unpredictable beacons even if there is no honest majority of computing power in the system. This 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 $\mathsf{RankedBroadcast}^\kappa$ 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).

7.3 Provably secure cryptocurrencies?

It might be tempting to say that the honest majority created by the HonestMaj protocol 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 (using the HonestMaj protocol). 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 [20]).

References

1. Crypto-currency market capitalizations. coinmarketcap.com/, Accessed on 27.05.2014.
2. M. Andrychowicz, S. Dziembowski, D. Malinowski, and Ł. Mazurek. Secure Multiparty Computations on Bitcoin, 2014. 35th IEEE Symposium on Security and Privacy (Oakland). ACM.
3. James Aspnes, Collin Jackson, and Arvind Krishnamurthy. Exposing computationally-challenged byzantine impostors. *Department of Computer Science, Yale University, New Haven, CT, Tech. Rep*, 2005.
4. 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`.
5. 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.
6. Adam Back. Hashcash - a denial of service counter-measure, 2002. technical report.
7. Lear Bahack. Theoretical bitcoin attacks with less than half of the computational power (draft). *arXiv preprint arXiv:1312.7013*, 2013.
8. 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.
9. 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.
10. Nir Bitansky, Ran Canetti, and Shai Halevi. Leakage-tolerant interactive protocols. In Ronald Cramer, editor, *TCC 2012*, volume 7194 of *LNCS*, pages 266–284, Taormina, Sicily, Italy, March 19–21, 2012. Springer.
11. Bitcoin. Wiki. en.bitcoin.it/wiki/.
12. 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.
13. Elette Boyle, Shafi Goldwasser, Abhishek Jain, and Yael Tauman Kalai. Multiparty computation secure against continual memory leakage. In Howard J. Karloff and Toniann Pitassi, editors, *44th ACM STOC*, pages 1235–1254, New York, NY, USA, May 19–22, 2012. ACM.
14. 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.
15. David Chaum, Amos Fiat, and Moni Naor. Untraceable electronic cash. In Shafi Goldwasser, editor, *CRYPTO'88*, volume 403 of *LNCS*, pages 319–327, Santa Barbara, CA, USA, August 21–25, 1988. Springer.
16. 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`.
17. 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.
18. 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/`.
19. 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.
20. Nicolas T. Courtois. On the longest chain rule and programmed self-destruction of crypto currencies. *CoRR*, abs/1405.0534, 2014.
21. Nicolas T Courtois and Lear Bahack. On subversive miner strategies and block withholding attack in bitcoin digital currency. *arXiv preprint arXiv:1402.1718*, 2014.
22. Ivan Damgård, Carmit Hazay, and Arpita Patra. Leakage resilient secure two-party computation. Cryptology ePrint Archive, Report 2011/256, 2011. `http://eprint.iacr.org/2011/256`.
23. Danny Dolev and H. Raymond Strong. Authenticated algorithms for byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983.
24. 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.
25. 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.
26. 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.
27. 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`.

28. 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.
29. Ittay Eyal and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. *arXiv preprint arXiv:1311.0243*, 2013.
30. 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.
31. Stephen Foley. Dish network signs up to accepting bitcoin, May 2014. Financial Times, www.ft.com/intl/cms/s/0/3a22880c-e742-11e3-88be-00144feabdc0.html.
32. 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.
33. Christina Garman, Matthew Green, and Ian Miers. Decentralized anonymous credentials. Cryptology ePrint Archive, Report 2013/622, 2013. `http://eprint.iacr.org/2013/622`.
34. O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. STOC, 1987.
35. 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.
36. 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.
37. Keith Griffith. ebay is considering integrating bitcoin into paypal, May 2014. Business Insider, www.businessinsider.com/ebay-is-considering-integrating-bitcoin-into-paypal-2014-5.
38. Nermin Hajdarbegovic. Amazon awarded bitcoin-related cloud computing patent, May 2014. CoinDesk, www.coindesk.com/amazon-awarded-bitcoin-related-cloud-computing-patent.
39. 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.
40. Martin Hirt and Ueli M. Maurer. Player simulation and general adversary structures in perfect multiparty computation. *Journal of Cryptology*, 13(1):31–60, 2000.
41. 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.
42. Sunny King and Scott Nadal. Ppcoin: Peer-to-peer crypto-currency with proof-of-stake. .
43. Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.
44. 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.
45. 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.
46. Andrew Miller, Elaine Shi, and Jonathan Katz. Non-outsourceable scratch-off puzzles to discourage bitcoin mining coalitions, 2014.
47. 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.
48. Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Consulted*, 1(2012):28, 2008.
49. Michael O. Rabin. Transaction protection by beacons. *Journal of Computer and System Sciences*, 27(2):256 – 267, 1983.
50. Bitcoin Talk. [ann] litecoin - a lite version of bitcoin. launched! bitcointalk.org/index.php?topic=47417.0, Accessed on 27.05.2014.
51. Bitcoin Wiki. Denial of service (dos) attacks. en.bitcoin.it/wiki/Weaknesses , Accessed on 26.09.2014.
52. Bitcoin Wiki. Network. en.bitcoin.it/wiki/Network, Accessed on 26.09.2014.
53. 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 [48,11,2]). The same idea is used in several other cryptocurrencies like Litecoin [50] (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 [42] (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 [48] 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 perfectly sound, 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 [29] 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 [7] 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 [21].

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 [41,36,32].

A.3 The "genesis block" generation.

One a more theoretical side, what may be considered unsatisfactory is the fact that the Bitcoin 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* [49] 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 [49,28], voting schemes [47], or zero-knowledge protocols [5,35]. Note that a random 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 [16] (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). 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 [29] 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[10].

B Additional machinery for the random oracle model

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 7. *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 8. *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 $\hat{t}^2 \cdot 2^{-\kappa}$. □

[10] 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.

C Proof of Lemma 1

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 7 and 8 the probability that the execution was either not canonical or not collision-free is at most $3\hat{t}^2 \cdot 2^{-\kappa}$. □

D Proof of Lemma 2

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 $\alpha \cdot \lceil \log_2 t \rceil$. □

E Proof of Lemma 3

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 B) 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_λ^κ 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_λ^κ 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 $\mathcal{X}_1^q, \ldots, \mathcal{X}_\alpha^q$ are independent. Moreover, since the outputs of the random oracle are uniform, hence for every i the probability of $\mathcal{X}_i^q$ is equal to the cardinality m of $|\Sigma|$ divided by the total number $\lceil t/2 \rceil \geq t + 1$ of leaves in $\mathsf{Merkle}_{t,c}^\kappa$. 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. $\mathcal{X}^q := \wedge_{i=1}^{\alpha} \mathcal{X}_i^q$. From the independence of $\mathcal{X}_i^q$'s we get that

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

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 $\mathbb{P}(\mathcal{X}) = \hat{t}_1((\hat{t}_1 + 1)/t)^\alpha$. 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 3. Since, by Lemmas 7 and 8 (in Appendix B) 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 $(3\hat{t}^2 + 1) \cdot 2^{-\kappa} + \hat{t}_1((\hat{t}_1 + 1)/t)^\alpha$. □

F Proof of Lemma 4

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 interval. 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_i to compute it.

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

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

What remains is to prove the *bounded creation of identities* property. To see why it holds observe that each party P_i 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_i also checks if $c_i^{\ell-k} \prec B^{\ell+1-k}$, where $c_i^{\ell-k}$ is P_i's challenge from the $(\ell-k)$th interval. Since this challenge needs to be a function of P_i's' challenge c_i^0 from the 0th interval, 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

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

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

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

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 $\lceil\pi_{\mathcal{A}}/\pi\rceil$. □

G Communication and message complexity of the RankedKeys$^{\kappa}$ protocol

Communication and message complexity in the public channel model. Before we provide an efficiency analysis of the RankedKeys$^{\kappa}$ scheme let us optimize it a bit[11]. First of all, observe that the only reason why

[11] 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.

$(\mathsf{Key}^k, \mathsf{pk}, A_i^{\ell-k}, B^{\ell+1-k}, \ldots, B^{\ell+1}, \mathsf{Sol})$ is sent by P_i at the end of the kth round of the "key ranking phase" is that each other party P_j needs to be able to check in the the $(k+1)$st round that $c_j^{\ell-(k+1)} \prec A_i^{\ell-k}$ and $F(A_i^{\ell-k}) \prec B^{\ell+1-k}$, 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 5) in the following way. Instead of using a hash function F we use MHash, and then instead of sending $A_i^{\ell-k}, B^{\ell+1-k}, \ldots, B^{\ell+1}$ we send $A_i^{\ell-\kappa}, \mathsf{MHash}(B^{\ell+1-k}), \ldots, \mathsf{MHash}(B^{\ell+1})$ together with the Merkle proofs that $F(A_i^{\ell-k})$ 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 1 (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 [12]. 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 $A_i^{\ell-k}$ vectors is to allow each party to check that here challenge (that is $c_j^{\ell-(k+1)}$) was used to compute $c_i^{\ell-k} = F(A_i^{\ell-k})$. In the bilateral channels model we can modify this by using once again the Merkle trees: each P_i sends to each P_j a Merkle hash $\mathsf{MHash}(A_i^{\ell-k})$ together with a proof that P_j's challenge $c_j^{\ell-(k+1)}$ was used to compute it. Later, P_j 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 $O(n\ell)$.

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