Blockchain Protocols: The Adversary is in the Details

Rachid Guerraoui
EPFL
rachid.guerraoui@epfl.ch

Matej Pavlovic
EPFL
matej.pavlovic@epfl.ch

Dragos-Adrian Seredinschi*
EPFL
dragos-adrian.seredinschi@epfl.ch

ABSTRACT

Blockchain-like protocols are flourishing. Maybe not surprisingly, the differences among these protocols are often subtle and difficult to understand. More importantly, it is often unclear what the weaknesses of each of these protocols are and how easily they can be attacked. The goal of this paper is to shed light on the important differences between blockchain protocols and the impact these differences can have in terms of their vulnerabilities. We cover well-studied protocols ranging from those inspired from the distributed systems literature (e.g., PBFT), to recent research prototypes (e.g., ByzCoin or Algorand), including the popular Bitcoin protocol.

Towards reaching our goal, we first precisely define the problem that these protocols seek to solve. Then we propose a unifying scheme that captures, at a high level, the behavior of any blockchain protocol. Interestingly, this scheme is also sufficiently low level to highlight the important differences between these protocols. We show that blockchain-like protocols can be differentiated according to their degree of indulgence—i.e., tolerance towards node misbehavior or towards network asynchrony—which translates into the different vulnerabilities of each of these protocols.

1. INTRODUCTION

Since the advent of Bitcoin, tens—if not hundreds—of variations on this protocol have been proposed. These variations, which we call simply Bitcoin-like protocols, usually have a twofold purpose: (1) to improve reliability with respect to the original Bitcoin protocol by withstanding severe attacks, or (2) to improve efficiency by either decreasing latency or increasing throughput. Usually, these objectives are antagonist. For example, decreasing the time it takes to commit a transaction in a blockchain protocol can make that protocol more vulnerable to double spending [22].

The differences between various blockchain protocols are often subtle, and each improvement to a certain protocol may open vulnerability breaches which are not clear a priori. For instance, it is perhaps alarming to see the recent uncovering of critical errors in algorithms that can be used to implement blockchains, namely in Tangaroa [10], Zyzzyva and FaB [3]. This makes the current blockchain ecosystem very chaotic, which, to say the least, is rather disappointing, given that these protocols are mainly aimed at implementing distributed trust.

The motivation of this paper is to help clarify this state of affairs. Our aim is not a rigorous formalization of a specific blockchain protocol and its properties, as done in a significant body of related work [8, 18, 29]. Instead, we propose a high-level, *adversary-oriented* approach to deconstructing blockchain protocols. Ultimately, our goal is to offer a better understanding of blockchain variations (e.g., efficiency or reliability enhancements), discussing which variation opens which vulnerability breach. Our principled approach is inspired from the theory of distributed computing. As we will recall, blockchain protocols are solving a classical distributed computing problem. We proceed in several steps.

First, we define precisely the blockchain problem, namely the problem that seeks to be solved by the original Bitcoin protocol and its many variants. Roughly speaking, the problem consists of building a highly available (replicated) set of single owner bank accounts while avoiding double spending. Indeed, blockchain is both the name of (a) the chain of transaction blocks that need to be replicated and maintained consistently to enable Bitcoin transactions, as well as (b) the protocol that maintains this consistency. In a sense, blockchain is the name of the *solution* (b), as well as the name of the *problem* (a) being solved. Whilst solutions have been discussed at great length, we believe the problem specification has been largely ignored.

We define the problem precisely in terms of safety and liveness properties [5]. We then use classical results in distributed computing to highlight how the blockchain problem is harder, for example, than building a replicated file system [7], but is equivalent to the celebrated consensus [31] and State Machine Replication (SMR) problems [32]. It is known that there is no deterministic solution to consensus if we assume that the network can be asynchronous and at least one node can crash (even if no node can act adversarially) [17]. No matter what solution is designed, adverse network conditions can defeat it.

We then present a general scheme that unifies solutions to the blockchain problem. We show how all solutions to this problem restrict the power of the adversary in one way or another, e.g., either by assuming a bound on the number of misbehaving nodes which an adversary controls, or by assuming a bound on network asynchrony. We study here well-known protocols like PBFT [11] or Bitcoin [27], as well as recent research efforts

*This work has been supported in part by the European ERC Grant 339539 - AOC

Symposium on Foundations and Applications of Blockchain (FAB '18) March 9, 2018, Los Angeles, California, USA.

such as ByzCoin [24], Bitcoin-NG [15], and Algorand [19].

In short, our general scheme builds upon two fundamental components: a *leader election* subprotocol and a *commitment* subprotocol. The goal of the first subprotocol is to elect a node (or a set of nodes) to lead the task of ordering transactions. The goal of the second subprotocol is to make sure the ordering is global and the decision is unique, in case a new (or concurrent) leader is elected and considers a different ordering. This intuitive decomposition helps describe the avenues for attack which an adversary can take to subvert a blockchain protocol. It also enables us to point out critical differences between blockchain protocols as well as draw parallels between protocols.

We point out the existence of two classes of protocols. A protocol which represents the first class is Castro and Liskov's PBFT [11]. This class of protocols preserves its safety property, namely, consistency, despite the harshest conditions of the network (i.e., asynchrony). We say that this class is indulgent towards asynchrony and call it *asynchrony-indulgent* (or A-indulgent). A representative of the second class is Bitcoin [27]. This protocol continues executing (i.e., preserves liveness) despite an adversary mounting a Sybil attack, polluting the system with many misbehaving nodes. We say that this protocol is *behavior-indulgent* (or B-indulgent).

We organize the rest of this paper as follows. We discuss the problem addressed by blockchain protocols through the lens of distributed computing, and introduce the A-indulgent and B-indulgent classes of blockchain protocols (§2). We then introduce a general scheme which captures the essential behavior of any blockchain protocol, and use this scheme to discuss two notable blockchain protocols—PBFT and Bitcoin—showing how each is a typical example of respectively the A-indulgent and B-indulgent class (§3). We also relate a few other protocols to our general scheme and discuss their indulgence (§4), and then we conclude this paper (§5).

2. THE PROBLEM

2.1 State Machine Replication and Consensus

On-line services often employ replication to ensure their availability despite failures in the underlying systems. A common method to achieve this is via *state machine replication (SMR)* [32]. In SMR, a service, such as a financial ledger or an online shopping cart, is modeled as a deterministic state machine. The service consists of (1) a service state, and (2) operations that can be applied on this state. Typically, each replica (or node) of the system maintains its own local copy of the state, and updates this state as a result of applying client operations.

In SMR, the operations have to be deterministic, i.e. the operation result and the new state it produces are a function of only the previous state and the operation itself. Any service state can thus be uniquely defined by the initial state and a sequence of operations applied on this initial state.

In order to keep the service state consistent, replicas need to apply the *same operations in the same order*. In other words, SMR requires that the sequence of operations applied at all replicas is the same; the main challenge in implementing SMR is ensuring this requirement. The challenge can be reduced to the fundamental problem of *consensus* (agreement) in a distributed system, where all replicas need to agree on what the n-th operation of the sequence will be, for an ever-increasing n. In Figure 1 we sketch the typical architecture of an SMR system as we have presented it so far. The system comprises 6 replicas, labeled from 0 to 5. At the heart of the system

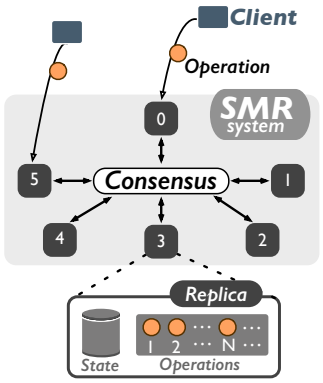

Figure 1: State Machine Replication in action.

lies a distributed consensus algorithm which the replicas use to agree on the sequence of incoming client operations and maintain the consistency of the replicated state.

A consensus algorithm must satisfy three essential properties:

- **Validity:** The agreed-upon operation must be the input of one of the replicas (e.g., a client operation).
- **Agreement:** The agreed-upon operation is the same for all correct replicas.
- **Termination:** The replicas will eventually agree on some operation.

Validity and agreement are *safety properties*: they define events that must never happen in a correct execution. Termination, on the other hand, is a *liveness property*, defining that a correct execution must make some progress [5].

2.2 Replicated Ledgers

We focus on a specific type of service: a *ledger*. Without loss of generality, we assume that a ledger describes the movement of money across different bank accounts. Concretely, a ledger is an ever-growing sequence of transactions, each of which transfers money between the users (i.e., clients) of the system.

Replicating a ledger is non-trivial, and even the relation between a ledger and consensus is not immediate. To understand this relation, we start from the simple observation that a ledger is no different than a *fetch-and-add* object [20]. As shown by Herlihy [23], such an object has consensus number 2. This means that, in a shared memory model, up to two processes (but not more) can solve consensus among themselves if they have access to fetch-and-add, i.e., a ledger object. Any object weaker than a ledger, such as simple *read-write register*, is insufficient to solve consensus in shared memory. In more practical terms, the problem of implementing a replicated ledger is strictly more difficult than that of implementing a file system or a key-value store, both of which have a *read-write* interface [7].

Note, however, that we are not interested in the shared memory model, but in the message passing model (detailed in §2.4). Interestingly, in a message passing system, Delporte-Gallet et al. [13] showed that replicating any object that has a consensus number greater than one—such as *fetch-and-add*—is equivalent to solving consensus. In other words, replicating a ledger is equivalent to solving consensus. We also know that solving consensus allows us to replicate *any* object via the SMR approach. Thus, we conclude that in the message passing model there is no object that is harder to replicate than a ledger.

In the following, we explain how the abstract notions of consensus and SMR relate to distributed ledgers.

2.3 Ledgers as Replicated State Machines

Modeling a ledger as a replicated state machine is straightforward. The SMR state is an ordered sequence of all transactions performed in the past, while each new SMR operation represents the appending of a new transaction to the ledger.

The system replicas must agree (i.e., solve consensus) on which transaction should take the n-th position in the ledger. Validity is easy to satisfy in this context using standard cryptography, so we do not focus on it henceforth. The challenge is to achieve both agreement (safety) and termination (liveness).[1] We require a replicated ledger to have the following properties:

1. **Safety**: If a replica accepts a transaction T ordered after some transaction T', then no replica accepts T without having ordered T' before T.
2. **Liveness**: If a client issues a transaction, the transaction is eventually accepted at all correct replicas.

We note that *accepting* a transaction has a nuanced meaning. In certain protocols, like PBFT, there is a specific point where a transaction becomes irrevocable (this is often referred to as *consensus finality* [33]). We say that such a transaction is accepted. Protocols which lack consensus finality, like Bitcoin, always permit the revocation of transactions—albeit with diminishing probability. When a revocation occurs, it represents a safety violation; in this sense, Bitcoin is prone to violating safety (§3.2.2).

Informally, safety prescribes that different replicas never have a different view of what the n-th transaction is. To understand the importance of safety, imagine, for example, two replicas R_0 and R_1 participating in a protocol that replicates a ledger. Both R_0 and R_1 have the same view of the first $n-1$ transactions; only one of these transactions states that a client called Eve receives 100$. However, Eve manages to make replica R_0 believe that the n-th transaction is "Eve transfers 100$ to Alice", while convincing R_1 that the n-th transaction is "Eve transfers 100$ to Bob". Such a situation clearly violates agreement. The balance in Eve's account was 100$, so only *one* of these transactions should be accepted by the system. If Alice and Bob consult R_0 and R_1 respectively to obtain the state of the ledger, they may both believe to have received money from Eve and provide her with some goods or services in exchange.

Such situations, where Eve effectively spends the same money twice, are known as *double spending*. Indeed, in the context of ledgers, safety violations can always be related to double spending. Different protocols have different approaches to prevent this problem. We elaborate later how double spending can occur in notable blockchain protocols (§3.2 and §4).

We recall that replicating a ledger is equivalent to replicating any state machine, as we argued earlier (§2.2). Traditionally, protocols like PBFT are employed for general-purpose SMR. Recently, however, protocols in the vein of Bitcoin are used as well towards implementing SMR (e.g., Ethereum, which generalizes transactions to so-called smart contracts [2]). For simplicity, we restrict ourselves to distributed ledgers which contain only monetary transactions; these are sufficient to explain all principles discussed in this paper, and a generalization to arbitrary state machines is straightforward.

[1]Note that, in general, achieving only one of liveness and safety is trivial. A protocol doing nothing never violates safety, but violates liveness. On the other hand, it is easy to make progress (satisfying liveness) if the output need not be correct (violating safety).

2.4 The Adversary

We consider a message passing model where nodes communicate by exchanging messages. The adversary can control various parts of the system, and there are two kinds of assumptions on the capabilities of this adversary.

Behavior assumptions define how much control the adversary can exert over the behavior of nodes (i.e., over the correctness of their computation). These are typically known as fault-threshold assumptions in distributed computing [26], and a common example is that at least two thirds of replicas are correct and follow the protocol faithfully [11], i.e., these replicas are not corrupted by the adversary.

Synchrony assumptions define how much control the adversary has over the speed of (otherwise corect) computation at nodes, as well as over the message transmission delays and delivery guarantees of the network. For example, the reliable message delivery or the absence of network partitions both fall under synchrony assumptions.

The precise definition of what the adversary can and cannot do plays a great role. A truly Byzantine adversary has no restrictions [25], but such a model is very restrictive in terms of the solutions it allows. Perhaps the most common assumption on a Byzantine adversary is that it cannot subvert cryptographic primitives. For instance, standard cryptographic assumptions prevent the adversary from inverting a secure hash function or producing a valid cryptographic signature without knowledge of the corresponding private key [11].

Another common assumption on the Byzantine adversary limits its interference with the nodes which it does not directly control. For example, it is often assumed that the adversary cannot prevent correct nodes from making progress (e.g., communicating with each other) indefinitely through a denial of service attack, a permanent network partition, or unremitting dropped messages. Typically, the assumption is that messages sent by correct nodes eventually reach their destination.

As we will see in the following sections, it is very often exactly these fine details in what the adversary can and cannot do—and for how long—that make the difference between the guarantees various protocols offer. We will discuss different protocols also from this point of view, i.e., we show how these assumptions influence the protocols' safety and liveness guarantees.

We ask ourselves the following question: *What is necessary to compromise the liveness and / or safety of a blockchain protocol?* Obviously, correctness of a protocol (i.e., upholding both safety and liveness) always depends on every assumption a protocol makes, otherwise the assumption would not be needed in the first place. However, not necessarily both of safety and liveness break when certain assumptions are violated. Focusing primarily on safety, we discuss two classes of protocols:

- **A-indulgent** protocols: protocols indulgent to *asynchrony*. These are protocols which focus on maintaining safety while putting minimal restrictions on the adversary in terms of synchrony.
- **B-indulgent** protocols: protocols which are indulgent towards bad or malicious node *behavior*. These protocols focus on maintaining safety while tolerating a relatively large number of malicious nodes.

The famous FLP impossibility result [17] shows that solving consensus without restricting the adversary is impossible. Protocols thus rely on various assumptions to circumvent this impossibility and guarantee both safety and liveness. PBFT, for example, can guarantee safety without any synchrony

assumptions on the adversary whatsoever. Synchrony assumptions are, however, required for liveness. Bitcoin, on the other hand, remains correct even when allowing the adversary more control over node behavior (an overwhelming fraction of nodes can behave maliciously as long as they do not possess enough computing power). Bitcoin, however, requires additional synchrony assumptions to remain safe. In this sense, PBFT is A-indulgent and Bitcoin is B-indulgent.

In the rest of this paper, we examine in more detail why PBFT and Bitcoin are each a representative of one of these classes. We then discuss other protocols, which, interestingly, can lie in between the two classes (specifically, Algorand), or are a combination of both classes (ByzCoin).

3. GENERAL SCHEME

In this section we introduce a general scheme that captures, at a high level, the behavior of any protocol implementing a blockchain. We also discuss how two notable blockchain protocols can be expressed using our scheme, and show how these protocols lie in sharp contrast to each other, representing the A-indulgent and the B-indulgent class, respectively.

Implementing a blockchain that prohibits double-spending is a challenging multi-level problem. First, most protocols rely on a leader election mechanism. Leader election must ensure that a unique leader presides over the protocol steps; this is important for maintaining consistency, since two leaders can engender disagreements. Generally, the existence of a leader simplifies implementations and reasoning about distributed protocols [28]. Blockchains operate in a Byzantine environment, however, where some replicas—including the leader—may fail arbitrarily. Even if a unique leader is correctly chosen, it can act maliciously, e.g., by equivocating. A second problem, then, is ensuring that transactions do not conflict and all correct replicas maintain the same view on the blockchain data structure. Yet a third problem can appear in protocols which are optimistic and permit temporary conflicts to exist across replicas; in such cases, additional measures are necessary to resolve conflicts. We capture all these difficulties in a general scheme that most algorithms follow in one way or another.

3.1 General Scheme

In broad strokes, we argue that the behavior of any protocol implementing a blockchain comprises four basic steps. These steps are as follows:

① a client *issues* a transaction;

② a *leader election* protocol determines a leader to marshal the transaction;

③ the replicas *commit* on an ordering proposed by the leader, i.e., they externalize the output, for instance, by replying to the client or by executing the transaction on their local state;

④ if the protocol allows conflicts to arise (which are often called *forks*), then a *recovery scheme* triggers to reconcile such conflicts.

The same replica may play different or multiple roles—be it client, leader, or ordinary replica—in this four-piece scheme. Indeed, in most protocols, each replica may become a leader at some point. The most notable differences between blockchain protocols arise at the level of steps ② and ③, namely in the protocol's method of dealing with the problems of leader election and commitment on a proposal. These steps are particularly difficult because it is at either of these two points where disagreements among replicas may arise, which can lead to safety violations (i.e., double-spending). In some protocols (such as PBFT), leader election takes place a priori, and the same leader tends to be reused across multiple transactions, as long as that leader behaves correctly and is able to communicate with a certain fraction of the system replicas [11]. In other protocols (such as Bitcoin), leader election happens on the critical path of handling transactions. Often, step ④ is absent from blockchain protocols. Unless we explicitly state what this step entails, we consider it to be absent because the protocol avoids disagreement by design, and hence no recovery is necessary.

3.2 Two Extremes of Blockchain Algorithms

We use our general scheme to present a breakdown, at a very high level, of PBFT [11] and Bitcoin [27] protocols. Interestingly, these protocols represent two extremes with respect to leader election and commitment, which translates into each of them belonging to one of the two indulgence classes, as we show next.

3.2.1 PBFT – Practical Byzantine Fault Tolerance

In PBFT, clients send their transactions to some replica i which they believe to be the current leader; if replica i is not the leader, then i simply forwards the request to the actual leader. This is step number ① of our scheme.

In PBFT, the leader role switches from one replica to another in a round-robin manner. Leader election—i.e., step ②—takes place only if there is a suspicion that the current leader has failed, prompting the system to switch to the next leader by executing a *view-change* sub-protocol.[2] If the leader acts correctly and the network is synchronous so as to permit progress, then no leader election occurs.

Step ③ in PBFT takes the form of a three-phase protocol. This protocol is essentially a quorum-gathering technique with Byzantine fault-tolerance, and ensures that if a correct replica commits on the leader's proposal, then no correct replica commits on a different proposal. Any correct replica in the PBFT protocol commits on some proposed ordering for a transaction after that replica is certain that a majority of replicas also commit on the same ordering.

A major drawback of PBFT-like protocols is that all replicas must have complete and consistent knowledge of all other replicas (i.e., the *membership set*) in the system at a given time. This information can be statically setup at system deployment and never changed, which is impractical for real-world replicated ledgers, as participants are expected to change over time. Dynamic membership can be achieved through a reconfiguration module [4, 9]. For instance, the very same mechanism that is used to agree on the contents of the ledger can also be used to agree on the membership. In this case, membership is redefined using a special transaction which changes the membership set. When nodes commit on such a special transaction, they also agree to update their view of the current membership set.

PBFT as an A-indulgent algorithm.

Informally, in PBFT any decision happens after the leader

[2]In PBFT, the leader is called the *primary* replica, and each replica is a primary in a given *view*. The view-change sub-protocol achieves the switching from a view to the next (hence, it also switches the primary to a different one), which we do not explain here. We refer the interested reader to the original PBFT description for more details [11].

asks permission from a quorum; in this case, a quorum comprises more than 2/3 of the replicas [11, 26]. Hence, a supermajority of the system replicas must coordinate via a three-phase protocol to agree on committing any decision. Irrespective of how badly the network behaves—such as being asynchronous or affected by a severe partition—PBFT always remains *safe* as long as the 1/3 threshold of faulty replicas is maintained. For this reason, we call PBFT an *A-indulgent* algorithm. In the classic sense defined by Guerraoui [21], PBFT is indulgent towards asynchrony in the network, permitting arbitrary periods of such asynchrony, because each correct replica refrains from taking any decisive step before consulting with a majority of replicas to agree on that step.

The only attack vector on PBFT's safety is controlling a fraction of at least a third of all replicas. In a dynamic setting, as we described earlier, gaining control over a third of replicas can be easy for an adversary. In a Sybil attack [14], the adversary simply spawns many replicas and makes them all join the system. As creating replicas is comparatively cheap in terms of computational and communication resources, enacting such an attack is realistic in practice. When controlling more than a third of replicas, such an adversary can convince two correct nodes to accept different transactions, *t1* and *t2*, at the same position in their ledger. These transactions can be crafted so as to permit double-spending, i.e., both *t1* and *t2* can be spending the same money, as in our example with Eve from §2.3.

To prevent the adversary from controlling a big fraction of replicas, PBFT requires an additional protection mechanism. This mechanism can take various forms, such as an access control scheme based on a certificate authority, a mechanism able to identify Sybil identities [6], or requiring participants to dispose of important amounts of a scarse resource (similar in spirit to Bitcoin's proof-of-work, which we will discuss next). To wrap-up, PBFT is mainly susceptible to an adversary that can manipulate the behavior of a large number of replicas in the system. As noted, however, PBFT is A-indulgent in the sense that its safety is resilient towards asynchrony.

The next algorithm we visit is Bitcoin. This protocol lies in sharp contrast with PBFT in its indulgence, i.e., in the way it deals with asynchrony or with adversarial behavior.

3.2.2 Bitcoin

Nodes in Bitcoin-like protocols are called miners. At step ①, clients issue their transactions to multiple miners, typically through a gossip-based broadcast scheme. Each miner independently assembles a block of transactions and then starts executing a proof-of-work (PoW) algorithm.

The PoW algorithm serves, primarily, as a leader election scheme, that is step ②. In contrast to PBFT (where leaders succeed each other whenever the views change), in Bitcoin all miners are striving to become a leader for every new block of transactions. Briefly, any miner can become the leader if it successfully solves a cryptographic puzzle—essentially, inverting a hash function [27]—faster than other miners.

Upon finding a puzzle solution, the miner becomes the de facto leader, and it broadcasts its solution, proposing an ordering and commencing step ③. The solution is uniquely bound to the block which the miner assembled earlier, and chained to the whole history of older blocks via a hashing algorithm, hence giving the name of blockchain to the resulting data structure.

Step ③ is more nuanced in Bitcoin than in other systems. Briefly, when another node observes a block, committing on it simply means appending the corresponding block at the end of its local blockchain. Note, however, that leader election does not guarantee a unique leader, because multiple miners can solve the puzzle for the same position in the blockchain. Since all puzzle solutions are equally valid, this gives rise to a fork. To select a specific branch of the fork and recover from the conflict, nodes in Bitcoin simply wait until one of the branches is extended with subsequent solutions. This is known as "the longest chain wins" rule, and represents step ④ in our general scheme. Typically, before committing some block b, nodes wait until additional blocks—called *confirmations*—are found, thereby extending b and raising the confidence that b will not be abandoned. The number of confirmations can vary; a node might even choose to wait for no confirmations and instantly accept a block without any waiting time [1].

Bitcoin as a B-indulgent algorithm.

At a high level, PBFT and Bitcoin differ in one critical aspect. Nodes in PBFT choose leaders and commit on values *after* consulting with a majority of the system. Nodes in Bitcoin commit on a value after some time passed and that value accumulated a certain number of confirmations; the Bitcoin protocol, briefly, relies on timing assumptions in the commitment step. For this reason, Bitcoin does not qualify as an A-indulgent algorithm: if timing assumptions do not hold (e.g., an adversary partitions the system), then Bitcoin is vulnerable to attacks on its safety. Instead, we label Bitcoin as a B-indulgent algorithm, since this protocol tolerates adversarial behavior in the nodes, even if the adversary controls the vast majority of the identities (as long as their combined computation power stays small enough). This difference between PBFT and Bitcoin reflects mainly at steps ② and ③ of these protocols.

Another important difference between Bitcoin and PBFT is the notion of quorum. Unlike PBFT that relies on a quorum in terms of the number of participating replicas, Bitcoin requires the cooperation of replicas that together possess a majority of the computing power in order to make decisions. While this has severe impact on energy efficiency and performance of the system, it serves as a protection against Sybil attacks, towards which Bitcoin is not vulnerable.

Various attacks have been crafted against the Bitcoin protocol. Several attacks have to do with increasing the rewards a miner can obtain in an unfair way, e.g., through selfish mining [16], block withholding, or fork after withholding attacks. Perhaps more important than fairness in rewards, we are interested in correctness attacks, that is, attacks that potentially lead to double spending. An eclipse attack is interesting for our discussion because it illustrates a concrete exploitation of the optimistic nature of the Bitcoin protocol [22]. An attacker with a sufficient number of IP addresses (in the order of thousands) can pollute—i.e., eclipse—an honest node's membership view, so that the honest node only has connections with the attacker and no other honest node in the Bitcoin network. Essentially, the attacker partitions the honest node from the rest of the network. Thereafter, the attacker creates a fork in the blockchain: in one branch the attacker spends money on certain goods, while in another branch the attacker is buying something from the honest node. The former branch is part of the actual Bitcoin network (and hidden from the eclipsed node), while the latter branch is eventually orphaned. Note that the attacker spent the same money on both branches. To summarize, an eclipse attack is a way to exploit Bitcoin's timing assumptions towards carrying out a double spending attack, i.e., violating Bitcont's safety.

4. BLOCKCHAIN VARIATIONS

In this section, we position a few notable blockchain protocols (Algorand, ByzCoin and Bitcoin-NG) in relation to the general scheme we introduced earlier (§3.1). For brevity, we adopt a high-level view on each protocol and the focus will be on their properties and assumptions (not on algorithmic details) with respect to A-indulgence and B-indulgence.

4.1 Algorand

Algorand [19] is a recent blockchain protocol designed for a permissionless environment, and exhibits very good performance improvements over Bitcoin. In a nutshell, step ② in this algorithm uses verifiable random functions to sample nodes across the whole system and elect a small committee. This committee is meant to act, as a whole, in the role of the leader. Algorand does not assign voting power based on identities (as PBFT does), nor based on computational power (as done in Bitcoin), but instead associates a weight with each node in the system based on the amount of money that node possesses. This approach is known as proof-of-stake.

Step ③ in Algorand takes the form of a novel Byzantine agreement algorithm, called BA⋆. At a high level, this is a two-phase agreement protocol. Each phase comprises a series of steps (typically between 2 and 11) leading either to agreement among correct nodes or to a recovery subprotocol. Recovery is triggered in case agreement is not reached (e.g., because of a network partition) and a fork occurred. This subprotocol represents step ④, that is, reconciling conflicting views, and highlights the importance of this step in our general scheme.

To ensure safety, Algorand assumes that the network experiences bounded periods of asynchrony, and each such period is followed by a bounded period of synchrony [19]. The exact length of these periods is irrelevant; what matters to our discussion is that safety in Algorand relies on the existence of synchronous periods. A malicious adversary can therefore exploit the assumption on the length of the synchronous period towards subverting this system's safety.

Algorand is an example of a protocol lying in between PBFT and Bitcoin in terms of its indulgence. It is more A-indulgent than Bitcoin: a violation of synchrony assumptions is not sufficient to violate safety if all participants are honest (unlike in the case of Bitcoin). Algorand is also more B-indulgent than PBFT, in the sense that controlling the majority of nodes is not enough to subvert the system (a substantial fraction of the money is also necessary). To subvert Algorand's safety, two conditions must be met: (1) malicious nodes must control some part of the money (not necessarily a big fraction), and (2) the network must experience some asynchrony. Thus, Algorand is both less A-indulgent than PBFT and less B-indulgent than Bitcoin.

4.2 Bitcoin-NG

Bitcoin-NG [15] is an important protocol because it introduces the idea of decoupling leader election (step ②) from agreement on blocks (step ③). In the original Bitcoin protocol, as described in §3.2.2, these two steps are entangled: a single leader is elected via PoW and nodes may immediately commit on the leader's proposal (if they choose to do so), or may wait for some confirmations, but no additional mechanism is necessary towards commitment.

Using this decoupling strategy, Bitcoin-NG can reach superior throughput compared to Bitcoin. Clearly, however, the elected node has too much responsibility: on its own, this leader can throw the system into a state of inconsistency by introducing forks and allowing double-spending. Essentially, this system has very similar assumptions and vulnerabilities as Bitcoin, and thus we consider it a typical B-indulgent protocol.

4.3 ByzCoin

ByzCoin [24] is an instance of a hybrid blockchain protocol [30], as it combines PBFT-style with Bitcoin-style agreement. Other such protocols exist [12, 30]; we focus here on ByzCoin, but we believe that our conclusions equally apply to other hybrid algorithms.

The common motivation underlying hybrid protocols is that neither Bitcoin's agreement (based on PoW), nor PBFT-style agreement are perfect, but they are in a sense complementary to each other. The former is very slow but has the advantage of being resilient to Sybil attacks. The latter is faster, but performs optimally if running on a small set of nodes (e.g., 4 to 7 replicas), and has no in-built protection against Sybil attacks. Hybrid protocols combine these two styles of agreement in the hope of avoiding their individual pitfalls and merging their specific strengths.

Similarly to Bitcoin-NG, ByzCoin decouples leader election (step ②) from agreement on transactions (step ③). However, leader election in ByzCoin means electing a whole committee of nodes (not just a single one). This committee runs PBFT, which is used to quickly serialize new transactions.

To be part of a committee, a node must pass a simple requirement: run PoW and be one of the last w nodes to find a solution. The parameter w, called a "share window" [24], defines the number of successful miners that will form the committee. This parameter encapsulates an important trade-off, from the point of view of this protocol's guarantees. On one hand, smaller w means smaller committee and a bigger threat to safety, as it is more likely that an adversary with sufficient computational resources gains control of the committee—i.e., obtain a third of total shares in the PBFT committee—and hence subvert the system's safety, as we explained earlier (§3.2.1). On the other hand, bigger w means larger committees: less potential for an adversary to control the committee, but increased risk of losing liveness if a third of the nodes in the committee are unresponsive, e.g., by being inactive at certain times or just leaving the network.

From the point of view of indulgence, ByzCoin is a combination of an A- and B-indulgent algorithm. The PBFT-like consensus algorithm ensures safety in periods of asynchrony, while the Bitcoin-like leader election protects against malicious node behavior such as Sybil attacks.

5. CONCLUSIONS

In this paper, we have presented a brief overview of several notable blockchain protocols, relating them to well-established concepts from distributed computing. We proposed a general scheme which unifies classical (PBFT-like) state machine replication protocols with the increasingly popular blockchain protocols. Our main goal was to shed light on the differences between these protocols. In doing so, we also pointed out the existence of two classes of protocols, defined in terms of how an adversary can go about subverting their safety. Asynchrony-indulgent protocols maintain their safety despite the harshest conditions of the network. A second class is that of malicious behavior-indulgent protocols, which maintain safety while tolerating big numbers of malicious nodes. We have shown how some protocols in the blockchain ecosystem are representatives of one class or another, or how they combine these two classes.

6. REFERENCES

[1] Confirmation – Bitcoin wiki. https://en.bitcoin.it/wiki/Confirmation.

[2] Ethereum project. http://www.ethereum.org.

[3] I. Abraham, G. Gueta, D. Malkhi, L. Alvisi, R. Kotla, and J.-P. Martin. Revisiting fast practical byzantine fault tolerance. 2017.

[4] I. Abraham and D. Malkhi. Bvp: Byzantine vertical paxos. 2016.

[5] B. Alpern and F. B. Schneider. Defining liveness. *Information processing letters*, 21(4):181–185, 1985.

[6] L. Alvisi, A. Clement, A. Epasto, S. Lattanzi, and A. Panconesi. Communities, random walks, and social sybil defense. *Internet Mathematics*, 10(3-4), 2014.

[7] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM (JACM)*, 42(1), 1995.

[8] C. Badertscher, U. Maurer, D. Tschudi, and V. Zikas. Bitcoin as a transaction ledger: A composable treatment. In *Eurocrypt*, 2017.

[9] A. Bessani, J. Sousa, and E. E. Alchieri. State machine replication for the masses with BFT-SMaRt. In *IEEE/IFIP DSN*, 2014.

[10] C. Cachin and M. Vukolić. Blockchains consensus protocols in the wild (keynote talk). In *DISC*, 2017. arXiv preprint arXiv:1707.01873.

[11] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4), 2002.

[12] C. Decker, J. Seidel, and R. Wattenhofer. Bitcoin meets strong consistency. In *Proceedings of the 17th International Conference on Distributed Computing and Networking*, page 13, 2016.

[13] C. Delporte-Gallet, H. Fauconnier, and R. Guerraoui. Tight failure detection bounds on atomic object implementations. *JACM*, 57(4), 2010.

[14] J. R. Douceur. The sybil attack. In *International Workshop on Peer-to-Peer Systems*, pages 251–260. Springer, 2002.

[15] I. Eyal, A. E. Gencer, E. G. Sirer, and R. Van Renesse. Bitcoin-NG: A Scalable Blockchain Protocol. In *NSDI*, 2016.

[16] I. Eyal and E. G. Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *International conference on financial cryptography and data security*, pages 436–454. Springer, 2014.

[17] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *JACM*, 32(2), 1985.

[18] J. Garay, A. Kiayias, and N. Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Eurocrypt*, 2017.

[19] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *SOSP*, 2017.

[20] A. Gottlieb and C. P. Kruskal. Coordinating Parallel Processors: A Partial Unification. *SIGARCH Comput. Archit. News*, 9(6), 1981.

[21] R. Guerraoui. Indulgent algorithms (preliminary version). In *PODC*, 2000.

[22] E. Heilman, A. Kendler, A. Zohar, and S. Goldberg. Eclipse attacks on bitcoin's peer-to-peer network. In *USENIX Security Symposium*, 2015.

[23] M. Herlihy. Wait-free synchronization. *ACM TOPLAS*, 13(1), 1991.

[24] E. K. Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, and B. Ford. Enhancing bitcoin security and performance with strong consistency via collective signing. In *USENIX Security*, 2016.

[25] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM TOPLAS*, 4(3), 1982.

[26] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed computing*, 11(4), 1998.

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

[28] D. Ongaro. *Consensus: Bridging theory and practice*. PhD thesis, Stanford University, 2014.

[29] R. Pass, L. Seeman, and A. Shelat. Analysis of the blockchain protocol in asynchronous networks. In *Eurocrypt*, 2017.

[30] R. Pass and E. Shi. Hybrid consensus: Efficient consensus in the permissionless model. *Cryptology ePrint Archive*, 2017. http://eprint.iacr.org/2016/917.pdf.

[31] M. C. Pease, R. E. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980.

[32] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[33] M. Vukolić. The Quest for Scalable Blockchain Fabric: Proof-of-work vs. BFT Replication. In *International Workshop on Open Problems in Network Security*, pages 112–125. Springer, 2015.