Author: Miguel Correia
Book title: Essentials of Blockchain Technology
Chapter 3
Editor: CRC Press
Year: 2019

# From Byzantine Consensus to Blockchain Consensus

## CONTENTS

## 1.1 Introduction

Blockchain is an exciting new technology that is making headlines worldwide. The reasons behind the success of a technology are often unclear, but in the case of blockchain it is safe to say that an important factor is that is has two killer apps, not a single one. The first killer app are cryptocurrencies, as the *original blockchain* is the core of Bitcoin [128], the first cryptocurrency and the one that is fostering the adoption of cryptocurrencies. The second killer app are *smart contracts*, first introduced in the Ethereum system [40], with their promise of computerizing legal contracts and of supporting a countless number of applications [161, 153, 90]. Moreover, the sky seems to be the limit for the applications people are imagining for blockchain.

A blockchain is essentially a secure, unmodifiable, append-only, log of transactions. The word *transaction* should be taken in a broad sense; in Bitcoin a transaction is a transfer of currency between accounts, but in smart contracts transactions do not necessarily involve money. Blockchains trade performance and resource-usage efficiency for security, in the sense that they are implemented by a set of redundant nodes that store the same state and run an algorithm to make *consensus* over the order of transactions (more precisely of blocks of transactions), even if some of the nodes misbehave in some way.

### Byzantine consensus

This last aspect —replication plus consensus despite misbehavior— is a topic of research since the late 1970s [133, 105]. The problem of reaching consensus in such conditions has been first proposed by Pease, Shostak, and Lamport in 1980 [133], but popularized by the same authors when they explained it as a story of Byzantine generals that have to agree on a common attack plan by exchanging messages [105]. The problem considers the existence of *arbitrary faults*, i.e., some nodes deviating from the algorithm they are supposed to execute, both in the domain of time (e.g., delaying messages or stopping to communicate) and the domain of value (e.g., sending wrong values in messages) [21], but that later work led to the term *Byzantine faults* being used to mean the same. These works considered a fully connected network (all nodes can communicate with all), but Dolev generalized the model to consider a mesh network, in which faulty nodes may corrupt and discard messages [73]. Consensus can be used to replicate a service in a set of nodes, in such a way that clients of that service observe a correct (non-faulty) service even if some nodes are faulty [144].

These earlier works consider a synchronous model, i.e., they assume that the communication and processing delays are bounded (they do not state it this way, but they assume it is possible to know if a message was not received, which is equivalent). However, this kind of model is inadequate for most distributed systems due to the uncertainty of communication delays in the Internet and of processing in typical nodes (workstations, mobile devices, servers, etc.). Moreover, they had a theoretical vein. In the 1990s there was a line of work on algorithms for reliable communication between nodes assuming an asynchronous model, i.e., that there are no time bounds. Notable examples are the work by Reiter et al. [141, 140] and by Kihlstrom et al. [96,

97]. This work has lead to the first efficient asynchronous Byzantine fault-tolerant replication algorithm, often designated PBFT, due to Castro and Liskov [47]. After PBFT, many others appeared [63, 102, 51, 146, 14, 53, 52, 145, 15, 94, 30, 137, 25], including our own [57, 156, 157, 158, 129]. All these algorithms provide well-defined safety and liveness properties [13], under well-defined assumptions. We will designate the problem solved by these algorithms informally by *Byzantine consensus*.

### Blockchain

Interestingly, despite all this research effort, Satoshi Nakamoto, the author or authors of Bitcoin, decided to use an entirely different approach to achieve consensus in the system that implements that cryptocurrency [128]. This consensus algorithm, often designated *Nakamoto consensus*, is based on the notion of *proof-of-work* (PoW). Nodes flood the network with transactions, which they collect and add to blocks. When a block is appended to the chain (in the log), nodes close the block they were creating and start solving a cryptopuzzle in order to obtain a PoW for their block. When a node solves the cryptopuzzle, it broadcasts its block with the PoW, which is validated then appended by all nodes to the chain in the other nodes. The time needed to solve the cryptopuzzle throttles the addition of blocks to the chain, which is a requirement when there are may competing nodes, that may pretend to be even more (a Sybil attack [74]). Moreover, the randomness of the time to solve it provides a sort of consensus, as it supports the assumption that no two nodes will obtain PoWs concurrently. However, this process does allow two different PoWs to be obtained concurrently, leading different subsets of the network of nodes to append different blocks to the chain, breaking agreement, which is an important property of any consensus algorithm. This is a clear disadvantage of Nakamoto consensus in relation to Byzantine consensus algorithms. On the positive size, the Nakamoto consensus is more scalable, as it can be used on top of a mesh network, using a peer-to-peer dissemination protocol, as in the case of Bitcoin [128, 70, 67, 159].

The potential of the blockchain technology has lead to an increase on the research in the area and on the actual implementation of blockchain systems and applications. In terms of research, a large number of papers have been published on improved consensus algorithms [99, 29, 87, 149], blockchain scalability [67, 159, 95, 101], attacks against cryptocurrencies and blockchains [80, 148, 116, 19], among many other topics. In relation to implementations, for example there are now more than 2,000 cryptocurrencies with a global value of more than 100 billion euros (around 50% due to Bitcoin) [54], and in 2016 during a period of a few months there were around 15,000 smart contracts deployed in Ethereum [110].

### This chapter

This chapter presents a state of the art in the area of Byzantine consensus and its application in blockchains. The chapter is organized as follows. Section 1.2 presents the first part of the state-of-the-art, about Byzantine consensus. Section 1.3 presents the second part, on blockchain based on the Nakamoto consensus and related schemes.

Section 1.4 presents the third part, about blockchain based on Byzantine consensus. Finally, Section 1.5 concludes the chapter.

## 1.2 Byzantine Consensus

This section presents a state-of-the-art in what we call *Byzantine consensus*, which excludes the Nakamoto consensus based on PoW. As already mentioned, research in the area started in the late 1970s [133, 105, 73] with synchronous algorithms and evolved to asynchronous algorithms later. This section is about more recent work, starting with the FLP impossibility result, then with practical algorithms starting with PBFT.

### *1.2.1 On System Models*

Consensus algorithms depend strongly on the system model, i.e., on the assumptions made about the environment and the system. In this chapter we consider a basic system model: message-passing communication; Byzantine (or arbitrary) faults; asynchrony. We say that this is the *basic* system model because all the algorithms considered refine it somehow.

The reason for this model is that it expresses well the conditions that exist in today's distributed systems such as those based on the Internet:

- *Message-passing* is a convenient communication model as in the Internet, and any other modern network, communication is broken-down into some sort of messages: packets, datagrams, application-layer messages, etc. Even network technologies that use virtual circuits in the lower layers (e.g., SDH/SONET and ATM), are used to transmit messages in the upper layers. Except when noticed, we consider that communication is done using *authenticated channels* that authenticate messages (prevent impersonation of the sender) and ensure their integrity (detect and discard modified messages). These characteristics are easy to implement with protocols such as SSL/TLS [71]. An alternative system model would be *shared memory* [20, 83, 111, 11, 32], but in the Internet the shared memory itself would have to be implemented using message-passing algorithms.

- Assuming *Byzantine faults* means to make no assumptions about how individual nodes fail. Poor assumptions may be vulnerabilities, so making no assumptions about faults is convenient when the objective is to make a system dependable and secure. Byzantine faults can be intentional, malicious, so tolerating such faults allows improving the security of the system, e.g., of the blockchain. A particularly pernicious subclass of Byzantine faults are inconsistent value faults that happen, e.g., when a faulty process sends two messages with the same identifier and different contents to two subsets of processes.

- *Asynchrony* means also to make no assumptions, but about bounds on communication and processing delays. This non-assumption is also convenient because, otherwise, adversaries might do attacks that cause delays on purpose. In fact, attacks against time, namely denial-of-service attacks, are very common because they tend to be easier to do than attacks against integrity or confidentiality.

A system model that considers Byzantine faults and asynchrony is very generic in the sense that algorithms designed for this model are correct even if the environment is more benign, e.g., if nodes can only crash or delays are bounded. Unfortunately, consensus is not solvable in this basic model, a problem that we explain in Section 1.2.3.

### 1.2.2 Byzantine Consensus Definitions

There is no single definition of Byzantine consensus, also denominated *Byzantine agreement*. In fact there are many, with significative differences.

Consider the basic system model above and that the consensus algorithm is executed by a set of $n$ nodes or *processes*. We say that a process is *correct* if it follows its algorithm until termination, otherwise it is said to be *faulty*. We assume that there are at most $f ¡ n$ faulty processes. Each process *proposes* a value (sometimes called the *initial value*) and *decides* a value.

Two common and similar definitions of consensus are *binary consensus* and *multi-valued consensus*. They differ in terms of the range of admissible value, respectively, binary or arbitrary. Otherwise the definition is similar [27, 139, 77, 112, 97]:

- *Validity:* If all correct processes propose the same value $v$, then any correct process that decides, decides $v$.
- *Agreement:* No two correct processes decide differently.
- *Termination:* Every correct process eventually decides.

The first two properties are *safety properties*, i.e., properties that say that bad things cannot happen, whereas the last is a *liveness property* that states that good things must happen [12]. A weaker validity property is the following [76, 75, 23]:

- *Validity':* If a correct process decides $v$, then $v$ was proposed by some process.

Although Validity is stronger than Validity', it does not say much about the value decided in case not all correct processes propose the same value. This limitation lead to the definition of *vector consensus* in which processes decide on the same vector with one value per process, for at least *n-f* processes [76, 131]. This definition of consensus is related to the *interactive consistency* problem, which however considered a synchronous system model [133]. In terms of definition, the difference between vector consensus and multi-valued consensus is the validity property that becomes:

- *Vector validity:* Every correct process that decides, decides on a vector $V$ of size $n$:
  - $\forall_{p_i}$: if $p_i$ is correct, then either *V[i]* is the value proposed by $p_i$ or $\perp$;
  - at least *f+1* elements of $V$ were proposed by correct processes.

A different solution to the same difficulty with the validity properties is given by the *validity predicate-based consensus* [65]. We will come back to this definition as it was written with blockchains in mind. The definition is similar to the previous ones, but the validity property depends on an application-specific *valid()* predicate:

- *Predicate-based validity:* If a correct process decides $v$, then $v$ satisfies the *valid()* predicate.

These definitions of consensus consider that all processes play the same role: all propose a value and all decide a value (at least if they are correct). Lamport introduced an alternative definition in an algorithm known as Paxos [103, 104], which has been thoroughly studied and modified [106, 163, 113, 63]. In Paxos, processes play one or more of the following roles: *proposers*, which propose values; *acceptors*, which choose the value to be decided; and *learners*, which receive the chosen value. The problem can be defined in terms of five properties [104, 113]:

- *Safety 1:* Only a value that has been proposed may be chosen.
- *Safety 2:* Only a single value may be chosen.
- *Safety 3:* Only a chosen value may be learned by a correct learner.
- *Liveness 1:* Some proposed value is eventually chosen.
- *Liveness 2:* Once a value is chosen, correct learners eventually learn it.

This definition is related to *state machine replication* (SMR) or the state machine approach [144, 31], but first let us introduce *atomic broadcast* or total order broadcast. Atomic broadcast is a problem that is different from consensus, but the two have been shown to be equivalent in several system models [35, 44, 58, 89]. The problem essentially states that all processes deliver the same messages in the same order, which is equivalent to running a sequence of consensus instances to decide what message(s) to deliver next. Atomic broadcast can be defined in terms of four properties [61]:

- *Validity:* If a correct process broadcasts a message $m$, then some correct process eventually delivers $m$.
- *Agreement:* If a correct process delivers a message $m$, then all correct processes eventually deliver $m$.
- *Integrity:* For any identifier *id* and sender $p$, every correct process $q$ delivers at most one message $m$ with identifier *id* from sender $p$, and if $p$ is correct then $m$ was previously broadcast by $p$.

- *Total order:* If two correct processes deliver two messages $m_1$ and $m_2$, then both processes deliver the two messages in the same order.

State machine replication is related to Paxos because it involves two kinds of processes: clients that make requests and receive replies (similarly to Paxos' proposers and learners); servers that provide a service to the clients and do consensus about the order of execution of the requests (similarly to Paxos' acceptors). This approach involves ordering requests using an atomic broadcast protocol, which, as explained, is equivalent to consensus. Consider a *state machine* that provides a service and that is characterized by a set of state variables that define its state, and by a set of commands that modify the state variables. All correct servers follow the same history of states if four properties are satisfied:

- *Initial state:* All correct servers start in the same state.
- *Agreement:* All correct servers execute the same commands.
- *Total order:* All correct servers execute the commands in the same order.
- *Determinism:* The same command executed in the same initial state in two different correct servers generates the same final state.

If these properties are satisfied, the service is correct as long as no more from $f$ servers are faulty. The relation between $n$ and $f$ depends on the algorithm, e.g., it can be $n \geq 3f+1$ [47, 63, 102, 14] or $n \geq 2f+1$ [57, 51, 157, 158]. SMR is a problem different from consensus, but we will abuse the language and call it *consensus* as it involves solving consensus. We prefer to use this term because it is common in the blockchain domain.

The Paxos definition of consensus is interesting because it allows a simple implementation of SMR in the crash failure model [144, 104]. However, in the Byzantine failure model this is more complicated because faulty processes can deviate from the algorithm arbitrarily [47, 113].

There are several other consensus variants. In the *k-set consensus* problem, correct processes can decide at most $k$ different values [50, 68]. The *Byzantine generals with alternative plans* problem takes into account the fact that processes may have several views about what decisions/actions are acceptable and unacceptable [55]. Each process has a set of good decisions and a set of bad decisions. The problem is to make all correct processes agree on good decisions proposed by a correct process, and never on a bad decision.

All these definitions of consensus consider that the algorithm is executed by a fixed set of $n$ known processes. There are a few works on consensus in dynamic systems in which the set of processes is unknown and varying [126, 7, 10, 9]. The single Byzantine algorithm of this kind, BFT-CUP, is based on an oracle called participant detector that provides hints about the active processes [10, 9].

Table 1.1 presents a summary of the consensus definitions.

| Definition | Characteristics | References |
|---|---|---|
| binary consensus | agreement about a binary value | [27, 139] |
| multi-valued consensus | agreement about an arbitrary value | [77, 112, 97] |
| vector consensus | agreement about a vector with values from at least *n-f* processes | [76, 131] |
| interactive consistency | similar but for synchronous system models | [133] |
| validity predicate-based consensus | validity property depends on an application-specific *valid()* predicate | [65] |
| Paxos | multi-valued consensus with 3 roles: proposers, acceptors, learners | [103, 104, 106, 163, 113, 63] |
| atomic broadcast | processes broadcast and deliver the same messages in the same order | [35, 44, 58, 89, 61] |
| state machine replication | servers implement a replicated service; clients request that service | [144, 31, 57, 51, 157, 158] |
| k-set consensus | correct processes can decide at most *k* different values | [50, 68] |
| consensus with unknown processes | for dynamic systems in which the set of processes is unknown | [126, 7, 9, 10, 9] |

**Table 1.1: Definitions of consensus and related problems.**

### *1.2.3 FLP Impossibility*

A problem with consensus in the basic system model is that it is not solvable. This fact derives trivially from an impossibility result known as FLP, after the names of its proponents [82]. FLP considers binary consensus and a different, weaker, system model (let us call it the FLP system model). This system model also considers message-passing communication and asynchrony, but only that a single process can fail simply by crashing (neither any process, nor arbitrarily). It also excludes the existence of random numbers so the statement is actually about deterministic algorithms. An intuition of this result is that the combination of uncertainty in terms of time (asynchrony) and uncertainty in terms of failure (a process may fail) does not allow an algorithm to distinguish if a process is slow or faulty.

This result is inconvenient because it requires modifying the system model, but, interestingly, has also fostered research on consensus. Specifically, there has been a lot of research on system models that are similar to the FLP system model and the basic system model and allow solving consensus. These new conditions in which the problem is solvable are often said to *circumvent* FLP, whereas, in fact, they change its premises.

The main ways of circumventing FLP are the following [62]:

1. *Add time assumptions to the model*, thus partially sacrificing asynchrony. The idea is to add these assumptions in a way that is realistic in the Internet and other networks. Dwork et al. presented two partial synchrony models that add such time assumptions and allow solving consensus [77]. A partial synchrony model captures the intuition that systems may behave asynchronously (i.e., with variable/unknown processing/communication delays) for some time interval, but that they tend to eventually stabilize. Therefore, the idea is to let the

system be mostly asynchronous but to make assumptions about timing properties that are eventually satisfied. Algorithms based on this model are typically guaranteed to terminate only when these timing properties are satisfied. Chandra and Toueg proposed a third partial synchrony model that is similar but weaker [49]: for each execution, there is an unknown global stabilization time GST, such that an unknown bound on the message delivery time $\Delta$ is always satisfied from GST onward. When designing PBFT, Castro and Liskov used an even weaker model in which delays are assumed not to grow exponentially forever [47]. This last model has been adopted by many Byzantine consensus and SMR algorithms [63, 53, 52, 156]. NewTOP [127] and XPaxos [109] consider stronger time assumptions, respectively that pairs of nodes and correct replicas can communicate within a known delay $\Delta$.

2. *Add oracles to the model* that provide hints about process failure, thus partially sacrificing asynchrony. This idea was introduced by Chandra and Toueg [49]. The FLP result derives from the impossibility of distinguishing if a process is faulty or simply very slow, therefore, intuitively, having a hint about the failure/crash of a process may be enough to circumvent FLP. The idea is to associate an unreliable failure detector (UFD) to each process, which provides hints about other processes failures. Chandra and Toueg presented eights classes of UFDs based on properties of accuracy and completeness [49]. They also proved that extending the FLP system model with a rather weak UFD was enough to solve consensus [48]. These oracles are an elegant construct but, at the end of the day, they hide time assumptions that are necessary to implement them (the system cannot be asynchronous). There are other oracles that allow solving consensus and are not failure detectors, e.g., the $\Omega$ detector, which provides hints about who is the leader process [48], and ordering oracles, which provide hints about the order of messages broadcasted [136].

3. *Use a hybrid system model* that includes a subsystem with stronger time assumptions, again partially sacrificing asynchrony. A *wormhole* is an abstraction that system-wise is a component of the system, but model-wise is an extension to the system model [154, 155]. The first work on wormholes to solve consensus [56] considers the basic system model extended with a wormhole called *Trusted Timely Computing Base* (TTCB) [60]. The TTCB is a secure, real-time, and fail-silent distributed component, which provides enough timeliness to circumvent FLP. Applications implementing the consensus algorithm run in the normal system, i.e., in the asynchronous Byzantine system, but use the services provided by the wormhole. In this case, the consensus algorithm relies on the Trusted Block Agreement Service, which essentially makes an agreement on small values (typically hashes) proposed by a set of processes. Later, a simpler multi-valued consensus algorithm and a vector consensus based on the TTCB were also proposed [130, 131]. There are other consensus and SMR algorithms based on wormholes, although most authors do not use the term "wormhole" [57, 59, 51, 61, 157, 158, 94, 26].

4. *Add randomization to the model*, partially sacrificing determinism. FLP applies to deterministic algorithms, so a solution to circumvent this result is to add randomization to the system model —the ability to generate random numbers— and design probabilistic algorithms [28, 45, 46, 84, 139, 151, 43, 58, 122, 125]. This involves changing one of the properties that define consensus and make it probabilistic.

These ways of circumventing FLP are also a form of classifying algorithms in the area. Table 1.2 presents a summary of the techniques to circumvent FLP.

| Technique | Positive / Negative | References |
|---|---|---|
| add time assumptions | algorithms efficient if network stable / delayed otherwise | [77, 49, 47, 63, 53, 52, 156] |
| add oracles | same as previous | [49, 48, 136] |
| hybrid model | efficient algorithms / additional assumptions | [60, 130, 131, 57, 59, 51, 61, 157, 158, 94] |
| allow randomization | no assumptions / less efficient algorithms | [28, 45, 46, 84, 139, 151, 43, 58, 122, 125] |
| | | [123, 124, 118, 119] |

**Table 1.2: Techniques to circumvent FLP.**

### *1.2.4 Byzantine Consensus Patterns*

The consensus algorithms mentioned in the previous section follow two major communication patterns: *decentralized* and *leader-based*. In this section we use these patterns to present briefly how Byzantine consensus algorithms work. Moreover, this will become useful later to understand blockchain consensus algorithms.

The two patterns are represented in Figure 1.1. In decentralized consensus algorithms, all processes play the same role and try to individually reach a decision. In leader-based (or coordinator-based or primary-backup) consensus algorithms, there is a leader (or primary or coordinator) that tries to impose a decision; if the leader is faulty, a new leader has to be elected. The fact that in decentralized consensus algorithms all processes communicate with all others imposes a quadratic message complexity ($O(n^2)$). On crash fault-tolerant leader-based consensus algorithms it is possible to achieve linear message complexity ($O(n)$) [49, 92, 143], but in Byzantine leader-based consensus algorithms the need to deal with inconsistent value faults imposes a quadratic message complexity.

Decentralized consensus algorithms work basically the following way (Figure 1.1(a)). In the first step, every process broadcasts a proposal, and waits for $n$-$f$ messages. In the second step, every process picks a proposal with enough votes or a default value, broadcasts it, and waits for $n$-$f$ messages. In the third step every pro-

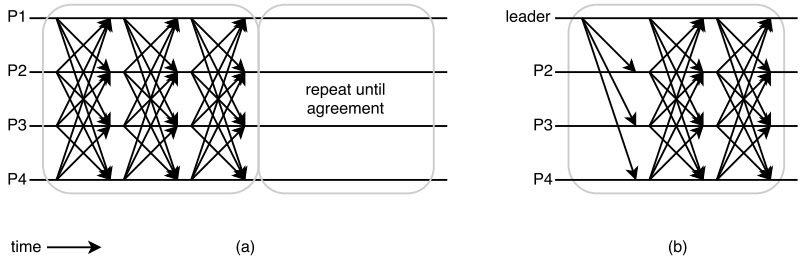


**Figure 1.1: The two main organization patterns of Byzantine consensus algorithms: (a) decentralized and (b) leader-based.**

cess picks a proposal if it has enough votes, broadcasts it, waits for *n-f* messages, and decides the value if it received enough copies. Otherwise, the whole process is repeated. This explanation and the figure are based on Bracha's algorithm [34]. In that algorithm the broadcast primitive is not standard, unreliable, network broadcast, but *reliable broadcast* [34, 89]. The reliable broadcast problem consists in guaranteeing that when a process sends a message, all processes deliver that message, or possibly no message at all if the sender is faulty. This problem is weaker than atomic broadcast (no ordering) and also weaker than consensus, but requires 3 communication steps and $O(n^2)$ messages in the basic system model. Therefore, there are many more messages being actually sent than those represented in the figure; Bracha's algorithm has $O(n^3)$ message complexity.

Leader-based consensus algorithms work essentially as shown in Figure 1.1(b). In the first step, the leader broadcasts a proposal and in the following two steps the other processes agree on accepting it and confirm their acceptance. This pattern corresponds to the normal mode operation of PBFT, omitting the interaction with the clients (recall that PBFT is a SMR algorithm). In that algorithm, as in many others, in case the leader is suspected, a new leader may have to be elected (also not represented in the figure). There is another class of leader-based consensus algorithms that rotate the leader, so there is no need for electing a new one, including some our own [77, 156, 52, 157, 61].

### *1.2.5 Hybrid Models to Reduce Processes*

After defining consensus and explaining the main ways to circumvent FLP, we present two areas in which we did much work related to the topic (this and the next section).

Most Byzantine consensus and SMR algorithms consider the basic system model extended with time assumptions (e.g., the same as PBFT [47]) or failure detectors (1 and 2 in Section 1.2.3). In these models, the relation between the number of processes $n$ and the maximum number of faulty processes $f$ is $n \geq 3f+1$ [47, 63, 102, 14]. This means that 4 processes are needed to mask 1 that is faulty, 7 to mask 2, and so on. In

the 1990s and 2000s these numbers were considered high, and the folklore in the area said that many companies refused to use such algorithms due to the costs involved. At the time many companies used crash fault-tolerant algorithms, which required only $n \geq 2f+1$, e.g., as part of Chubby [38] or Zookeeper [91]. Interestingly the concern with the number of replicas vanished with Bitcoin and blockchain (Section 1.3), but reducing the number of processes is clearly an important goal. Notice that in SMR, more processes (server replicas in that context) mean more hardware, more software licences, and more administration costs. Moreover, it is important to avoid common mode failures, which requires different replicas, i.e., diversity [108].

We were the first to show that it is possible to reduce the minimum number of replicas in Byzantine SMR from $3f+1$ to $2f+1$ using a hybrid failure model (3 in Section 1.2.3) [57, 59]. That algorithm, BFT-TO, is based on a wormhole called Trusted Ordering Wormhole (TO wormhole).[1] This wormhole is distributed (has a component in each replica) and provides an ordering service, so it solves consensus. It has to be implemented in a way that it satisfies two security properties: integrity (its service and data cannot be tampered with by an adversary) and confidentiality (of the cryptographic material used to protect the communication). Therefore, the TO wormhole fails only by crashing, so it can solve consensus with only $2f+1$ replicas, e.g., using Schiper's algorithm [143]. BFT-TO uses the TO wormhole to order hashes of the requests, so it is not bound by the $n \geq 3f+1$ relation of other Byzantine consensus algorithms.

A few years later, Chun et al. proposed the *Attested Append-Only Memory* (A2M), another wormhole used to implement state machine replication with only $2f+1$ replicas [51]. This Byzantine SMR algorithm was called A2M-PBFT-EA. Like the TO wormhole, A2M has to be tamperproof, but it is local to the computers, not distributed, which is a significative improvement. Replicas using the A2M are forced to commit to a single, monotonically increasing sequence of operations. Since the sequence is externally verifiable, faulty replicas cannot present different sequences to different replicas.

Later, we presented an even simpler wormhole that also allows implementing state machine replication with only $2f+1$ replicas, the *Unique Sequential Identifier Generator* (USIG) [157, 158]. This component contains only a counter and a few cryptographic functions that are used to associate sequence numbers to certain operations done by the replicas, e.g., producing a signed certificate that proves unequivocally that the number is assigned to that message. The USIG has been used to implement three SMR algorithms: MinBFT and MinZyzzyva that are inspired on PBFT and Zyzzyva [102] but with less replicas [158]; and EBAWA that uses a rotating leader and has other characteristics adequate for wide-area networks [157]. The USIG is also the basis for a methodology to transform consensus algorithms that tolerate crash faults and require $2f+1$ processes, into similar algorithms that tolerate Byzantine faults also with $2f+1$ processes [61].

We implemented the USIG as a thin layer on top of the Trusted Platform Module

[1] This explanation is based on the most recent, more refined, version of the algorithm [59], instead of the original [57].

(TPM), a security chip designed by the Trusted Computing Group and now present in the mainboard of mosts PCs [152, 158]. Our implementation was based on one of the TPM's monotonic counters and signatures. Abraham et al. presented a Byzantine consensus based on the TPM that uses one of the TPM's Platform Configuration Registers (PCRs) instead of the monotonic counter [5]. Our experiments have shown that using the TPM was a bad option, as the signature implementation was quite slow (approximately 0.4s to obtain a signature). Both USIG and Abraham et al.'s abstraction are similar to Trinc [107].

Kapitza et al. implemented our USIG service on an FPGA and called it Counter Assignment Service in Hardware (CASH) [94]. Then, they designed CheapBFT, a Byzantine SMR architecture and algorithm that tolerates that all but one of the replicas active are faulty in normal-case operation. CheapBFT runs only $f+1$ replicas in normal-case operation and keeps the other $f$ replicas on hold; it activates these $f$ replicas in case there is disagreement between the active replicas because one or more are faulty. In total it uses $2f+1$ replicas, similarly to MinBFT in which it is based. This reduction of the number of replicas is interesting, but comes at the cost of having a trusted infrastructure to detect disagreement between replicas and activate the replicas on hold, which is much more complex than the USIG service.

Hybster parallelizes consensus instances to be able to order more than 1 million operations per second, increasing one order of magnitude the performance of algorithms in the area [26]. That scheme is based on TrInX, a software version of USIG/Trinc/CASH implemented in Intel SGX enclaves [114, 16]. These enclaves are trusted execution environments that run software components isolated from the rest of the platform, including the operating system. This isolation is enforced by the CPU itself. TrInX is different from similar components in two ways. First, for parallelization purposes, it can implement an arbitrary number of counters and provide multi-counter certificates. Second, for performance, it uses message authentication codes based on cryptographic hash functions instead of digital signatures.

Table 1.3 summarizes the works on using hybrid models to reduce the number of processes.

| **Algorithm** | **Subsystem** | **Characteristics** | **References** |
|---|---|---|---|
| BFT-TO | TO wormhole | distributed subsystem | [57, 59] |
| A2M-PBFT-EA | A2M | local subsystem | [51] |
| MinBFT, MinZyzzyva, EBAWA | USIG | local and simple subsystem | [157, 158] |
| | TPM / PCRs | based on common hardware, slower | [5] |
| CheapBFT | CASH | hardware implementation of USIG | [94] |
| Hybster | TrInX | alternative implementation of USIG | [26] |

**Table 1.3: Byzantine consensus algorithms based on hybrid models.**

### *1.2.6 Randomization*

Another area in which we did significant work was on solving Byzantine consensus using randomization. As mentioned in Section 1.2.3, adding randomization to the model allows circumventing FLP and involves making one of the properties that defines the consensus problem probabilistic, instead of deterministic. Most authors choose to make a liveness property probabilistic (e.g., Termination), as the algorithm may run until this property is satisfied, so its probabilistic nature becomes irrelevant. The Termination property becomes:

- *Probabilistic termination:* Every correct process eventually decides with probability 1.

Curiously there are a few papers that sacrifice a safety property instead of liveness. Specifically, these algorithms satisfy an agreement property with a certain probability [151, 46]. This is arguably a bad idea, but is also the option taken in several blockchains, starting with Bitcoin's.

Randomized Byzantine consensus algorithms have been around since 1983 [27, 139]. All randomized consensus algorithms are based on a random operation, typically picking randomly 0 or 1, something that is often designated *tossing a coin*. Randomized consensus algorithms can be classified in two classes, depending on how the coin tossing operation is done:

- *Local coin*: algorithms in which each process tosses its own coin. The first algorithm is due to Ben-Or [27]. Local coin tossing is trivial to implement: it requires just a random number generator that returns binary values. However, when consensus algorithms based on local coins require tossing a coin, there tends to be disagreement about the coin values in each process, so these algorithms terminate in an expected exponential number of communication steps [27, 34].

- *Shared coin*: algorithms in which processes toss coins cooperatively. The first algorithm is due to Rabin [139]. Shared coin tossing is more complicated because it requires a distributed algorithm that provides the same coin value to all non-faulty processes. However, these algorithms can terminate in an expected constant number of steps, as all processes see the same sequence of coins tossed [28, 45, 46, 84, 139, 151].

Randomized consensus algorithms have raised a lot of interest from a theoretical point of view. However, on the practical side they have often been assumed to be inefficient due to their probabilistic nature, which leads to high *expected time complexity* (number of rounds to terminate) and high *expected message complexity* (number of messages sent). However, those assumptions miss the fact that consensus algorithms are not usually executed in oblivion, but in the context of a higher-level problem (e.g., atomic broadcast) that can provide a context that promotes fast termination (e.g., many processes proposing the same value can lead to a quick termination). Moreover, adversary models usually consider a strong adversary that completely controls

the scheduling of the network and decides which processes receive which messages and in which order, whereas in practice, an adversary is not able or interested in doing such attacks (it would be simpler to block the whole communication instead). Therefore, in practice, the network scheduling may also foster fast termination.

A second, more real, reason for inefficiency is that these algorithms are not executed alone but stacked to solve a relevant problem, such as SMR. Most randomized consensus algorithms in the literature solve binary consensus [28, 45, 46, 84, 139, 151], as the expected time complexity becomes much worse if the random number has more than one bit [81]. Therefore, to do something useful, we need *transformations*, e.g., from binary consensus and broadcast primitives into multi-valued consensus and from multi-valued consensus into vector consensus or atomic broadcast [43, 58, 122]. This stacking of algorithms can lead to large expected time and message complexities.

We have shown that randomized consensus can be efficient in two steps. As a first step, we developed a set of transformations from binary randomized consensus to multi-valued consensus, vector consensus and atomic broadcast [58]. The proposed transformations have a set of properties that we believed had the potential to provide good performance. Firstly, they do not use digital signatures constructed with public-key cryptography, a well-known performance bottleneck in such algorithms at the time. Secondly, they make no synchrony assumptions, since these assumptions are often vulnerable to subtle but effective attacks. Thirdly, they are completely decentralized, thus avoiding the cost of detecting faulty leaders that exists in leader-based protocols like PBFT [47, 156, 52]. Fourthly, they have optimal resilience, i.e., $n \geq 3f+1$. In terms of time complexity, the multi-valued consensus protocol terminates in a constant expected number of rounds, while the vector consensus and atomic broadcast protocols have $O(f)$ complexity.

The second step involved implementing minor variations of these algorithms and evaluating their performance experimentally [122, 125]. The algorithms were implemented as a protocol stack called RITAS. At the lowest level of the stack there are two broadcast primitives: reliable broadcast and echo broadcast (essentially weaker versions of atomic broadcast that do not require ordering or consensus). On top of these primitives, stands Bracha's binary consensus [34]. Building on the binary consensus layer, multi-valued consensus allows the agreement on values of arbitrary range. At the highest level there is vector consensus that lets processes decide on a vector with values proposed by a subset of the processes, and atomic broadcast that ensures total order. The algorithms have shown good performance, e.g., with latencies in the order of milliseconds and throughput of a few thousand messages per second delivered by the atomic broadcast primitive [125].

ABBA was a major step in randomized consensus algorithms [45]. This binary consensus algorithm is based on a shared coin created using a novel combination of cryptographic schemes, mainly threshold signatures and a threshold coin-tossing scheme, which has lead to constant expected time complexity ($O(1)$) and expected message complexity $O(n^2)$. This algorithm theoretically performs better than Bracha's (which uses local coins), but resorts to expensive public-key cryptography, so we made a thorough experimental comparison of the two [121]. In a nutshell,

Bracha's algorithm performed always better, but we considered only 4, 7, and 10 processes, and ABBA seemed to scale much better, so with a few more processes it would start performing better and stay that way.

When further experimenting with these algorithms on WiFi networks (802.11a/g at 54 Mb/s and 802.11b at 11 Mb/s), both with PDAs (circa 2006 there were no smartphones) and laptops, we observed that the performance was much worse than in wired networks, possibly due to the lower bandwidth [120]. Later, we concluded that the reasons were more profound and that the *basic system model* was inadequate for wireless networks, most especially wireless ad-hoc networks [123, 124]. In the basic system model every pair of processes is connected by an authenticated channel, so when a process broadcasts a message to all others —a common communication pattern in these algorithms— it actually sends *n-1* messages; as the medium is shared, it becomes occupied for the period of sending all these individual messages. Moreover, there are several effects in wireless networks that lead to retransmissions: loss of signal due to mobility, interference from other wireless networks, electromagnetic interference from other sources, reflection and blocking caused by objects. Therefore, the problem is not only that in theory the bandwidth is lower than in typical wired networks, but that in practice the available bandwidth is even further reduced.

This lead us to investigate an adequate system model for wireless networks [123, 124]. In this system model, processes broadcast messages to all others and there can be dynamic omission faults that prevent any number of processes from receiving the message sent in a round. Interestingly, Santoro and Widmayer have proved that, even with strong synchrony assumptions, there is no deterministic solution to any non-trivial form of agreement if *n-1* or more messages can be lost per communication round [142]. We proposed two binary consensus algorithms that circumvent this impossibility result using randomization. The first tolerated only crash faults [123, 124], so we focus on the second, Turquois, which tolerates Byzantine faults [118, 119]. Turquois works in rounds, similarly to Bracha's algorithm, but does progress only in rounds when processes receive enough messages. Moreover, it uses a novel message validation scheme that has two aspects: authenticity validation and semantic validation. Turquois was evaluated experimentally and proved to be much faster than ABBA and Bracha's algorithm.

Despite our work in the early 2010s, randomized consensus algorithms continued not to raise much practical interested until very recently with the appearance of HoneyBadgerBFT [117]. This algorithm is inspired on SINTRA, a protocol stack that builds on ABBA [43] (our own stack, RITAS, was inspired on SINTRA, but avoided public-key cryptography for efficiency). HoneyBadgerBFT solves atomic broadcast and manages to improve over SINTRA by resorting to an efficient transformation from multi-valued consensus to vector consensus (that they call common subset agreement) using batching. The algorithm solves atomic broadcast with time complexity *O(n)*.

Table 1.4 summarizes the works on randomized consensus.

| Algorithm / Stack | Characteristics | References |
|---|---|---|
| several with local coin | binary consensus | [27, 34] |
| several with shared coin | binary consensus | [28, 45, 46, 84, 139, 151] |
| SINTRA | protocol stack; builds on the ABBA binary consensus algorithm | [43] |
| RITAS | protocol stack; avoids public-key cryptography for efficiency | [58, 122] |
| HoneyBadgerBFT | efficient atomic broadcast based on an efficient transformation from multi-valued consensus to vector consensus | [117] |

**Table 1.4: Randomized Byzantine consensus algorithms and stacks.**

Summary

This section provided a brief state-of-the-art on Byzantine consensus. We started with a discussion on the system models that are relevant in this context, then presented several definitions of consensus protocols, including related notions such as state machine replication and atomic broadcast. Then, we presented the influential FLP impossibility result and the main approaches to circumvent it. Next, the organization patterns used in most algorithms are presented. Finally, we presented two areas in which we did significant work: using hybrid models to solve consensus with less processes, and efficient randomized consensus algorithms. The next section presents the notions of blockchain and Nakamoto consensus, as well as several related algorithms.

## 1.3 Blockchains with Nakamoto Consensus

As mentioned in the introduction, the original blockchain —the one at the core of Bitcoin— runs a consensus algorithm based on the notion of *proof-of-work* (PoW) [128], which we denominate *Nakamoto consensus*. This section presents the original blockchain, its consensus algorithm, the killer applications (cryptocurrencies and smart contracts), and variants of PoW-based consensus. It does not present more recent blockchains that use Byzantine consensus related to those presented in the previous section (Section 1.4).

### *1.3.1 Bitcoin's Blockchain and Consensus*

This section presents Bitcoin's blockchain and consensus. They were not introduced in a scientific paper, but on a white paper that presents how they work in an informal way [128]. In a sense, that paper presents a brilliant piece of work, which puts together several previously existing concepts to create a novel, highly successful, system. However, it misses important aspects such as a clearly defined system model, properties, communication protocol, and consensus algorithm. Moreover, the Bit-

coin system is not static, but a system in production, with frequent software updates and conceptual changes. Fortunately, there is more rigorous documentation from the Ethereum project [40, 160] (Ethereum in many aspects is very close to Bitcoin), from Garay et al. [85], and others [18].

Bitcoin's blockchain is a secure, unmodifiable, append-only, log of transactions, as mentioned before. This blockchain is executed in a set of nodes that run Bitcoin's software and form a *peer-to-peer network* (P2P). This network is decentralized, i.e., it has no central control or authority, although the team that develops that software has some control, at least as far as they manage to convince the nodes to install the new versions of the software (complete autonomy is a myth). The nodes run a consensus algorithm to decide which transactions to log. Every consensus decides the next *block* (or set) of transactions that is added to the log.

Each node stores locally a *chain of blocks* or blockchain (Figure 1.2). This chain is a linked list that contains the log itself. The first block is called genesis block. Each record contains the following items:

- *Version:* a non-trivial combination of version of the Bitcoin software used to create the block and a bitmask [162];
- *Hash previous block:* cryptographic hash (SHA-256) of the header of the previous block (i.e., the previous block except the transactions);
- *Hash Merkle root:* the cryptographic hash that is at the root of the Merkle tree that summarizes the transactions;
- *Time:* a timestamp that indicates when the block was created;
- *Nonce:* a number that has the property of making the cryptographic hash of the header to be below a certain threshold (related to PoW, explained later);
- *Transactions:* list of the block's transactions (the only item that is not part of the header).

A Merkle tree is a binary tree where the root and every node is a hash of its two children, except the leafs that are, in this case, transactions. The use of such a tree allows nodes to delete or not download the transactions that are not relevant to them, as the Merkle tree allows nodes to verify transactions individually (instead of needing all the transactions to check the hash of a block).

Notice that the consensus algorithm also defines the order in which the nodes are appended to the chain. Therefore, it would be more consistent with the original distributed algorithms nomenclature (Section 1.2) to say that the algorithm solves atomic broadcast or SMR. Nevertheless, we will stick to the term consensus, as it is the one that is used in the blockchain and cryptocurrency literature. One aspect in which the consensus used in blockchains differs from these two other problems is that each block depends on the previous, as it includes its hash.

There are several types of nodes in Bitcoin. Each type of node implements a subset of the following four *roles* [18]:

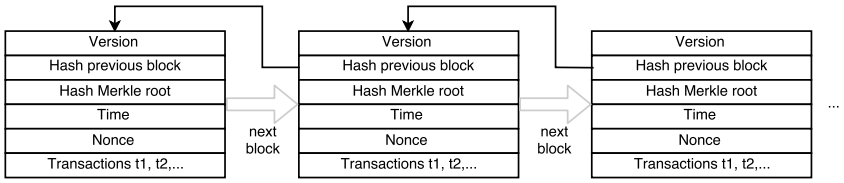


**Figure 1.2: Chain of blocks in Bitcoin's blockchain.**

- *Wallet:* storing the data necessary to access the coins (BTC) of the node's owner (including cryptographic material);
- *Miner:* solving the cryptopuzzle necessary to obtain PoWs for blocks;
- *Full blockchain:* storing the full chain of blocks;
- *Network routing:* implementing Bitcoin's P2P protocol, making the node part of the Bitcoin network.

When a node (with wallet and network routing roles) wants to do a transaction, it disseminates that transaction in the P2P network. All nodes (with mining role) collect transactions and add them to blocks. When a node (with mining role) appends a block to the chain, it closes the block it was creating and starts solving the cryptopuzzle in order to obtain a PoW for that block. A node appends a block to the chain when it receives a block that has the hash of the last block in the chain and a valid PoW.

The cryptopuzzle is the problem of finding a nonce that makes the hash of the block lower than a certain *target*, or, equivalently, to start with a number of 0-bit values. The target number is defined dynamically, in order to tune the difficulty of the problem (the target may be, e.g., $2^{187}$). This process of solving the cryptopuzzle is called *mining*, involves a huge amount of energy, and is compensated with two forms of incentives in coins: new minted coins created by the block itself (50 coins in the beginning of Bitcoin, but periodically reduced), and a transaction fee (payed by everyone that has a transaction in the block). Today (successful) mining is not done by individual nodes, but by large groups of nodes: *mining pools* where ad-hoc sets of nodes share mining work and split the reward [86] or *mining datacenters* dedicated to mining Bitcoin [135].

This algorithm is clearly not a classical Byzantine consensus as those presented in Section 1.2:

- *Validity* is ensured by the nodes that receive the block, which do not add it to the chain unless the two hashes are correct, the timestamp is within certain bounds, and the transactions satisfy certain conditions (e.g., they do not spend money already spent).

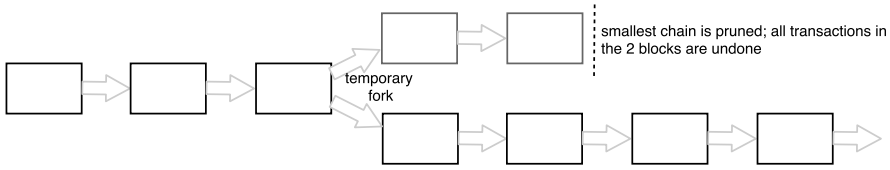


**Figure 1.3: Temporary fork in Bitcoin's blockchain.**

- *Agreement* can be violated, as the algorithm allows PoWs for two different blocks to be obtained concurrently by competing groups, leading different subsets of the network of nodes to append different blocks to the chain (blocks from different groups of nodes must be different, as the first transaction of the block is a special transaction that creates a number of coins to reward the effort of the group of nodes). Such *temporary forks* are healed by letting the chains grow and pruning the smallest subchain(s), effectively undoing the transactions in all blocks in these smallest chains since the fork point (Figure 1.3).

- *Termination* is only eventually satisfied because in case there is a temporary fork it has to be healed for consensus on a block to terminate. Moreover, Nakamoto claims that as long as a majority of the CPU power —*mining power*— is controlled by nodes that are not faulty, they will generate the longest chain and the properties of the blockchain will be assured [128]; however, Eyal and Sirer have shown an attack in which colluding miners obtain a revenue larger than their fair share, making it rational to collude and create a pool with more than half of the mining power [80].

Bitcoin's blockchain may be said to implement a *speculative* variation of *validity predicate-based consensus* (Section 1.2.2). There are several speculative versions of SMR algorithms [102, 88, 158, 138]. These algorithms reply faster to clients in normal conditions, but in some cases servers may have to rollback part of the execution. This is essentially what happens also with Bitcoin's blockchain, which may have to remove from the chain some previously added blocks, undoing many transactions.

This *speculative validity predicate-based Byzantine consensus* can be defined in terms of the following properties:

- *Predicate-based validity:* If a correct process decides $v$, then $v$ satisfies the *valid()* predicate.

- *Eventual agreement:* Eventually no two correct processes decide differently.

- *Termination:* Every correct process eventually decides.

In Bitcoin the *valid()* predicate checks if: the previous block referenced by the block exists; the timestamp of the block is greater than the timestamp of the previous

block and less than 2 hours in the future; the PoW of the block is valid; no transaction causes an error (tries to spend unavailable money or is wrongly signed).

Vukolic formalizes the necessity that consensus is not undone in terms of the following property, which is not satisfied by consensus algorithms based on PoW [159]:

- *Consensus finality:* If a correct node $p$ appends block $b$ to its copy of the blockchain before appending block $b'$, then no correct node $q$ appends block $b'$ before $b$ to its copy of the blockchain.

These properties define the consensus variation. A *blockchain* has to satisfy a set of additional properties, which are particularly challenging when consensus is based on PoW [33]:

- *Exponential convergence:* The probability of a fork of depth $d$ is $O(2^{-d})$.
- *Liveness:* New blocks containing new transactions will continue to be appended to the chain.
- *Correctness:* All blocks in the longest chain will only include valid transactions.
- *Fairness:* A miner with a ratio $\alpha$ of the total mining power will mine a ratio $\alpha$ of the blocks when time tends to infinite.

The first property is important to give users confidence that waiting for a number of blocks being appended to the chain after the one containing their transaction will ensure this transaction is permanently part of the chain.

As mentioned above, there are different types of Bitcoin nodes, that implement different subsets of the four roles. The most important types of nodes are the following [18]. A *reference client* implements the four roles, i.e., it provides the full functionality. A *full blockchain node* is a node that keeps a copy of the chain, so it implements the full blockchain and the network routing roles. A *solo miner* is a node that is concerned only with mining, so it implements all roles except the wallet. A *lightweight wallet* or *simplified payment verification node* is a basic user wallet that provides only the wallet and network routing roles. Notice that these are the original types of nodes; mining pools and mining datacenters have servers that play mainly two roles: miner and routing (using another protocol, typically Stratum [147]).

There is no trivial matching between these node types and SMR's client and server, but it is reasonable to say that nodes that implement the wallet and network routing roles correspond to clients, whereas nodes that implement the miner, full blockchain, and network routing roles correspond to servers. Interestingly, the reference node type includes all these roles, so such nodes are both clients and servers.

Figure 1.4 represents the Bitcoin's consensus communication pattern. Major differences in relation to Byzantine consensus patterns (Figure 1.1) are that there are less communication steps and servers do not communicate all-with-all. These differences are associated to a much lower message complexity, but also to a weakening of the agreement and termination properties.

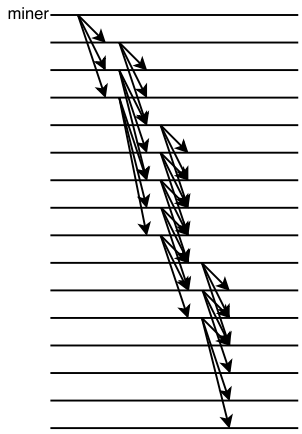


**Figure 1.4: Communication pattern in Bitcoin and other blockchains.**

This does not mean that Bitcon's blockchain performs well. Quite on the contrary, as the threshold of the cryptopuzzle is configured for miners to take 10 minutes to find a solution and there is a maximum of transactions per block (imposed by a maximum block size of 1 MB), the throughput is limited (7 transactions per second until recently). The latency is also huge due to those 10 minutes, plus the existence of a queue of about 1 hour of pending transactions, plus about 1 hour for 6 blocks to be inserted in the chain, providing some assurance that the block with the relevant transaction will no longer be removed (recall that termination is not guaranteed). It is also noteworthy that the energy consumed by the Bitcoin network is estimated to be in the order of a few tens of TWh per year and raising due to the PoW mechanism, which corresponds to the energy spend by a medium-sized European country [72]. On the positive side, the Bitcoin network supports a large number of nodes —more than 10,000 currently [3]— whereas Byzantine consensus algorithms support only some 10 to 100 servers or so.

In the context of Bitcoin and generically blockchain, the term fork is also used with two other meanings. There is a *soft fork* when there is a new version of the node software that is backwards compatible with the previous version. A *hard fork* is similar except that the new version is not backwards compatible. When there is a hard fork, two subnetworks may start existing as some users may change to the new versions while others do not. In fact, although we talk simply about Bitcoin in this chapter, today there are several co-existing Bitcoin networks created by hard forks: Bitcoin, Bitcoin Cash, Bitcoin SV, Bitcoin Gold, etc.

### 1.3.2 Blockchain Applications

This section deviates from the line of the rest of the chapter on consensus algorithms. The objective is to present briefly what we called the two blockchain's killer apps: cryptocurrencies and smart contracts, as this is important to better understand the notion of blockchain.

The first *cryptocurrency*, in the sense we use the term, was Bitcoin [128], which we use as example. As already explained, Bitcoin is based on the original blockchain. Bitcoin's blockchain does not store the amount of money owned in users's *accounts*, but transactions involving accounts. The money in an account is designated the unspent transaction output (UTXO) [40].

An account is identified by a 20-byte address that is the hash of a public key $K_u$ of an elliptic curve public-key cryptographic (ECC) scheme. The corresponding private key $K_r$ is kept by the account's owner in a wallet. There are many wallet implementations, from those more secure that use specific hardware, to others less secure in software an even at the cloud.

A transaction contains one or more inputs. Each input contains a reference to a UTXO (a transaction in a block of the chain) and a signature produced by the owner of the account. That signature has two components: a real signature, obtained with the ECC scheme with $K_r$; and the public key $K_u$.

Every coin is created by a special transaction; every time a new block is mined and added to the chain, the first transaction of the block is such a transaction that creates coins.

Bitcoin's ledger can be considered a *state machine*: the state is the ownership of every coin (the UTXO of every account) and the transactions are commands that modify the state, either creating coins or transfering currency to other accounts (Section 1.2.2). On the contrary of most state machine implementations, in Bitcoin the state is not stored explicitly, but implicitly, i.e., in terms of an initial empty state and commands that modify it. The outcome of a transfer transaction can be the transfer of currency or an error (because the account does not have enough money or the signature is invalid), but recall that blocks with transactions that return error are not stored in the blockchain (Section 1.3.1). The major objective is to avoid *double spending*, i.e., that the same owner uses the same coins more than once.

The notion of *smart contracts* was introduced by Szabo in 1997 [150]. The general idea is the one of contracts that are automatically executed and, to the possible extent, automatically enforced. The author gives as example a vending machine, which can be considered to be a legacy, automatically executed and enforced, smart contract: a person inserts coins and automatically receives from the vendor a product. According to Szabo, "smart contracts go beyond the vending machine in proposing to embed contracts in all sorts of property that is valuable and controlled by digital means". However, there was no solution for implementing the concept at large scale in the late 1990s.

Buterin created Ethereum as a platform for the implementation of smart contracts in a blockchain, using a cryptocurrency when necessary [40]. Ethereum provides a cryptocurrency called *ether* that is similar to Bitcoin's. The way it works is very

similar to what we just described, except that the part of the state that is modified by the transactions in a block is stored in the chain, at the end of that block. However, Ethereum did not aim to be just another cryptocurrency, but to support the execution of distributed applications, designated by Dapps.

Besides a cryptocurrency, Ethereum provides support for the execution of smart contracts. In Ethereum an *account* is more complex than in Bitcoin. An account has four data items:

- *Nonce:* a counter used to guarantee that each transaction is processed only once;
- *Ether balance:* the currency in ether held by the contract;
- *Contract code:* the program that implements the contract;
- *Storage:* space to store data persistently.

Ethereum supports two types of accounts. *Externally owned accounts* have no contract code, are controlled by private cryptographic keys, and serve to do payments, just like Bitcoin's accounts.

*Contract accounts* are controlled by their contract code. When a contract account receives a *message*, its contract code is executed, possibly reading and writing the account's storage, sending messages, and creating new contracts. A message can come in a transaction from an externally owned account or from a contract. Notice that by *receiving* a message/transaction we actually mean the moment when a node accept there is consensus on a block and triggers that transaction for processing.

A message contains the following fields:

- *Nonce:* the nonce of the externally owned account that sent the message;
- *Recipient:* the recipient account for the message;
- *Ether:* the amount of currency (ether) to transfer from the sender's account to the recipient's account (if any);
- *Data:* an opaque data field with input(s) for the contract;
- *Startgas:* the maximum number of computational steps the transaction is allowed to execute;
- *Gasprice:* the *transaction fee* the sender account pays per computational step to the miner (the node that created the PoW for the block).

The *startgas* and *gasprice* have the purpose of preventing accidental or intentional denial of services. In practice, they force users to pay for the execution of contracts, limiting the amount of code they can execute. The term *gas* denominates the unit of computation. Ethereum executes smart contract code written in the Ethereum virtual machine (EVM) code language. For every EVM opcode there is a gas cost defined, varying from 0 (e.g., for the STOP opcode that halts the execution) to 32,000

(for the CREATE opcode that created a new account with code) [1]. There is an additional fee (5 gas) per byte in the transaction data field. The objective of this mechanisms is to make the sender —potentially an attacker— to pay the miner for the computational resources it uses.

Transactions containing a message from a contract to another contract have the same fields as a message (above), plus a *signature* created by the sender with its private key.

When a transaction for a contract account is received, Ethereum checks if: its syntax is correct, the signature is valid, and the nonce matches the sender's account nonce. If not, it returns an error, else it calculates the transaction fee (*startgas x gasprice*), subtracts the fee from the sender's account balance (if enough, otherwise returns an error), and increments the sender's nonce. Then, it transfers the transaction's ether from the sender's account to the recipient's and executes the contract code until it terminates or runs out of gas. If the value transfer fails because the sender does not have enough money or the code runs out of gas, all state changes are reverted, except the fees that are added to the miner's account. Otherwise, the fee corresponding to the remaining gas is transferred back to the sender's account.

EVM is a low-level programming language that is inconvenient for humans to write. Therefore, smart contracts are written in a higher-level language, typically Solidity [78], although there are other options (e.g., Serpent and LLL). The approach is, therefore, similar to Java and .NET: a high-level language (Java, C#, VB.NET, etc.) is compiled to a lower-level language (bytecodes, Common Intermediate Language) that is interpreted or goes through just-in-time compilation. EVM has the important characteristic of being Turing-complete. Bitcoin already supported basic programming in a language that was not Turing-complete (Script). Hyperledger Fabric uses an interesting alternative: smart contracts —*chaincode* in their lingo— are executed in Docker containers, so they can be programmed in several different languages [93].

### *1.3.3 Nakamoto Consensus Variants*

Creating variants of Bitcoin and Nakamoto consensus is one of the more prolific research topic in the blockchain area, so this section will focus on some of the most relevant works.

Many works improve the Nakamoto consensus with the goal of solving problems of *fairness* (Section 1.3.1). One of the first works in this line points out that nodes that receive transactions directly from non-miner nodes (lightweight wallets) benefit from keeping information about these transactions to themselves, so that they can put them in a block and receive the associated reward [22]. Babaioff et al. augment the Nakamoto consensus with a scheme to reward information propagation with fees, and prove that it can achieve an equilibrium using a game-theoretic model.

A *selfish mining attack* starts from the same idea of a node withholding transactions [80]. The attack consists in a (selfish) mining pool to withhold transactions, privately mine blocks with these transactions, and, when the length of the public chain gets shorter than the length of the private chain, the pool disseminates their blocks, intentionally forking the chain and making non-faulty nodes do useless com-

putation. This attack makes all nodes waste resources, but the selfish pool looses less and earns more reward than what corresponds to its mining power. Moreover, it is rational for the non-selfish nodes to join the selfish mining pool, increasing its share of the total mining power. Eyal and Sirer show that Bitcoin is not *incentive-compatible*, as it is vulnerable to selfish mining. They propose a modification to the Bitcoin algorithm to resist this attack: all (non-faulty) nodes shall propagate information about all competing blocks that lead to branches of the same length and pick randomly the one to which they will start mining the next block.

Sompolinsky and Zohar have shown that blockchains with many nodes and in which blocks are mined fast have high fork rates, allowing reasonably weak attackers to reverse payments they made [148]. To solve this problem, they present the *greedy heaviest-observed sub-tree* (GHOST) mechanism. Recall that in Bitcoin temporary forks are healed by pruning the smallest subchain(s). GHOST does not prune the smallest but the lightest subchains, i.e., it selects the heaviest subchain (or "subtree"). The weight of a subchain is the number of blocks it contains, counting all its branches. Ethereum implements a version of GHOST that considers only 7 generations of blocks, to simplify implementation and to keep the incentive of mining blocks for the longest chain [40].

Miller et al. recognize the problems of fairness created by large mining pools, referring that in 2014 the largest mining pool, GHash.IO, exceeded 50% of the mining power [116]. To deal with this problem, they introduce the notion of *strongly nonoutsourceable puzzles* (SNP), which are alternatives to PoW that are not amenable for mining pools. Specifically, a SNP is a cryptopuzzle that satisfies the following property: if a pool operator outsources mining work to a worker, then the worker can steal the reward without producing any evidence that can potentially implicate itself [116]. The authors present a SNP scheme called *scratch-off puzzles*. The scheme involves creating puzzles that are solvable in parallel, in such a way that the workers retain a signing key that can later be used to obtain their reward. Moreover, it provides a zero-knowledge spending option that allows workers to spend their reward in a way that does not reveal information.

The PoW mechanism provides the benefit of allowing arbitrary nodes to participate in a blockchain at the cost of consuming much energy (tens of TWh per year in Bitcoin, as mentioned before). Therefore, there is a quest for alternative mechanisms that solve the same problem without such a cost. The *proof-of-stake* (PoS) mechanism is one of the first. The idea is that nodes have a certain *stake* in the network, instead of having done a certain work. Nodes are called *stakers* or *minters*, no longer miners. Stakers do not get a reward for mining a block, only transaction fees. PPCoin uses a hybrid scheme in which there are two types of blocks: PoW blocks, the same as in Bitcoin; and PoS blocks that contain a single transaction called coinstake [99]. Its authors introduce the notion of *coin age* that is the number of coins the user holds multiplied by the time he holds them. In a coinstake transaction an user pays himself an amount, effectively reducing his coin age. Producing a PoS block involves solving a cryptopuzzle similar to obtaining a PoW, but requiring low effort and low energy consumption (the search space is limited). Their scheme uses PoW blocks in the begining but substitutes them by PoS blocks at some stage (undefined). The difficulty of

the cryptopuzzle is variable and is lower the higher the coin age consumed. Casper the Friendly Finality Gadget (Casper FFG) is a PoS mechanism being develop for Ethereum [41]. Casper FFG does not aim to create chains of blocks —in Ethereum these will continue to be created using PoW for now— but to select which subchain to maintain and which to prune, in case there are conflicting subchains. The main idea is to use Casper FFG to produces periodic checkpoints (1 every 100 blocks) using PoS. Moreover, it provides an accountability scheme that allows punishing faulty nodes. In the context of Ethereum a complementary mechanism called Casper the Friendly GHOST (Casper TFG) is being designed, but less information about it is available. Activating Casper in Ethereum will involve a hard fork of that blockchain.

There is a line or research on *useful puzzles*, i.e., on alternatives to PoW that spend energy solving useful tasks. The challenge is to find puzzles that are hard to compute, but with solutions that that can be verified efficiently. Primecoin seems to be the first cryptocurrency based on such an puzzle [98]. Its puzzles involve finding chains of prime numbers that are large enough to be hard to find, but not to verify. These prime numbers might be useful as cryptographic keys. Permacoin substitutes PoW by *proofs-of-retrievability* (PoR) to support distributed storage of data [115]. The purpose of a PoR is to certify that a node is using storage space to store a file. Therefore, Permacoin is a peer-to-peer file storage system in which nodes have an incentive to provide storage space, instead of mere altruism.

Apostolaki et al. show that by hijacking less than 100 BGP prefixes it is possible to isolate more than 50% of the Bitcoin mining power, even when considering that mining pools are strongly multi-homed [19]. They present several countermeasures to reduce the impact of these attacks on Bitcoin, most of them related to increasing the number or diversity of connections between nodes (8 by default in Bitcoin) and to protect the communication (by default not encrypted or integrity-protected).

Table 1.5 summarizes the works on variants of Nakamoto consensus presented.

| **Algorithm** | **Mechanism** | **Problem Handled** | **References** |
|---|---|---|---|
| Bitcoin | proof-of-work (PoW) | double spending | [128] |
| Ethereum | PoW / PoS | efficiency, using PoS | [40] |
| Bitcoin variant | PoW | fairness, by rewarding | [22] |
| Bitcoin variant | PoW | selfish mining | [80] |
| Bitcoin variant | greedy heaviest-observed sub-tree (GHOST) | high fork rate | [148] |
| Bitcoin variant | strongly nonoutsourceable puzzles (SNP) | large mining pools | [116] |
| PPCoin | proof-of-stake (PoS) | PoW inefficiency | [99] |
| Ethereum | Casper FFG / TFG | PoW inefficiency | [41] |
| Primecoin | useful puzzles | useless spending of energy | [98] |
| Permacoin | proofs-of-retrievability (PoR) | distributed storage of data | [115] |
| Bitcoin variants | several countermeasures | mining power isolation | [19] |

**Table 1.5: Nakamoto consensus variants.**

Summary

This section provided a brief state-of-the-art on blockchain with Nakamoto consensus and its variants. We started with a presentation of the Bitcoin blockchain and its consensus algorithm. Then, we presented two major applications for blockchains: cryptocurrencies and smart contracts. Finally, we presented several variants of the Nakamoto consensus and PoW, such as proof-of-stake and other related schemes. The next section presents blockchains based on Byzantine consensus algorithms (Section 1.2), some of them also with PoW and other mechanisms related to Nakamoto consensus.

## 1.4 Blockchains with Byzantine Consensus

The previous section presented the original blockchain (Bitcoin's), Ethereum, and several variants of Nakamoto consensus. These consensus algorithms have essentially the same goal of the Byzantine consensus algorithms of Section 1.2 —doing agreement— but trade scalability for performance and weaker versions of the agreement and termination properties (Section 1.3.1). However, one of the evolutions of these first blockchains are the so called *permissioned blockchains*, which require a different tradeoff. This section starts by presenting such permissioned blockchains, then presents a new generation of *permissionless* blockchains that provide a service similar to Bitcoin and Ethereum, but better performance. The former are based on variants of Byzantine consensus algorithms such as BFT-SMaRt, whereas the later are based on *hybrid* consensus algorithms.

### *1.4.1 Permissioned Blockchains with Byzantine Consensus*

Around 2014 many private companies started to understand the potential benefits of blockchain technology. That year, a group of major financial institutions formed a consortium called R3 with the objective of understanding the benefits of blockchain for their operation, e.g., to support payments between them [134]. These companies and institutions soon understood that the *permissionless* nature of Bitcoin or Ethereum —the lack of need of permission to participate— was not what they needed. In fact, it was even unacceptable to put customer information in such infrastructures, from the point of view of compliance, privacy, etc.

This scenario has lead to the appearance of *permissioned blockchains*, in which nodes must authenticate themselves and be authorized to become peers. This allowed the creation of *consortia blockchains* (managed by a group of institutions) and *private blockchains* (managed by a single institution) [39]. Permissioned blockchains have important differences in relation to permissionless: they do not need the PoW mechanism to throttle appends to the blockchain; the number of nodes is typically much lower (e.g., 5 to 20 instead of hundreds or thousands); they can be implemented using Byzantine consensus algorithms (Section 1.2) and made much more efficient. In relation to consensus definitions, they can use validity predicate-based consen-

sus, no longer *speculative* validity predicate-based consensus as Bitcoin and other permissionless blockchains.

Next, we review some of the existing permissioned blockchains based on Byzantine consensus:

Hyperledger is a blockchain project of the Linux Foundation that is developing several blockchains. The most popular of these blockchains seems to be Hyperledger Fabric, a modular blockchain focused on running smart contracts [42, 17]. Hyperledger Fabric does not provide a cryptocurrency and runs smart contracts (chaincode) in Docker containers (Section 1.3.2). Fabric considers two types of nodes: *validating peers* that run the consensus algorithm, validate the transactions, and maintain the chain; and *non-validating peers* that function as proxies to connect clients that issue transactions, possibly validating but not executing these transactions. Despite these nodes being called peers, Fabric is not peer-to-peer, as all validating peers communicate with all the others directly.

The consensus algorithm used is configurable; when version 1.0 was published, there was a fault-tolerant consensus algorithm based on the Apache Kafta stream processing platform; recently a Byzantine consensus algorithm based on the BFT-SMaRt library [30, 149] has been included. This later algorithm is a variant of PBFT so it solves SMR, is leader-based, has optimal resilience ($n \geq 3f+1$), and assumes the basic system model extended with a weak time assumption to circumvent FLP. Unlike the blockchains presented in the previous section (Bitcoin's, Ethereum), it provides strong agreement and termination properties, not their eventual versions. BFT-SMaRt's default algorithm is quite efficient, with a throughput of tens of thousands of transactions per second and latency below 1s [149]. BFT-SMaRt provides another consensus algorithm called WHEAT that uses speculative executions and other mechanisms to be more efficient in large-scale deployments, at the cost of becoming slower if not all replicas reply within certain time bounds.

Corda was developed by the R3 consortium [36]. Corda is similar to Hyperledger Fabric in many aspects. Its purpose is also to support the execution of smart contracts, although the motivation of the consortium is specifically financial agreements. In Corda, the consensus algorithm is executed by nodes called *notaries* and is also not peer-to-peer. This algorithm is again configurable and the Byzantine fault-tolerant version is also based on the BFT-SMaRt library.

Tendermint provides its own consensus algorithm, but it is similar to PBFT and BFT-SMaRt's default algorithm [37]. There is, however, a major difference in relation to Fabric and Corda: Tendermint's communication is peer-to-peer, as each node sends messages (gossip) only to a subset of the existing nodes. This in theory might allow a better throughput than using BFT-SMaRt, but the experimental results suggest their performance is similar [37], although it is not fair to compare experiments done in different settings.

The Red Belly Blockchain is a permissioned blockchain based on an efficient Byzantine consensus algorithm [66, 64]. The algorithm has two components. The first is a Byzantine binary consensus that, as the previous algorithms, assumes the basic system model extended with a weak time assumption. Moreover, it is also leader-based, although it rotates the leader so there is no leader election involved

(similarly to our Spinning algorithm [156]). The second is a transformation from binary consensus into multi-valued consensus. The overall algorithm has optimal resilience and terminates in O(f) steps (or O(1) if there are no faulty nodes). Red Belly Blockchain supports some hundreds of nodes and a throughput of several hundred thousand transactions per second [66].

These works do not consider a hybrid model (Section 1.2.5), so nodes do not require trusted components (wormholes) and their resilience is $n \geq 3f+1$. In large blockchains like Bitcoin's there is no need to improve this resilience, but in consortium or private blockchains there may be a benefit on having only $n = 2f+1$ nodes, e.g., only 5 or 7 instead of 7 or 10, which we have shown to be possible with trusted components [57, 158]. Moreover, with a trusted component it is also possible to reduce the number of communication steps and messages exchanged, which is beneficial in terms of latency and throughput [158]. Interestingly, the Hyperledger project is implementing a version of our MinBFT algorithm based on Intel SGX [4].

For completeness, we mention that there are many other permissioned blockchains available, although they use different approaches to do consensus: Quorum, an enterprise version of Ethereum; Hyperledger Burrow, from the Hyperledger project; Guardtime's KSI blockchain, developed for the Estonian government; Ripple, that provides a money transfer service; Stellar, that supports the deployment of financial products; and others.

### *1.4.2 Permissionless Blockchains with Hybrid Consensus*

The blockchains presented in the previous section are unsuitable for permissionless operation for at least two reasons. First, they become less efficient when the number of nodes grow; in fact they certainly cannot support a number of nodes similar to Bitcoin (more than 10,000 [3]) or Ethereum (also more than 10,000 [2]). Second, they reach consensus based on node votes, so they need the nodes that can vote to be clearly identified, otherwise they would be vulnerable to Sybil attacks (a node that pretends to be several) [74] or to collusions of faulty nodes.

There are some recent proposals of permissionless blockchains that leverage the benefits of Byzantine consensus algorithms, whereas also using PoW and related mechanisms to allow any node to be part of the system. These blockchains are based on what some authors call *hybrid* consensus algorithms, in the sense that they mix the two types of algorithms [24]. Next, we present briefly some of these systems.

Bitcoin-NG aims to provide a more scalable version of Bitcoin [79]. The idea is to use a leader-based Byzantine consensus algorithm to order blocks of transactions; to deal with the fact that any node can be part of the blockchain, Bitcoin-NG uses PoW to elect the leader. However, this use of PoW allows the existence of temporary forks, as in Bitcoin's blockchain.

PeerCensus [69] and Hybrid consensus [132] are based on a similar idea: they use PoW to limit the rate at which nodes can join the blockchain, and run a Byzantine consensus algorithm among the blockchain nodes to order blocks. Again, the possibility of temporary forks remains. PeerCensus is used to build a cryptocurrency called Discoin. More importantly, the Hybrid Consensus work studies the conditions

in which a permissionless consensus can satisfy the property of *responsiveness*, i.e., in which the latency for a transaction to be confirmed is a function of the network delay. They prove that permissionless consensus is impossible in partially synchronous models, and that their hybrid consensus is responsive (by leveraging PoW).

Byzcoin is based on the same idea of using a variation of Byzantine consensus for ordering and PoW for defining the committee of nodes that run the consensus [100]. However, its authors introduce two interesting ideas for performance reasons. First, they substitute PBFT's vectors of MACs by signatures and use a *collective signing* protocol (nodes jointly produce signatures) to avoid the leader to check O(n) signatures during consensus. Second, they use the notion of *communication trees* from multicast protocols to reduce the number of messages sent.

Algorand is a cryptocurrency that introduces several new mechanisms and is not based on PoW [87]. First, it considers *weighted users*, i.e., the voting power of a node depends on the amount of money it holds. In that sense, there is some relation to PoS. Second, it does *consensus by committee*, i.e., only a subset of the nodes (committee) run the Byzantine consensus algorithm, greatly reducing the number of messages exchanged. The committee members are chosen randomly, but the probability of a node being chosen is not constant, depends on its weight. Third, it uses a *cryptographic sortition scheme* for committee members to be chosen in a private and non-interactive way (i.e., without communication), making it harder for adversaries to do denial of service attacks against these nodes. Fourth, it uses *participant replacement* again to mitigate denial of service attacks. The idea is that once a committee member sends a message, it stops being a member, so it is no longer useful to do denial of service against it.

Solida is inspired by Byzcoin and Hybrid consensus [6]. Transactions are ordered by a committee using an adaptation of the PBFT algorithm. The number of members of the committee is fixed, so when a new member joins, the oldest leaves. A node joins the committee by presenting a PoW and, when it does so, it becomes the leader. If two nodes obtain a PoW concurrently, a leader election protocol is executed to select which one becomes the leader, avoiding the creation of temporary forks.

Other more recent algorithms that follow similar principles are Chainspace and Omniledger [8, 101].

Table 1.6 summarizes both the permissioned and permissionless blockchains presented.

## Summary

This section presents a set of recent blockchains based on Byzantine consensus. First, it presents permissioned blockchains, which run essentially a Byzantine consensus algorithm, typically a SMR algorithm. Then, it presents permissionless blockchains that require a more complex consensus algorithm to deal with the fact that participants are unknown and Sybil attacks may exist.

| Algorithm | Type | Characteristics | References |
|---|---|---|---|
| Fabric w/BFT-SMaRt | Permissioned | modular, efficient, not peer-to-peer | [17, 149] |
| Corda w/BFT-SMaRt | Permissioned | similar to the previous | [36] |
| Tendermint | Permissioned | peer-to-peer | [37] |
| Red Belly Blockchain | Permissioned | based on an efficient Byzantine consensus algorithm | [66, 64] |
| Bitcoin-NG | Permissionless / Hybrid | use PoW to elect the leader of a leader-based consensus | [79] |
| PeerCensus, Hybrid consensus | Permissionless / Hybrid | use PoW to throttle nodes joining the committee that runs consensus | [69, 132] |
| Byzcoin | Permissionless / Hybrid | similar but uses multicast trees | [100] |
| Algorand | Permissionless / Hybrid | committee members selected using a cryptographic sortition scheme | [87] |
| Solida | Permissionless / Hybrid | when node joins the committee it becomes leader and another leaves | [6] |
| Chainspace, Omniledger | Permissionless / Hybrid | similar | [8, 101] |

**Table 1.6: Blockchains with Byzantine consensus algorithms.**

## 1.5 Conclusion

The objective of the chapter was to present a state of the art in the area of Byzantine consensus and its application in blockchains.

Section 1.1 introduces the notions explored in the chapter, including those related to blockchain and Byzantine consensus.

Section 1.2 presents a state-of-the-art in what we call Byzantine consensus, which excludes the Nakamoto consensus based on PoW. The area started in the late 1970s with synchronous algorithms, but the section is about more recent work, starting with the FLP impossibility result, then with practical algorithms starting with PBFT.

Section 1.3 presents the original blockchain, its consensus algorithm, the killer applications (cryptocurrencies and smart contracts), and variants of PoW-based consensus.

Section 1.4 starts by presenting permissioned blockchains, then presents a new generation of permissionless blockchains that provide a service similar to Bitcoin and Ethereum, but better performance. The former are based on variants of Byzantine consensus algorithms such as BFT-SMaRt, whereas the later are based on hybrid consensus algorithms.

*Acknowledgements.* This work was partially supported by the EC through project 822404 (QualiChain), and by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2019 (INESC-ID).

# References


[1] Ethereum VM (EVM) opcodes and instruction reference. https://github.com/trailofbits/evm-opcodes, 2018.

[2] ethernodes.org: Network number 1. https://www.ethernodes.org/network/1, January 2018.

[3] Global Bitcoin nodes distribution. https://bitnodes.earn.com/, January 2018.

[4] Implementation of MinBFT consensus protocol. https://github.com/hyperledger-labs/minbft, 2018.

[5] Ittai Abraham, Marcos Kawazoe Aguilera, and Dahlia Malkhi. Fast asynchronous consensus with optimal resilience. In *Proceedings of the 24th International Symposium on Distributed Computing*, pages 4–19, 2010.

[6] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Alexander Spiegelman. Solida: A blockchain protocol based on reconfigurable Byzantine consensus. In *Proceedings of the 21st International Conference on Principles of Distributed Systems*, 2017.

[7] Marcos K. Aguilera. A pleasant stroll through the land of infinitely many creatures. *ACM SIGACT News*, 35(2):36–59, 2004.

[8] Mustafa Al-Bassam, Alberto Sonnino, Shehar Bano, Dave Hrycyszyn, and George Danezis. Chainspace: A sharded smart contracts platform. In *Proceedings of the Network and Distributed System Security Symposium*, 2018.

[9] Eduardo Adilio Pelinson Alchieri, Alysson Bessani, Fabiola Greve, and Joni da Silva Fraga. Knowledge connectivity requirements for solving Byzantine consensus with unknown participants. *IEEE Transactions on Dependable and Secure Computing*, 15(2):246–259, 2018.

[10] Eduardo Adílio Pelinson Alchieri, Alysson Neves Bessani, Joni da Silva Fraga, and Fabíola Greve. Byzantine consensus with unknown participants. In *Proceedings of the 12th International Conference on Principles of Distributed Systems*, pages 22–40, 2008.

[11] Noga Alon, Michael Merrit, Omer Reingold, Gadi Taubenfeld, and Rebeca Wright. Tight bounds for shared memory systems acessed by Byzantine processes. *Distributed Computing*, 18(2):99–109, November 2005.

[12] Bowen Alpern and Fred Schneider. Defining liveness. Technical report, Department of Computer Science, Cornell University, 1984.

[13] Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.

[14] Yair Amir, Brian Coan, Jonathan Kirsch, and John Lane. Byzantine replication under attack. In *Proceedings of the IEEE/IFIP 38th International Conference on Dependable Systems and Networks*, pages 197–206, June 2008.

[15] Yair Amir, Brian A. Coan, Jonathan Kirsch, and John Lane. Prime: Byzantine replication under attack. *IEEE Transactions on Dependable and Secure Computing*, 8(4):564–577, 2011.

[16] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[17] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the 13th ACM EuroSys Conference*, 2018.

[18] Andreas Antonopoulos. *Mastering Bitcoin*. O'Reilly, 2nd edition, 2017.

[19] Maria Apostolaki, Aviv Zohar, and Laurent Vanbever. Hijacking Bitcoin: Routing attacks on cryptocurrencies. In *Proceedings of the 2017 IEEE Symposium on Security and Privacy*, pages 375–392, 2017.

[20] Paul C. Attie. Wait-free Byzantine consensus. *Information Processing Letters*, 83(4):221–227, August 2002.

[21] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, Jan-Mar 2004.

[22] Moshe Babaioff, Shahar Dobzinski, Sigal Oren, and Aviv Zohar. On Bitcoin and red balloons. In *Proceedings of the 13th ACM Conference on Electronic Commerce*, pages 56–73, 2012.

[23] Roberto Baldoni, Jean-Michel Helary, Michel Raynal, and Lenaik Tanguy. Consensus in Byzantine asynchronous systems. *Journal of Discrete Algorithms*, 1(2):185–210, 2003.

[24] Shehar Bano, Alberto Sonnino, Mustafa Al-Bassam, Sarah Azouvi, Patrick McCorry, Sarah Meiklejohn, and George Danezis. SoK: Consensus in the age of blockchains. *arXiv preprint arXiv:1711.03936*, 2017.

[25] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. Consensus-oriented parallelization: How to earn your first million. In *Proceedings of the 16th Annual Middleware Conference*, pages 173–184, 2015.

[26] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. Hybrids on steroids: SGX-based high performance BFT. In *Proceedings of the 12th ACM European Conference on Computer Systems*, pages 222–237, 2017.

[27] Michael Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols. In *Proceedings of the 2nd ACM Symposium on Principles of Distributed Computing*, pages 27–30, August 1983.

[28] Michael Ben-Or. Fast asynchronous Byzantine agreement. In *Proceedings of the 4th ACM Symposium on Principles of Distributed Computing*, pages 149–151, August 1985.

[29] Iddo Bentov, Charles Lee, Alex Mizrahi, and Meni Rosenfeld. Proof of activity: Extending Bitcoin's proof of work via proof of stake. *ACM SIGMETRICS Performance Evaluation Review*, 42(3), 2014.

[30] Alysson Bessani, João Sousa, and Eduardo E. P. Alchieri. State machine replication for the masses with BFT-Smart. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 355–362, June 2014.

[31] Alysson Neves Bessani and Eduardo Alchieri. A guided tour on the theory and practice of state machine replication. In *Tutorials of the 32nd Brazilian Symposium on Computer Networks and Distributed Systems*, 2014.

[32] Alysson Neves Bessani, Miguel Correia, Joni da Silva Fraga, and Lau Cheuk Lung. Sharing memory between byzantine processes using policy-enforced tuple spaces. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):419–432, 2009.

[33] Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Narayanan, Joshua A Kroll, and Edward W Felten. SoK: Research perspectives and challenges for Bitcoin and cryptocurrencies. In *2015 IEEE Symposium on Security and Privacy*, pages 104–121, 2015.

[34] Gabriel Bracha. An asynchronous $\lfloor (n-1)/3 \rfloor$-resilient consensus protocol. In *Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing*, pages 154–162, August 1984.

[35] Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM*, 32(4):824–840, October 1985.

[36] Richard Gendal Brown, James Carlyle, Ian Grigg, and Mike Hearn. Corda: An introduction. *R3 CEV, August*, 2016.

[37] Ethan Buchman. Tendermint: Byzantine fault tolerance in the age of blockchains. Master's thesis, The University of Guelph, 2016.

[38] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the USENIX 7th Symposium on Operating Systems Design and Implementation*, pages 335–350, 2006.

[39] Vitalik Buterin. On public and private blockchains. https://blog.ethereum.org/2015/08/07/on-public-and-private-blockchains/, August 2015.

[40] Vitalik Buterin and Ethereum team. Ethereum - a next-generation smart contract and decentralized application platform. White Paper, 2014-17.

[41] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *arXiv preprint arXiv:1710.09437*, 2017.

[42] Christian Cachin. Architecture of the Hyperledger blockchain fabric. In *Workshop on Distributed Cryptocurrencies and Consensus Ledgers*, 2016.

[43] Christian Cachin and Jonathan A. Poritz. Secure intrusion-tolerant replication on the Internet. In *Proceedings of the Conference on Dependable Systems and Networks*, pages 167–176, 2002.

[44] Cristian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *Advances in Cryptology: CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[45] Cristian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in Contanstinople: Practical asynchronous Byzantine agreement using cryptography. In *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing*, pages 123–132, July 2000.

[46] Ran Canetti and Tal Rabin. Fast asynchronous Byzantine agreement with optimal resilience. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, pages 42–51, 1993.

[47] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, pages 173–186, February 1999.

[48] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996.

[49] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[50] Soma Chaudhuri. More choices allow more faults: Set consensus problems in totally asynchronous systems. *Information and Computation*, 105(1):132–158, July 1993.

[51] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. Attested append-only memory: making adversaries stick to their word. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, pages 189–204, October 2007.

[52] Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design & Implementation*, pages 153–168, 22–24 April 2009.

[53] Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. UpRight Cluster Services. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, October 2009.

[54] coinmarketcap. https://coinmarketcap.com/, March 2018.

[55] Miguel Correia, Alysson Neves Bessani, and Paulo Veríssimo. On Byzantine generals with alternative plans. *Journal of Parallel and Distributed Computing*, 68(9):1291–1296, September 2008.

[56] Miguel Correia, Nuno F. Neves, Lau Cheuk Lung, and Paulo Veríssimo. Low complexity Byzantine-resilient consensus. *Distributed Computing*, 17(3):237–249, 2005.

[57] Miguel Correia, Nuno F. Neves, and Paulo Veríssimo. How to tolerate half less one Byzantine nodes in practical distributed systems. In *Proceedings of the 23rd IEEE Symposium on Reliable Distributed Systems*, pages 174–183, October 2004.

[58] Miguel Correia, Nuno F. Neves, and Paulo Veríssimo. From consensus to atomic broadcast: Time-free Byzantine-resistant protocols without signatures. *Computer Journal*, 41(1):82–96, January 2006.

[59] Miguel Correia, Nuno Ferreira Neves, and Paulo Veríssimo. BFT-TO: intrusion tolerance with less replicas. *Computer Journal*, 56(6):693–715, 2013.

[60] Miguel Correia, Paulo Veríssimo, and Nuno F. Neves. The design of a COTS real-time distributed security kernel. In *Proceedings of the 4th European Dependable Computing Conference*, pages 234–252, October 2002.

[61] Miguel Correia, Giuliana Santos Veronese, and Lau Cheuk Lung. Asynchronous byzantine consensus with 2f+1 processes. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC)*, pages 475–480, 2010.

[62] Miguel Correia, Giuliana Santos Veronese, Nuno Ferreira Neves, and Paulo Veríssimo. Byzantine consensus in asynchronous message-passing systems: a survey. *International Journal of Critical Computer-Based Systems*, 2(2):141–161, 2011.

[63] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. HQ-Replication: A hybrid quorum protocol for Byzantine fault tolerance. In *Proceedings of 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 177–190, November 2006.

[64] Tyler Crain, Vincent Gramoli, Mikel Larrea, and Michel Raynal. Blockchain consensus. In *ALGOTEL 2017-19èmes Rencontres Francophones sur les Aspects Algorithmiques des Télécommunications*, 2017.

[65] Tyler Crain, Vincent Gramoli, Mikel Larrea, and Michel Raynal. (leader/randomization/signature)-free Byzantine consensus for consortium blockchains. *arXiv preprint arXiv:1702.03068*, February 2017.

[66] Tyler Crain, Vincent Gramoli, Chris Natoli, Gary Shapiro, Michael Spain, and Guillaume Vizier. Red Belly Blockchain. http://csrg.redbellyblockchain.io/rbbc/.

[67] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gün Sirer, Dawn Song, and Roger Wattenhofer. On scaling decentralized blockchains. In *International Conference on Financial Cryptography and Data Security*, pages 106–125, 2016.

[68] Roberto De Prisco, Dahlia Malkhi, and Michael Reiter. On k-set consensus problems in asynchronous systems. In *Proceedings of the 18th ACM Symposium on Principles of Distributed Computing*, pages 257–265, May 1999.

[69] Christian Decker, Jochen Seidel, and Roger Wattenhofer. Bitcoin meets strong consistency. In *Proceedings of the 17th International Conference on Distributed Computing and Networking*, 2016.

[70] Christian Decker and Roger Wattenhofer. Information propagation in the Bitcoin network. In *Proceedings of the IEEE 13th International Conference on Peer-to-Peer Computing*, pages 1–10, 2013.

[71] Tim Dierks and Christopher Allen. The TLS Protocol Version 1.0 (RFC 2246). IETF Request For Comments, January 1999.

[72] Digiconomist. Bitcoin energy consumption index. https://digiconomist.net/bitcoin-energy-consumption, December 2017.

[73] Danny Dolev. The Byzantine generals strike again. *Journal of Algorithms*, (3):14–30, 1982.

[74] John R Douceur. The Sybil attack. In *International Workshop on Peer-to-Peer Systems*, pages 251–260, 2002.

[75] Assia Doudou, Bernoit Garbinato, and Rachid Guerraoui. Encapsulating failure detection: From crash-stop to Byzantine failures. In *International Conference on Reliable Software Technologies*, pages 24–50, May 2002.

[76] Assia Doudou, Bernoit Garbinato, Rachid Guerraoui, and Andre Schiper. Muteness failure detectors: Specification and implementation. In *Proceedings of the Third European Dependable Computing Conference*, pages 71–87, September 1999.

[77] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

[78] Ethereum team. Solidity. https://solidity.readthedocs.io/en/develop/, 2016–2017.

[79] Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert van Renesse. Bitcoin-ng: A scalable blockchain protocol. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation*, 2016.

[80] Ittay Eyal and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *International Conference on Financial Cryptography and Data Security*, pages 436–454, 2014.

[81] Paul Ezhilchelvan, Achour Mostefaoui, and Michel Raynal. Randomized multivalued consensus. In *Proceedings of the 4th IEEE International Symposium on Object-Oriented Real-Time Computing*, pages 195–200, May 2001.

[82] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[83] Roy Friedman, Achour Mostefaoui, Sergio Rajsbaum, and Michel Raynal. Distributed agreement and its relation with error-correcting codes. In *Proceedings of the 16th International Conference on Distributed Computing*, pages 63–87, October 2002.

[84] Roy Friedman, Achour Mostefaoui, and Michel Raynal. Simple and efficient oracle-based consensus protocols for asynchronous Byzantine systems. *IEEE Transactions on Dependable and Secure Computing*, 2(1):46–56, January 2005.

[85] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The Bitcoin backbone protocol: Analysis and applications. In *EUROCRYPT (2)*, pages 281–310, 2015.

[86] Arthur Gervais, Ghassan O Karame, Karl Wüst, Vasileios Glykantzis, Hubert Ritzdorf, and Srdjan Capkun. On the security and performance of proof of work blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 3–16, 2016.

[87] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling Byzantine agreements for cryptocurrencies. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles*, pages 51–68, 2017.

[88] Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 BFT protocols. In *Proceedings of the 5th ACM SIGOPS/EuroSys European Systems Conference*, pages 363–376, 2010.

[89] Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425, Cornell University, Department of Computer Science, May 1994.

[90] Garrick Hileman and Michel Rauchs. Global blockchain benchmarking study. Cambridge Centre for Alternative Finance, 2017.

[91] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference*, 2010.

[92] Michel Hurfin and Michel Raynal. A simple and fast asynchronous consensus protocol based on a weak failure detector. *Distributed Computing*, 12:209–223, 1999.

[93] Hyperledger Fabric team. Hyperledger Fabric documentation. https://hyperledger-fabric.readthedocs.io/.

[94] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. Cheapbft: resource-efficient byzantine fault tolerance. In *Proceedings of the 7th ACM European Conference on Computer Systems*, pages 295–308, 2012.

[95] Rami Khalil and Arthur Gervais. Revive: Rebalancing off-blockchain payment networks. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 439–453, 2017.

[96] Kim Potter Kihlstrom, Louise E. Moser, and Paul M. Melliar-Smith. The SecureRing group communication system. *ACM Transactions on Information and System Security*, 4(4):371–406, November 2001.

[97] Kim Potter Kihlstrom, Louise E. Moser, and Paul M. Melliar-Smith. Byzantine fault detectors for solving consensus. *The Computer Journal*, 46(1):16–35, January 2003.

[98] Sunny King. Primecoin: Cryptocurrency with prime number proof-of-work. 2013.

[99] Sunny King and Scott Nadal. PPCoin: Peer-to-peer crypto-currency with proof-of-stake. self-published paper, 2012.

[100] Eleftherios Kokoris Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. Enhancing Bitcoin security and performance with strong consistency via collective signing. In *25th USENIX Security Symposium*, pages 279–296, 2016.

[101] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. OmniLedger: A secure, scale-out, decentralized ledger via sharding. In *Proceedings of the 39th IEEE Symposium on Security and Privacy*, 2018.

[102] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine fault tolerance. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, October 2007.

[103] Leslie Lamport. The part-time parliament. *ACM Transactions Computer Systems*, 16(2):133–169, May 1998.

[104] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):51–58, 2001.

[105] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[106] Butler Lampson. The abcd's of paxos. In *Proc. of the 20th annual ACM Symp. on Principles of Distributed Computing*, 2001.

[107] Dave Levin, John R. Douceur, Jacob R. Lorch, and Thomas Moscibroda. Trinc: small trusted hardware for large distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, pages 1–14, 2009.

[108] Bev Littlewood. The impact of diversity upon common mode failures. *Reliability Engineering & System Safety*, 51(1):101–113, 1996.

[109] Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, and Marko Vukolic. XFT: Practical fault tolerance beyond crashes. In *12th USENIX Symposium on Operating Systems Design and Implementation*, pages 485–500, 2016.

[110] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 254–269, 2016.

[111] Dahlia Malkhi, Michael Merrit, Micheal Reiter, and Gadi Taubenfeld. Objects shared by Byzantine processes. *Distributed Computing*, 16(1):37–48, February 2003.

[112] Dahlia Malkhi and Michael Reiter. Unreliable intrusion detection in distributed computations. In *Proceedings of the 10th Computer Security Foundations Workshop*, pages 116–124, June 1997.

[113] Jean-Philippe Martin and Lorenzo Alvisi. Fast Byzantine consensus. In *Proceedings of the IEEE/IFIP 35th International Conference on Dependable Systems and Networks*, pages 402–411, June 2005.

[114] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[115] Andrew Miller, Ari Juels, Elaine Shi, Bryan Parno, and Jonathan Katz. Permacoin: Repurposing Bitcoin work for data preservation. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 475–490, 2014.

[116] Andrew Miller, Ahmed Kosba, Jonathan Katz, and Elaine Shi. Nonoutsourceable scratch-off puzzles to discourage Bitcoin mining coalitions. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 680–691, 2015.

[117] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of BFT protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 31–42, 2016.

[118] Henrique Moniz, Nuno Ferreira Neves, and Miguel Correia. Turquois: Byzantine consensus in wireless ad hoc networks. In *Proceedings of the 2010 IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 537–546, 2010.

[119] Henrique Moniz, Nuno Ferreira Neves, and Miguel Correia. Byzantine fault-tolerant consensus in wireless ad hoc networks. *IEEE Transactions on Mobile Computing*, 12(12):2441–2454, 2013.

[120] Henrique Moniz, Nuno Ferreira Neves, Miguel Correia, Antonio Casimiro, and Paulo Verissimo. Intrusion tolerance in wireless environments: An experimental evaluation. In *Proceedings of the 13th IEEE Pacific Rim Dependable Computing Conference*, December 2007.

[121] Henrique Moniz, Nuno Ferreira Neves, Miguel Correia, and Paulo Verissimo. Experimental comparison of local and shared coin randomized consensus protocols. In *Proceedings of the 25th IEEE Symposium on Reliable Distributed Systems*, pages 235–244, October 2006.

[122] Henrique Moniz, Nuno Ferreira Neves, Miguel Correia, and Paulo Veríssimo. Randomized intrusion-tolerant asynchronous services. In *Proceedings of the IEEE/IFIP 36th International Conference on Dependable Systems and Networks*, pages 568–577, 25-28 June 2006.

[123] Henrique Moniz, Nuno Ferreira Neves, Miguel Correia, and Paulo Veríssimo. Randomization can be a healer: Consensus with dynamic omission failures. In *Distributed Computing, 23rd International Symposium*, pages 63–77, 2009.

[124] Henrique Moniz, Nuno Ferreira Neves, Miguel Correia, and Paulo Veríssimo. Randomization can be a healer: consensus with dynamic omission failures. *Distributed Computing*, 24(3-4):165–175, 2011.

[125] Henrique Moniz, Nuno Ferreira Neves, Miguel Correia, and Paulo Veríssimo. RITAS: services for randomized intrusion tolerance. *IEEE Transactions on Dependable and Secure Computing*, 8(1):122–136, 2011.

[126] Achour Mostefaoui, Michel Raynal, Corentin Travers, Stacy Patterson, Divyakant Agrawal, and Amr El Abbadi. From static distributed systems to dynamic systems. In *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems*, pages 109–118, October 2005.

[127] D. Mpoeleng, P. Ezhilchelvan, and N. Speirs. From crash tolerance to authenticated Byzantine tolerance: A structured approach, the cost and benefits. In *Proceedings of the IEEE/IFIP 33rd International Conference on Dependable Systems and Networks*, pages 227–236, June 2003.

[128] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[129] Ray Neiheiser, Daniel Presser, Luciana Rech, Manuel Bravo, Luís Rodrigues, and Miguel Correia. Fireplug: Flexible and robust n-version geo-replication of graph databases. In *Proceedings of the 32nd International Conference on Information Networking*, January 2018.

[130] Nuno F. Neves, Miguel Correia, and Paulo Veríssimo. Wormhole-aware Byzantine protocols. In *2nd Bertinoro Workshop on Future Directions in Distributed Computing: Survivability - Obstacles and Solutions*, June 2004.

[131] Nuno F. Neves, Miguel Correia, and Paulo Veríssimo. Solving vector consensus with a wormhole. *IEEE Transactions on Parallel and Distributed Systems*, 16(12):1120–1131, 2005.

[132] Rafael Pass and Elaine Shi. Hybrid consensus: Efficient consensus in the permissionless model. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 91. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[133] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

[134] Morgen E Peck. Blockchains: How they work and why they'll change the world. *IEEE Spectrum*, 54(10):26–35, 2017.

[135] Morgen E. Peck. Why the biggest Bitcoin mines are in China. *IEEE Spectrum*, 54(10), 2017.

[136] Fernando Pedone, André Schiper, Péter Urbán, and David Cavin. Solving agreement problems with weak ordering oracles. In *Proceedings of the 4th European Dependable Computing Conference*, pages 44–61, 23–25 October 2002.

[137] Daniel Porto, Joao Leitao, Cheng Li, Allen Clement, Aniket Kate, Flavio Junqueira, and Rodrigo Rodrigues. Visigoth fault tolerance. In *Proceedings of the 10th ACM European Conference on Computer Systems*, 2015.

[138] Dan RK Ports, Jialin Li, Vincent Liu, Naveen Kr Sharma, and Arvind Krishnamurthy. Designing distributed systems using approximate synchrony in data center networks. In *12th USENIX Symposium on Networked Systems Design and Implementation*, pages 43–57, 2015.

[139] Michael O. Rabin. Randomized Byzantine generals. In *Proceedings of the 24th Annual IEEE Symposium on Foundations of Computer Science*, pages 403–409, November 1983.

[140] Michael K. Reiter. A secure group membership protocol. In *Proceedings of the 1994 IEEE Symposium on Research in Security and Privacy*, pages 176–189, May 1994.

[141] Michael K. Reiter. Secure agreement protocols: Reliable and atomic group multicast in Rampart. In *Proceedings of the 2nd ACM Conference on Computer and Communication Security*, pages 68–80, November 1994.

[142] Nicola Santoro and Peter Widmayer. Time is not a healer. In *Proceedings of the 6th Symposium on Theoretical Aspects of Computer Science*, pages 304–313, 1989.

[143] André Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10:149–157, October 1997.

[144] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[145] Marco Serafini, Péter Bokor, Dan Dobre, Matthias Majuntke, and Neeraj Suri. Scrooge: Reducing the costs of fast Byzantine replication in presence of unresponsive replicas. In *Proceedings of the IEEE/IFIP 40th International Conference on Dependable Systems and Networks*, pages 353 –362, July 2010.

[146] Marco Serafini and Neeraj Suri. The fail-heterogeneous architectural model. In *Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems*, pages 103–113, 2007.

[147] Slushpool. Stratum mining protocol. https://slushpool.com/help/manual/stratum-protocol.

[148] Yonatan Sompolinsky and Aviv Zohar. Secure high-rate transaction processing in Bitcoin. In *International Conference on Financial Cryptography and Data Security*, pages 507–527, 2015.

[149] João Sousa, Alysson Bessani, and Marko Vukolic. A Byzantine fault-tolerant ordering service for Hyperledger Fabric. In *Proceedings of the 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2018.

[150] Nick Szabo. The idea of smart contracts. http://szabo.best.vwh.net/smart_contracts_idea.html.

[151] Sam Toueg. Randomized Byzantine agreements. In *Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing*, pages 163–178, August 1984.

[152] Trusted Computing Group. TPM main specification level 2 version 1.2, revision 116. https://trustedcomputinggroup.org/tpm-main-specification/, 2011.

[153] UK Government Office for Science. Distributed ledger technology: beyond block chain. A report by the UK Government Chief Scientific Adviser, 2016.

[154] Paulo Veríssimo. Uncertainty and predictability: Can they be reconciled? In *Future Directions in Distributed Computing*, volume 2584 of *Lecture Notes in Computer Science*, pages 108–113. 2003.

[155] Paulo Veríssimo. Travelling through wormholes: A new look at distributed systems models. *SIGACT News*, 37(1):66–81, 2006.

[156] Giuliana Santos Veronese, Miguel Correia, Alysson N. Bessani, and Lau Cheuk Lung. Spin one's wheels? Byzantine fault tolerance with a spinning primary. In *Proceedings of the 2009 28th IEEE International Symposium on Reliable Distributed Systems*, pages 135–144, 2009.

[157] Giuliana Santos Veronese, Miguel Correia, Alysson N. Bessani, and Lau Cheuk Lung. EBAWA: Efficient Byzantine agreement for wide-area networks. In *Proceedings of the 12th IEEE International Symposium on High-Assurance Systems Engineering*, pages 10–19, November 2010.

[158] Giuliana Santos Veronese, Miguel Correia, Alysson N. Bessani, Lau Cheuk Lung, and Paulo Veríssimo. Efficient Byzantine fault tolerance. *IEEE Transactions on Computers*, 62(1):16–30, 2013.

[159] Marko 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, 2015.

[160] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. Ethereum Project Yellow Paper, EIP-150 Revision, 2014.

[161] World Economic Forum. The future of financial infrastructure. An ambitious look at how blockchain can reshape financial services, 2016.

[162] Pieter Wuille, Peter Todd, Greg Maxwell, and Rusty Russell. Version bits with timeout and delay. BIP-0009, October.

[163] Piotr Zielinski. Paxos at war. Technical Report UCAM-CL-TR-593, Univ. of Cambridge Computer Laboratory, Cambridge, UK, June 2004.