The Append Memory Model: Why BlockDAGs Excel Blockchains

Darya Melnyk
dmelnyk@ethz.ch
ETH Zurich
Zurich, Switzerland

Roger Wattenhofer
wattenhofer@ethz.ch
ETH Zurich
Zurich, Switzerland

ABSTRACT

This paper presents a novel shared memory model that simplifies the analysis of consensus on a Chain and a DAG. In this new model, referred to as the append memory model, nodes are allowed to write new values to the unordered memory, but not to overwrite already existing values. We show that although this model differs from the standard shared memory model with n shared read-write registers, many known results from the shared memory model still hold in the append memory model: It is, for example, impossible to establish consensus on n nodes with one crash failure if the nodes in the system are asynchronous. We also consider the append memory model in a synchronous setting with Byzantine failures. For this case, we show that Byzantine agreement cannot be solved in less than $t + 1$ rounds, where t is the number of Byzantine nodes in the system. Assuming a probabilistic access restriction to the append memory, we compare the Byzantine agreement protocols on the Chain and the DAG. We show that the DAG structure achieves an almost optimal resilience (close to $t < n/2$) in contrast to the Chain structure that can tolerate less than $t < \frac{n}{1+\lambda\cdot(n-t)}$ Byzantine nodes, where λ is the rate at which the nodes access the memory.

CCS CONCEPTS

• **Computing methodologies → Distributed algorithms**; **Distributed computing methodologies**.

KEYWORDS

Chain, DAG, Shared Memory, Byzantine Agreement

ACM Reference Format:
Darya Melnyk and Roger Wattenhofer. 2020. The Append Memory Model: Why BlockDAGs Excel Blockchains. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '20), July 15–17, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3350755.3400272

1 INTRODUCTION

Blockchain systems are world-scale systems. Consequently, blockchain research is carried out in the message passing model. In this paper, we propose to examine blockchain systems from a *shared memory* point of view. In the past, considering distributed systems from the shared memory viewpoint has given the research community valuable insights, and this is not different for the new area of blockchains. In the course of our studies, we found out that shared memory simplifies reasoning. As such, shared memory is a valuable model that can help us understand the fundamentals of blockchains.

In this paper, we will introduce a new shared memory model called the *append memory* model. It is as a variant of the shared memory model, where data can only be appended (but not modified) in memory. On the one hand, append memory is stronger than standard shared memory, since all written commands appear in the memory. On the other hand, it is also weaker because two concurrent writes to the same register in the shared memory result in only one value being written, whereas in the append memory model such ties cannot be broken. We will show that the append memory model obeys some of the same fundamental impossibility results of asynchronous consensus, and that Byzantine agreement cannot be established in less than $t + 1$ rounds in the synchronous model.

Our main result will assume a probabilistic access to the append memory. This model variant is a clean version of message passing proof-of-work blockchains; it allows us to neatly examine one of the main open blockchain research questions: Are DAG-based blockchains superior to tree-based blockchains? In order to understand their difference, we compare Byzantine agreement on a classic tree-based blockchain and a DAG-based blockchain. We show that the DAG achieves an almost optimal resilience (close to $t < n/2$). An orthodox chain, on the other hand, tolerates less than $t < n/(1 + \lambda(n - t))$ Byzantine failures, where λ is the access rate of a node to the memory.

1.1 Model

The model consists of n nodes (in the literature often also referred to as processors), $v_1, \ldots, v_n$, that communicate via a shared memory and aim to establish a relative order on the messages written to the memory. The shared memory consists of n registers, each associated with exactly one node in the system. Each of the registers R_i is unbounded in space and formally supports two operations - R_i.read() and R_i.append(`msg`). We will therefore name this shared memory the *append memory*. The R_i.read() operation can be executed by any node in the system and it returns a complete view of the register R_i. The R_i.append(`msg`) operation can only be executed by the node v_i and it appends the message `msg` to the current state of the memory without removing any previous information from R_i. Since the idea of this memory is to establish a relative order of the messages written to the memory, we assume that a message `msg` from v_i contains some value from this node and a reference to a previous state of the memory that is defined by the underlying protocol. We assume that the appended messages, just like the registers, are unbounded in space.

In the above definition of the append memory, the n registers can also be viewed as one single register M to which all nodes append their values, with the exception that the single register cannot establish any order on the appended messages. These messages are instead weakly ordered using references to previous states from the underlying protocol. We will therefore also define the M.read() operation, which is going to read the whole memory. It corresponds to executing R_i.read() for all i. Analogously, the M.append($\texttt{msg}_i$) operation appends a new message at any place in M, which corresponds to executing R_i.append(msg) in the previous definition. Observe that the single registers R_i may establish a total ordering of all messages corresponding to the node v_i. This ordering can also be incorporated in the single register view by forcing all nodes to refer to their previous appends in the protocol.

Each node is assumed to have a binary input value at the beginning of the protocol. The nodes then communicate through the append memory in order to agree on a common output value. We differentiate between *correct* and *faulty* nodes in the protocol. The correct nodes follow the protocol at all times. For faulty nodes, we consider two settings: they can either have a *crash failure* or a *Byzantine failure.* Nodes that crash just stop executing the protocol at an arbitrary point in time. Byzantine nodes can, on the other hand, deviate from the protocol in any possible way, thus making their behavior unpredictable. The task of a protocol is to establish consensus on one of the input values in the presence of faulty nodes while satisfying the standard consensus properties:

Agreement: All correct nodes decide on the same value.

Termination: All correct nodes must terminate after executing a finite number of operations.

Validity: If all correct nodes have the same input value b, they must agree on b at the end of the protocol.

These three consensus properties can also be weakened such that each of the properties is only satisfied with high probability (w.h.p.). We call the relaxed properties *weak agreement, weak termination* and *weak validity* respectively. They will be necessary for the discussion of randomized consensus algorithms in the append memory model. The corresponding agreement problems will be then referred to as *weak consensus* or *weak Byzantine agreement.*

We will also discuss different communication models in this paper. Since all nodes are communicating through a shared memory we assume that the messages that are appended to the memory instantly become available for other nodes to read. The nodes themselves do, however, not have to be synchronized, and can perform their operations at different points of time. We therefore make use of the definitions similar to Dolev et al. [6] and differentiate between synchronous and asynchronous nodes:

Synchronous: There exists a constant $\Delta > 0$ such that any interval between two operations executed locally by a single node is bounded from above by Δ. The upper bound Δ is known to all nodes.

Asynchronous: The time between two operations of a node is not bounded. However, in an infinite protocol run, each correct node must perform infinitely many operations. Otherwise, the node is called faulty.

The definitions so far do not put any restriction on the access to the memory. In the synchronous and the asynchronous model, the nodes decide locally and independently of other nodes when the memory will be accessed next. In Section 5, we will consider an alternative model inspired by the proof of work, where access to the memory is restricted by a Poisson process. We will assume that all nodes can read the memory at any time. An append operation, however, will require a token that is given to the node by some authority who controls the access:

Randomized Memory Access: The access probability to the append memory model for each node v inside the time interval Δ is a Poisson distributed random variable X_v with rate λ. All random variables $X_v, v \in \{v_1, \ldots, v_n\}$ are independent and therefore the access rate to the memory by all nodes is described by the random variable $Y := \sum_v X_v \sim \text{Pois}(\lambda n)$

Note that the proposed append memory model deviates from the standard shared memory models in several ways: the append memory cannot order the access threads from different nodes, as the ability to do so would directly imply consensus. This assumption is inherited from the message passing model, where two nodes that received the same two messages might have received them in an opposite order. Moreover, the nodes in the append memory have the ability to read the whole memory content with one memory access. This assumption also comes from message passing systems where the nodes receive messages (appends) from other nodes while participating in the protocol and thus reconstruct the whole memory content. Also the randomized memory access is unusual for shared memory models, where the access is usually controlled by the memory itself. Since the append memory withdraws the power of ordering messages from the memory, an access strategy on the protocol side is required in order to be able to establish a weak ordering.

1.2 Related Work

The problem of sharing data among several processors in a system has been well studied in the literature. Early solutions to this problem required mutual exclusion [4, 5], i.e. only one process was allowed to access and alter the memory at a time while the other processors were denied access. The first discussion on the wait-free implementation of shared objects goes back to Herlihy [12]. In this paper, he defined the *consensus number* as the maximum number of nodes that can establish consensus in the system using arbitrarily many shared objects. According to this definition, a hierarchy of shared objects can be established. In particular, Herlihy showed that the consensus number of read-write registers is 1, i.e. consensus cannot be established by two processors using read-write registers.

Byzantine agreement in the shared memory was first considered by Malkhi et al. [16]. They used the concept of sticky bits [20] and access control lists in order to restrict Byzantine power. Sticky bits are bits that remain in the memory and cannot be overwritten. They also showed that Byzantine agreement is impossible in their model if $t > n/3$ and provided a protocol that could tolerate $(\sqrt{n} - 1)/2$ Byzantine failures. Alon et al. [2] later showed that the bound on the resilience is tight by using exponentially many sticky bits.

The first blockchain protocol was introduced by Satoshi Nakamoto [17], who invented Bitcoin – a distributed cryptocurrency system based on peer-to-peer communication and the construction of a Blockchain. The Nakamoto consensus needs to satisfy two main properties: consistency and liveness. The first rigorous analysis of Nakamoto consensus on a Blockchain was given by Garay et al. [9]. In their work, they analyzed the blockchain in the synchronous communication network, assuming that all messages of the current round arrive at the beginning of the next round. They showed that Nakamoto consensus solves Byzantine agreement with validity under the assumption that the Byzantine nodes have strictly less than $1/3$ of the hashing power of the network. They further propose a more elaborate consensus protocol on the Blockchain for which they show a resilience of up to $1/2$. Many following attempts were made in order to formalize Nakamoto consensus in a more general model: Pass et al. [18] extend the synchronous model of [9] and consider a δ-synchronous network, where δ is an upper bound on message delay known to all nodes in the network. Pass and Shi [19] later simplified the previous model mostly for didactic purposes. Another formalization of consistency of Nakamoto consensus was given by Kiffer et al. [13], who use a Markov chain based analysis to prove consistency in a synchronous setting.

All aforementioned papers note that the analysis of Nakamoto consensus generally is involved, and therefore needs complicated models in order to describe their system rigorously. The first simple analysis of Nakamoto consensus was provided by Ling Ren [21]. Instead of focusing on the definition of the communication model, [21] focuses on a correct analysis of the chain growth and therefore differentiates between blocks which are "non-tailgaters" and "loners". The former describes blocks which were mined after seeing the last correct block in the system, while the latter denotes blocks which are non-tailgaters and are not non-tailgated. This way, both, forks in the Blockchain introduced by the Byzantine nodes, as well as forks produced by the correct nodes, are taken care of. Ren further shows that Nakamoto consensus satisfies consistency and liveness under the honest majority assumption, provided that the block generation rate of the correct nodes is much larger than the communication delay.

Other structures for reaching Nakamoto consensus have also been considered in the literature. In [22], it was shown that the so-called inclusive Blockchain, which relies on the DAG structure, can provide safety in the Blockchain protocol even if the system is asynchronous for a short period of time. The DAG structure is usually considered under one of the tie-breaking rules, such as the GHOST protocol [22] or the pivot chain [14] rule.

Observe that, while many protocols also considered Byzantine agreement besides Nakamoto consensus, consistency and liveness actually do not necessarily require consensus as a building block. This was first shown by Gupta [11]. In a follow-up work, Guerraoui et al. [10] show that Nakamoto consensus has a consensus number 1. Other than the protocols mentioned previously, such systems work in the fully asynchronous setting but does not satisfy consistency at any point of time in the protocol, and would therefore require checkpointing techniques in order to be applied in cryptocurrency systems.

1.3 Our Contribution

Our contribution deviates from the previous work in several ways. Many papers try to directly solve Nakamoto consensus in the message passing model, thereby oversimplifying the communication model [9] or falsely calling a synchronous communication system (partially) asynchronous [13, 18]. Instead, we focus on deriving a shared memory model for Blockchain protocols, which allows us to assimilate the local views of the nodes and thereby derive simpler protocols for Blockchain and DAG. Note that the append memory is not as strong as the concept of sticky bits [16] since it does not make use of registers that implicitly solve consensus for two parallel writes. In Section 2, we therefore show that asynchronous consensus cannot be solved in this append memory model, as the nodes cannot uniquely define the ordering of concurrently appended commands. In Section 5, it is shown that this result also holds for the asynchronous communication model with randomized memory access. We further show that our proposed model is not stronger than the message passing model, as it can be simulated in the message passing at a high message complexity cost (see Section 4). The advantage of the append memory model is that it simplifies the analysis of Blockchain protocols. In Section 5, we will therefore compare the analysis of the DAG and the Chain in the append memory. We will show that Byzantine agreement on the DAG can achieve almost optimal resilience of $< 1/2$, while Byzantine agreement on the Chain highly depends on the append rate of the correct nodes. Our results suggest that the DAG is not only a better model for Byzantine agreement because of its simplicity compared to the Chain, but also because it achieves an optimal resilience.

2 IMPOSSIBILITY OF ASYNCHRONOUS DETERMINISTIC CONSENSUS IN THE APPEND MEMORY

The append memory provides a common history of the appended commands to all nodes participating in the consensus. In this section, we will show that it is impossible to reach consensus in the append memory if the processors in the system are asynchronous and at least one of the processors may crash. With respect to the append memory, the definition of asynchronous nodes says that an arbitrary amount of time can pass between a read and an append operation, meaning that a node might append to an obsolete state of the memory. Dolev et al. also provide a definition for synchronous and asynchronous communication, and show possibility and impossibility results for different communication and processor settings. Their impossibility results hold for the message passing model and rely on the fact that the buffers of the nodes may receive messages in a different order. Such an assumption cannot be made in the append memory since the append memory establishes a common view of the system to all nodes and the states of the system are defined by the point of time at which a node reads the memory.

We will follow the outline of the impossibility proof of Loui and Abu-Amara [15] who showed that it is impossible to reach consensus in the shared memory with read-write protocols. Among other results, the authors provide a proof of impossibility for t-resilient read-write protocols, where the number of read and write operations in the system is unbounded. Our analysis will deviate from the one by Loui and Abu-Amara because our append memory does snot allow processors to overwrite the memory cells.

Theorem 2.1. *There exists no t-resilient consensus protocol in the append memory for all $n > 2$.*

2.1 Definitions

A *consensus protocol* is a system that consists of n nodes $V = \{v_1, \dots v_n\}$. Each node is equipped with an initial bit which is the nodes input value. Since the nodes communicate via an append memory, all nodes have read access to the append memory M. We say that the state of a node is defined by the nodes current value and its last read of the append memory, i.e. if a node last read the memory at time τ, its local view of the memory will be $M(\tau)$. The state of the node from the last read operation is denoted by $s_i = (M(\tau), \mathtt{val}_i)$. A *configuration* C of the system is defined as the set of the states $\{s_1, \dots, s_n\}$ of all n nodes in the system together with the current view of the memory $M(\tau^*)$. We therefore define this configuration to be $C := \{s_1, \dots, s_n\} \times M(\tau^*)$. The *initial configuration* of the system consists of the initial bits of the nodes and an empty view of the memory and $M(0) = \{\varnothing\}$, denoted C_0. Each node supports the read and append operations. We will call an execution of a read or an append operation by a node v an *event* e_v. We assume that the nodes are asynchronous. In this system, an event e_v can always be *applied* to a configuration C, if the event is a read operation. If the event is an append operation, it can only be applied to C if it follows the construction rules of the append memory. Let the current state of node v in the configuration C be $s_i = (M(\tau), \mathtt{val}_i)$. An event e_v applied to C at time τ' transitions C to a new configuration $e_v(C)$ in the following way:

(a) e_v is a read operation of the append memory with $M(\tau) \subsetneq M(\tau')$: node v will possibly update its value $\mathtt{val}_i$ and transition to a new state $s'_i = (M(\tau'), \mathtt{val}_i)$. The corresponding configuration C will transition to the configuration $e_v(C)$ where $e_v(C) = \{s_1, \dots, s_{i-1}, s'_i, s_{i+1}, \dots, s_n\} \times M(\tau')$.

(b) e_v is a read operation of the append memory at time $\tau' > \tau$, with $M(\tau) = M(\tau')$: node v will not change its state and therefore $e_v(C) = C$ holds.

(c) e_v is an append operation executed at time τ' for the view $M(\tau)$ of the append memory: according to the definition of the append memory, a value from the node v can be appended to an obsolete state of the memory as long as it does not contradict the order of messages of v in the current append memory state $M(\tau^*)$. A state is considered obsolete with respect to a weak ordering defined by the underlying protocol. Let $M(\tau')$ be the state of the append memory after event e_v has been applied to it. Then, the new state of the system is described by $e_v(C) = \{s_1, \dots, s_n\} \times M(\tau')$.

We say that a configuration C' is *accessible* from another configuration C if there is a sequence $e_1, \dots, e_j$ of applicable events, such that $C' = e_i\,(e_{i-1}\,(\dots e_1(C)))$. In the following, we define a *computation graph* G for an algorithm $\mathcal{A}$ which describes the accessible configurations: The vertices of G contain all possible initial configurations of the protocol as well as all configurations accessible from these states. There is a directed edge from a configuration C to another configuration C', iff there exists an event e which is applicable to C such that $C' = e(C)$ holds. In this case e will be the label of the edge from C to C'. Note that Property (b) allows self-loops at each configuration of the computation graph. Note that a configuration C' is accessible from a configuration C if there is a directed path from C to C' in G.

Next, we define a *computation* of $\mathcal{A}$ as a (not necessarily finite) sequence of configurations $C_0, C_1, C_2, \dots$, such that for each pair of consecutive configurations C_i, C_{i+1} there exists a directed edge from C_i to C_{i+1} in the computation graph G. A configuration is said to have a *decision state* if one of the nodes in the configuration has decided on one of the values in $\{0, 1\}$. A computation *terminates* in a configuration C_j if every correct node in C_j has reached a decision state.

Since algorithm $\mathcal{A}$ solves consensus, in some of the configurations there will be nodes which have reached a decision state. Also, the algorithm has to satisfy the consensus properties from Section 1.1:

- $\mathcal{A}$ satisfies agreement, if every configuration has at most one decision state.
- $\mathcal{A}$ satisfies termination, if for every initial configuration C_0 every computation terminates.
- $\mathcal{A}$ satisfies validity, if for the input configuration where every node has input value 0, i.e. $C_0^{(0)} := \{(\varnothing, 0), (\varnothing, 0), \dots, (\varnothing, 0)\} \times \{\varnothing\}$, every computation terminates with every correct node deciding 0. Analogously, every computation starting in $C_0^{(1)} := \{(\varnothing, 1), (\varnothing, 1), \dots, (\varnothing, 1)\} \times \{\varnothing\}$ terminates with every correct node deciding 1.

Let $\mathcal{C}(C)$ be the set of decision values which are accessible from configuration C under $\mathcal{A}$. We say that C is *bivalent*, if $\mathcal{C}(C) = \{0, 1\}$. C is called univalent if $|\mathcal{C}(C)| = 1$. In particular, C is 0-valent if $\mathcal{C}(C) = 0$ and it is 1-valent if $\mathcal{C}(C) = 1$.

So far, we have only addressed correct nodes in the definition. We call a node correct if for an infinite computation the node takes infinitely many steps. Otherwise, the node is faulty. A computation that that has no events by some node v is called *v-free.* An algorithm $\mathcal{A}$ is 1-resilient, if for any $v \in V$ and any reachable configuration C in G all v-free computations terminate.

2.2 Proof of Theorem 2.1

The proof of Theorem 2.1 is organized as follows: In Lemma 2.2 we will first present that for any algorithm $\mathcal{A}$ which satisfies the properties of consensus there exists a bivalent initial configuration C_0. In Lemma 2.3 we will show that for any bivalent configuration C and any node v there exists another reachable bivalent configuration C', such that on the directed path from C to C' v performs an event. Finally, we will show how these lemmas can be used in order to establish an infinite computation starting in C_0 in which every correct node performs infinitely many events.

Lemma 2.2. *There exists a bivalent initial configuration of the system if $t \geq 1$.*

Proof. The proof is based on the fact that the validity condition has to be satisfied for both input values and is almost identical to proof in [15]. Will therefore skip it at this point. □

Lemma 2.3. *Let C be a bivalent configuration and e_p be an applicable event to C. Let $\mathcal{D}$ be the set of configurations reachable from C that do not contain any events from p. Then, there is a bivalent configuration $C' \in e_p(\mathcal{D})$ reachable from C.*

Proof. Observe first that the event e_p is applicable to every configuration in $\mathcal{D}$: if e_p is a read command, it is applicable to every configuration in $\mathcal{D}$. On the other hand, if e_p is an append command, it can either be appended to the configuration C, or it can be appended to any future configuration as the nodes are asynchronous and there can be an arbitrary delay between a read and an append command in the system. Let $e(\mathcal{D})$ be the set of all configurations that result from $\mathcal{D}$ by applying the event e_p to them. For the rest of the analysis, assume by contradiction that there are only univalent configurations in $e_p(\mathcal{D})$.

We will first show that $e_p(C \cup \mathcal{D})$ contains both, 0- and 1-valent configurations: By definition, C must contain 0- and 1-valent configurations since it is bivalent. Assume w.l.o.g. that $e_p(C)$ is a 0-valent configuration. Note that all configurations that are reachable from $e_p(C)$ must also be 0-valent. Therefore, there must exist a configuration $D \in \mathcal{D}$ which is 1-valent. Then, the configuration $e_p(D) \in e_p(\mathcal{D})$ is also 1-valent. Further, there exist two configurations D_0 and D_1 in $C \cup \mathcal{D}$ such that $D_1 = e_q(D_0)$ and such that $E_0 := e(D_0)$ is 0-valent and $E_1 := e(D_1)$ is 1-valent. This follows by an induction argument on the path from C to D.

We differentiate between four cases for the events e_p and e_q:

e_p and e_q are read events: If both events are read events, then the corresponding events are commutative since they do not change the view of the memory. Consider the configuration $e_q(E_0)$. Since E_0 is a 0-valent configuration, $e_q(E_0)$ must also be 0-valent. However, since e_p and e_q are commutative, $e_q(E_0)$ is equal to the configuration E_1, which is a 1-valent configuration. This is a contradiction.

e_p and e_q are append events: We need to show that the append events are also commutative. Note that the append events are consecutive and that neither of the nodes p or q has a read event in between. if both nodes previously read the same view of the memory, they will append to the same view in the memory. Any later read operation will not be able to differentiate which of the appends was written to the memory first. The same property holds if the nodes append to different views of the memory. Since the nodes are asynchronous, they can always append to previous views of the memory and therefore it is not possible to determine the order of the appends. Since the events are commutative, the same analysis applies as in the first case.

e_p is a read and e_q an append event: Note that if e_p is a read configuration, it does not change the view of the memory, i.e. in configurations D_0 and E_0 the view of the memory is identical. Therefore, if e_q is applied to either of the configurations, the configurations of all nodes excluding p are the same in D_1 and $e_q(E_0)$. As the node p might crash during the execution of the algorithm and since the algorithm is deterministic, by applying the same set of events, the remaining nodes must end up in the same configurations independent of whether they started in D_1 or in $e_q(E_0)$. This is a contradiction since the corresponding configurations have different valencies.

e_p is an append and e_q is a read event: In this case we can assume that node q crashes after reaching the configurations E_0 or E_1. The rest of the analysis is analogous to the previous case. □

Proof of Theorem 2.1. Let $\mathcal{A}$ be a deterministic algorithm that solves consensus in the append memory and can tolerate crash failure. We will prove Theorem 2.1 by showing that there exists a scheduling of events under which $\mathcal{A}$ will not terminate. Note that there exists an initial configuration C_0 that is bivalent according to Lemma 2.2. From this configuration on, we consider a sequence of events in which each node takes infinitely many turns: let $v_1 \dots v_n$ be some ordering of all nodes in the system. By Lemma 2.3 there exists a path from the bivalent configuration C_0 to a bivalent configuration C_1, in which the node v_1 executes an event e_{v_1}. By applying Lemma 2.3, we find another path from the bivalent configuration C_1 to a bivalent configuration C_2, in which the node v_2 executes an event. Lemma 2.3 can be applied infinitely many times in a round robin fashion to the nodes in the sequence $v_1 \dots v_n$. Since each new configuration C_i is a bivalent configuration, $\mathcal{A}$ will not terminate for any node in the system. This concludes the impossibility proof.

3 CONSENSUS IN THE APPEND MEMORY WITH SYNCHRONOUS NODES

In this section we will discuss possibility and impossibility results for the introduced append memory when the nodes in the system are synchronous. We will first show that the lower bound on the number of rounds needed to solve Byzantine agreement is $t + 1$. We will achieve this bound by slightly adapting the proof of Aguilera and Toueg [1] which was originally introduced for the synchronous consensus in the message passing model with crash failures. The advantage of this proof in comparison to the lower bounds in [7, 8] is that the authors use the notation of univalent and bivalent configuration which we have introduced in the previous section. Other than in the mentioned papers, our lower bound will only hold for the Byzantine model. The previous papers assume that a crashed node can send messages to a subset of the nodes in the system before crashing. This cannot happen in the append memory since the nodes only communicate with one authority (which has control over the memory). Therefore, all values that have reached the memory will be available to all correct nodes after a time interval of Δ. This implies that agreement with crash failures can be solved in the append memory with synchronous nodes within one round only.

In the case of Byzantine failures, the situation becomes different. A Byzantine node can exploit the small asynchrony of Δ in the reads of the nodes such that its append commands will be read by a subset of the nodes in the same round. This gives us the possibility to apply the results from Aguilera and Toueg [1] as we will explain in the next section. In Section 3.2, we will provide a matching upper bound which is based on the upper bounds for interactive consistency in the work of Dolev and Strong [7].

3.1 Lower Bound on the Number of Rounds

In this section, we will make use of the notation presented in Section 2.1. Since we consider a round based algorithm, a configuration C will be associated with the configuration at the end of a round. A round is defined as a communication step with the memory, which includes at most one append and one read operation per node. A transition from one configuration to the next can be described by a combination of append and read operations from all n nodes.

Lemma 3.1. *For any round i with $0 < i \leq t$, where t denotes the number of Byzantine nodes in the system, holds: if at the end of round $i - 1$ the system was in a bivalent configuration, there is a*

Algorithm 1 Byzantine Agreement with Synchronous Nodes (code for node v)

```
Input: append memory $M$, input value $\mathtt{val}(v)$
1: for round $r = 1, \dots, t+1$ do
2:     $M$.append($\mathtt{val}(v), L_{r-1}$), where $L_0 := \{\emptyset\}$
3:     Wait for $\Delta$ time
4:     $M$.read and let $L_r$ be the set of all appended commands in
       Round $r$
5: end for
6: Let a value $\mathtt{val}(w)$ be accepted, if there exists a chain of $t+1$
   distinct nodes $v, w_1, w_2, \dots, w_t$ such that $(\mathtt{val}(v), \emptyset)$ is listed
   in $(w_1, L_1)$, $(w_1, L_1)$ is in $(w_2, L_2)$, ..., and $(w_{t-1}, L_{t-1})$ is in
   $(w_t, L_t)$
7: Decide on the majority of all accepted values in $\mathtt{val}(w)$
```

computation in which at the end of round i the system is again in a bivalent configuration.

Proof. We will prove this lemma by induction over i. In Lemma 2.2 we showed the base case, i.e. that there exists an initial bivalent configuration. By the induction hypothesis, we assume that C is the bivalent configuration at the end of round $i-1$. Next, we can assume that at most one node can exhibit Byzantine behavior per round, we call this node b_{i-1}. The power of b_{i-1} in the append memory lies in the fact that it can delay its own messages such that only part of the nodes will see its message in the memory in round i, and the other nodes will only be able to see it with the next read in round $i+1$.

Assume for contradiction that all configurations at the end of round i are univalent. Since C is bivalent, there must exist transitions to 0 and 1-valent configurations. Let the configuration which is reached by a transition where all nodes perform their actions correctly be 1-valent w.l.o.g. We denote this configuration C^1. Moreover, let C^0 be a 0-valent configuration which results through some transition from C. Note that the transitions $C \rightarrow C^0$ and $C \rightarrow C^1$ only differ in the actions of b_{i-1}, since all other nodes behave correctly and deterministically. Similar to Lemma 2.2, we can construct a sequence of neighboring configurations that differ in the view of one node only such that they have to have the same valency. The construction is the same as in [1]: Let S denote the (possibly empty) set of nodes that see the append of b_i in the memory in the configuration C^0. Consider a configuration C' which results from C^0 by letting an additional node $v \notin S$ see the append of b_i. Note that C^0 and C' are indistinguishable if v fails in round i and therefore both have to be 0-valent. We can continue adding nodes one by one to S and apply the previous argument repeatedly to show that the configuration C^1 also has to be 0-valent. This is a contradiction to C^1 being 1-valent, and thus there must exist a bivalent configuration in round i. □

Note that Lemma 3.1 implies that Byzantine agreement in the append memory cannot be solved with synchronous nodes in less than $t+1$ rounds. In particular, the lemma shows that there exist a bivalent initial configuration and a t-round computation such that the system ends up in a bivalent configuration at the end of round t. Therefore, the nodes need at least $t+1$ rounds in order to reach a univalent configuration and thus achieve agreement.

3.2 A Simple Deterministic Algorithm with Synchronous Nodes

The idea for the synchronous deterministic algorithm in the append memory is similar to the interactive consistency idea in Byzantine Agreement [7]. In interactive consistency algorithms, in every round nodes forward complete views of the system to all other nodes. After $t+1$ rounds, a decision can be made about whether to accept a proposed input value. This results in exponential information exchange. In our case, the views of the nodes are almost the same, since all nodes have access to the append memory. The only differences in the views of the memory can appear through the Byzantine strategy that was described in the previous seection. Algorithm 1 shows a possible implementation of Byzantine agreement with synchronous nodes.

Theorem 3.2. *Algorithm 1 solves Byzantine Agreement in the append memory for $t < n/2$ Byzantine nodes within $O(t\Delta)$ time.*

Proof. In order to prove the theorem, we need to show that Algorithm 1 satisfies termination, agreement, and validity. Termination is trivially satisfied since all nodes execute the algorithm for $O(t\Delta)$ time and decide. We will show that agreement is satisfied because all nodes will agree on whether to accept each input value or not, i.e. their decision will be based on the same input set. Validity will follow from agreement by showing that additionally to equivalent views, all nodes will accept all correct input values.

Note at first that every correct value will be accepted in Line 6, since there are $n-t > t+1$ correct nodes: Each correct node will see all correct values appended in the previous round. Therefore, each correct append will be listed in $n-t$ correct appends of the next round, i.e. there will be at least $t!$ chains starting in any correct value which satisfy the condition in Line 6.

Consider next a Byzantine input value b_1. Assume that no correct node contains this value in the set L_1, otherwise, the value will be accepted. b_1 cannot be accepted if no other Byzantine node contains this value in its set L_1. Assume next that the node b_2 contains b_1 in its set L_1. Then, either a correct or some other Byzantine node has to contain this append in their set L_2. Since the chain in Line 6 needs to have $t+1$ distinct elements and since there are at most t Byzantine nodes, at least one correct node has to contain some append from the Byzantine chain for the value b_1 to get accepted. If one correct node has a value from the Byzantine chain in its set L_i, every correct node will read this set and either accept the value b_1, if $i = t$, or extend the chain by referring to the correct append in the set L_{i+1}. Therefore, a byzantine value will be accepted by the algorithm iff at least one correct node extends the chain of Byzantine appends. □

4 SIMULATION OF THE APPEND MEMORY WITH MESSAGE PASSING

In this section, we show how the append memory can be simulated through the message passing model using a simulation similar to the ABD simulation [3]. This completes our theoretical analysis of the append memory and shows that it is a suitable abstraction for different Blockchain and DAG algorithms. In particular, the previous two sections already imply that asynchronous consensus

Algorithm 2 Simulation of M.append() (code for node v)

```
Input: append value val(v), local memory view $M_v$
1: Broadcast append(val(v))$_v$
2: upon receiving a message from node $w$ do
3:     Append message from $w$ to $M_v$
4:     Broadcast ack(append(val(w))$_w$)$_v$
5: end upon
6: upon receiving ack(append(val(v))$_v$)$_w$ from $> n/2$ nodes $w$ do
7:     terminate append operation
8: end upon
```

Algorithm 3 Simulation of M.read() (code for node v)

```
1: Broadcast $M$.read()
2: upon receiving $M$.read() from node $w$ do
3:     Send local view $M_v$ to $w$
4: end upon
5: upon receiving $M_w$ from $> n/2$ nodes $w$ do
6:     Append all newly seen values in the local views $M_w$ to $M_v$ and terminate
7: end upon
```

is impossible in the append memory model as well and that the lower bound on the number of rounds for deterministic Byzantine agreement in the synchronous model is the well-known bound of $t+1$ rounds. This already shows that our append memory abstraction is not stronger than the results achieved in the message passing model. Here we will show that the message passing model can naturally simulate the append memory model if the nodes sign their messages and assuming that these signatures cannot be forged.

Algorithm 2 and 3 respectively present the simulation of the append and the read abstractions. Note that the correct nodes need to be *available* at all times, i.e., they always have to respond to append and read messages from other nodes. We will use signatures in order for the nodes to be able to prove that another node sent them a message. We will denote a message val(v) signed by node v by (val(v))$_v$.

Lemma 4.1. *Algorithm 2 correctly simulates an M.append(val(v)) operation in the append memory.*

Proof. Note at first that an M.append(val(v)) operation from a correct node will always reach all other correct nodes. Therefore, all correct nodes will append a correct value to their local view of M, and node v will receive $> n/2$ acks for its message.

Since Byzantine nodes cannot forge the signatures of the correct nodes, their local view will either contain a message from some correct node or no message at all. □

Lemma 4.2. *Algorithm 3 correctly simulates an M.read() operation in the append memory.*

Proof. By requesting the views of the memory from more than $n/2$ nodes in the system, a node will receive all append commands added to the local views of the memory by all correct nodes. This is because an append of a node only terminates if $> n/2$ nodes appended the corresponding value to their local view of the memory and by requesting the memory view from $> n/2$ nodes, the append operation will be visible in at least one memory view.

Observe that Byzantine nodes can append multiple values in parallel by sending different messages to different nodes. This is not a contradiction to the correctness of the simulation as such behavior is also possible in the append memory. Since nodes that see two values from a Byzantine party in the append memory cannot distinguish which of the values has been appended first, both values have to be accepted by the correct nodes. The same is achieved in Line 6 of Algorithm 3, where all correct nodes accept all values. □

Note that we made use of signatures in order to make sure that a Byzantine party cannot pretend to have received a different value from the correct node than the value sent by the correct node itself. The above algorithms would also work without signatures. In that case, nodes can only append a value to their own local memory, if they have seen it in at least $f+1$ different views of the memories. Such an adjustment would, however, reduce the resilience of our protocol.

Our analysis shows that the append memory abstracts away the unnecessary communication overhead which often makes the discussion of algorithms in the message passing model difficult and heavy in terms of message complexity. Observe that the size of the local view of the memory increases over with each append operation. Thus, a simulation of an algorithms where all nodes participate, such as Algorithm 1, would lead to exponential information exchange.

5 APPEND MEMORY WITH RANDOMIZED ACCESS

The benefit of a randomized access strategy to the append memory is that algorithms in the permissioned and permissionless settings of Byzantine agreement can be considered. In the first setting, the number of nodes and the corresponding signatures are known to all participants, while in the latter setting, only the upper bound on the fraction of Byzantine nodes is known. In this section, we will focus on the permissioned setting when deriving the bounds. All the presented results can, however, be trivially extended to the permissionless setting as well.

We will first discuss the randomized access to the append memory with respect to the synchronous or asynchronous nodes. We will show that the impossibility result from Section 2 can also be applied to this model:

Theorem 5.1. *There exists no deterministic protocol that can solve Byzantine agreement with asynchronous nodes in the append memory with randomized access.*

Proof. Note that the definition of asynchronous nodes states that arbitrary time can pass between any two local operations of a node. The randomized access to the append memory as defined in Section 1.1 only gives out tokens to nodes, such that the nodes can use the token in order to append commands to the memory. As the time between any two operations is unbounded, we can assume that the time between receiving a token and appending a message to the memory is also unbounded. In the worst case, the delays are significantly larger than the append rate to the memory, such that the access order of the memory defined by the random access rule

Algorithm 4 Byzantine Agreement with Absolute Timestamps (code for node v)

```
Input: append memory $M$, input value val($v$)
1: $M$.read()
2: while there are less than $k$ writes in the memory do
3:     $M$.read()
4:     upon granted access to the memory do
5:         $M$.append(val($v$))
6:     end upon
7: end while
8: Order all appends by the timestamps
9: Decide on the sign of the sum of the first $k$ appends
```

becomes insignificant. Therefore, the proof of Theorem 2.1 can also be applied to this setting. □

The proof of Theorem 5.1 only works because the rate for memory access is independent of the delay resulting from the asynchrony of nodes. This suggests that it is reasonable to consider randomized access to the append memory model together with synchronous nodes. Then, the access rate to the memory can be connected to the maximum delay given between any two operations of the nodes.

In the following sections, we will assume that the input values of the nodes are -1 or $+1$. The decision value will then be determined as the sign of the sum of all accepted values. Note that by the definition of the random access, each node v_i is associated with a random variable $X_i \sim \text{Pois}(\lambda)$ which denotes the expected number of appends by node v_i during Δ. Let $X := \sum_{i=1}^{n} X_i$ be the variable denoting how many appends appear in expectation during the time Δ in the memory. Note that the variable X is also Poisson distributed with $X \sim \text{Pois}(\lambda n)$.

In Section 5.1, we will discuss the best possible scenario for the append memory, where each append is equipped with an absolute timestamp. This example will serve as a baseline for the resilience that can be achieved in our model. In Section 5.2, we will show that the chain rule, which is often used as a base structure in Blockchain protocols, only tolerates up to $t < n/(1 + \lambda(n - t))$ Byzantine nodes. Finally, in Section 5.3, we will show that Byzantine agreement in the append memory model with DAGs gives optimal resilience.

5.1 Byzantine Agreement with Absolute Timestamps

In this section, we assume that all appends to the memory will be equipped with an absolute timestamp handed out by a central authority upon appending a command to the memory. This way, all appends in the memory will have a unique ordering which is visible to all nodes. Algorithm 4 shows how Byzantine agreement can be solved in this model. Although agreement and termination will follow trivially, the validity condition can only be satisfied with high probability. In particular, the probability to satisfy the validity condition will depend on the number of appends to the memory and the difference between the number of correct and Byzantine nodes in the system.

Theorem 5.2. *Algorithm 4 satisfies agreement, termination and weak validity.*

Proof. Algorithm 4 satisfies agreement, because the timestamps uniquely determine the first k writes into the memory, and because all nodes have the same view of the memory. By choosing k to be an odd number, the sum of the first k values will be either positive or negative, thus determining the decision value. Termination is satisfied since there eventually will be k writes to the memory such that all nodes will leave the while loop in Line 2.

The validity condition can only be satisfied with high probability, as there is always a negligible probability that the first writes will all be from Byzantine nodes. In order to show validity of Algorithm 4, we will consider the sum of all written values as the sum of binomially distributed random variables. For large number of nodes n, this sum can be approximated by the normal distribution due to the central limit theorem. We can use the tail bounds for the normal distribution in order to finally show that the majority of the k coinflips will be from correct nodes with high probability.

Assume that the validity assumption holds, i.e. that all correct nodes have the same input bit $+1$. W.l.o.g. we can assume that all Byzantine nodes will write the value -1 to the memory. Otherwise, the Byzantine strategy would not be optimal. Note that with probability $p_{corr} = \frac{n-t}{n}$, each append to the memory is from correct node, while with probability $p_{byz} = \frac{t}{n}$, it is Byzantine. Next, we only consider the first k appends to the memory. Then, the probability for each append to be correct or Byzantine will follow the Binomial distribution. Let Y_i be the random variable defining the value of the i-th append in the memory. With above probabilities, we have $\Pr[Y_i = +1] = \frac{n-t}{n}$ and $\Pr[Y_i = +1] = \frac{t}{n}$. We are interested in the probability for the sum of all random variables to be smaller than 0, i.e. the case when a majority of all appended values is Byzantine. Since the nodes in the algorithm wait for at least k coinflips to be appended, the sum of the coinflips converges to the normal distribution $\mathcal{N}\left(k \cdot \frac{n-2t}{n}, k - \left(k \cdot \frac{n-2t}{n}\right)^2\right)$. We can now compute the probability for the Byzantine nodes to reach a negative sum by appending negative values to the memory when given access:

$$\Pr\left[\sum_{i=1}^{k} Y_i < 0\right] < \Pr\left[\sum_{i=1}^{k} Y_i - \mu < \mu\right] < \exp\left(-\frac{\mu^2}{2\sigma^2}\right)$$

where $\exp\left(-\frac{\mu^2}{2\sigma^2}\right) = \exp\left(-\frac{k^2(\frac{n-2t}{n})^2}{2\cdot(k-k^2(\frac{n-2t}{n})^2)}\right) < \exp(\frac{1}{2} \cdot k \cdot (\frac{n-2t}{t})^2)$. Note that in the worst case, $\#corr - \#byz = n - 2t = \Omega(1)$, $k = \Omega(n\log(n))$ appends to the memory are needed in order to satisfy validity with high probability. If the difference however is equal to $\#corr - \#byz = \Omega(n)$, $k = \Omega(\log(n))$ appended values to the memory are sufficient to satisfy validity with high probability. □

5.2 Byzantine Agreement with Chains

In this section, we will review the results of Garay and Kiayias [9] and Ren [21]. We will show that Byzantine agreement on the chain can also be achieved for $t < n/2$ Byzantine nodes if the nodes use a randomized strategy in order to break ties. Algorithm 5 shows an example of such an implementation in the append memory. The idea of the algorithm is to let nodes append their values to the longest chain based on their view of the append memory. Since access to

Algorithm 5 Byzantine Agreement with Chains

```
Input: append memory $M$, input value val($v$)
 1: $M$.read()
 2: while there is no longest chain of length at least $k$ in the memory do
 3:     $M$.read()
 4:     upon granted access to the memory do
 5:         Let $C$ be the set of the last states in the longest chains of $M$
 6:         Choose $c \in C$ according to a tie-breaking rule
 7:         $M$.append($c$, val($v$))
 8:     end upon
 9: end while
10: Decide on the sign of the sum of the first $k$ appends in the longest chain
```

the memory is randomized, with a certain probability, there is a longest chain that consists of a majority of correct input values, such that the decision value satisfies validity. We will differentiate between two tie-breaking rules for the algorithm:

Deterministic tie-breaking: In this rule the correct nodes choose the first longest chain in the memory [9].

Randomized tie-breaking: In this rule, the correct nodes choose one of the longest chains uniformly at random [21].

Both these strategies were mentioned in [9], however, no analysis was given for the second rule. For the deterministic rule, the following upper bound on the number of Byzantine nodes holds:

Theorem 5.3. *Algorithm 5 with deterministic tie-breaking cannot solve weak Byzantine agreement for $t \geq n/3$ Byzantine nodes.*

The proof of this theorem is based on the following idea: Since the nodes choose the longest chain according to a deterministic rule, one can assume that all ties will be broken in favor of the adversary. Therefore, one can assume that every append to the memory from a Byzantine node will cause a fork in the chain, i.e. it will append its value to the same append as the last correct node, thus producing two longest chains. With this strategy, every second append to the longest chain will on average be Byzantine. If the Byzantine nodes form a majority, they can change the decision value of the correct nodes even if the validity condition is satisfied for $t \geq n/3$.

Next, we will consider the randomized rule for tie-breaking in Algorithm 5. In this case, the previous Byzantine strategy will not be successful, since the correct nodes will only include every second Byzantine append to the memory and the average ratio of Byzantine nodes in the longest chain will be 1/3.

In the next theorem, we provide a simple bound on the resilience of Byzantine agreement on the chain and show that it is dependent on the rate λ of the Poisson process. The theorem connects the resilience of Byzantine agreement and the access rate of the nodes:

Theorem 5.4. *Algorithm 5 with a randomized tie-breaking rule has a resilience of $\frac{t}{n} \leq \frac{1}{1+\lambda\cdot(n-t)}$. That is, for $\lambda \cdot (n-t) = 1$, the resilience is $\leq 1/2$ while for $\lambda \cdot (n-t) = 2$ it is $\leq 1/3$.*

Proof. For simplicity, we will restrict ourselves to an average analysis in this proof: The rate $\lambda \cdot (n-t)$ is a measure for how many appends from correct nodes to the memory will take place on average within the interval Δ. In the worst-case scenario, when the delay between any two operations of the correct nodes is δ, appends by correct nodes inside the same interval Δ will be concurrent and therefore generate a fork. Thus, all, but one such correct append can be considered wasted, as it will not be part of the longest chain.

Assume for contradiction that $\frac{t}{n} > \frac{1}{1+\lambda\cdot(n-t)}$. The Byzantine strategy that can be applied in this case is to play the role of a tie-breaker among the concurrent correct appends. This is possible due to the contradiction assumption - the Byzantine party will also have access to the memory in the same interval Δ. The Byzantine party can append its value simultaneously to the first correct append in the longest chain, and thereby prolong the chain by one additional append. Thus, all following correct appends from the same time interval will append their values to an "outdated" state of the memory, and therefore not make it into the longest chain. With this strategy, the longest chain of size k will have $k/2$ Byzantine values appended to it. Even if all correct nodes have the same input value, $k/2$ Byzantine values inside the longest chain of size k are enough to flip the decision value of the correct nodes. This would violate the validity condition. □

While the above analysis is simple, it is only derived for the average case. Note that, in order to derive a similar bound with high probability, intervals in which the correct nodes have strictly less than $\lambda \cdot (n-t)$ concurrent appends need to be estimated and compared to the number of intervals where the correct nodes have at least $\lambda \cdot (n-t)$ appends and the Byzantine party has also at least one append. Such an analysis has been conducted by Ren [21] who showed that Nakamoto consensus can achieve a resilience of almost 1/2 if the rate is much smaller than the delay, i.e. when $\lambda \ll \Delta$.

Unlike in the analysis of Ren [21], in Byzantine agreement, we use a fixed interval in order to decide on the agreement value. We would therefore need to take a closer look at what kind of strategies Byzantine nodes can apply just before the decision takes place. We will omit such an analysis for the chain at this point, since it is very similar to the analysis of the DAG that will be presented in the next section.

5.3 Byzantine Agreement with DAGs

Contrary to the chain, the DAG follows an inclusive strategy: The DAG is a directed acyclic graph that starts at some dummy append, e.g. at the empty state of the memory. All further values that are appended by the nodes only specify the latest seen appends to the memory. That is, if a node sees that another node has appended a value val(v) to the dummy value in the memory, it will list val(v) as its preceding value instead of the dummy append. Listing preceding appends can be viewed as drawing an arrow from the new append to all previous ones which do not have any incoming arrows yet. This strategy generates a directed acyclic graph. Note that the idea of the DAG is very similar to Algorithm 1, where all nodes refer to all values they read in the previous round. Since an implementation of rounds requires the nodes to always participate in the broadcast, DAG can be seen as a lighter version of it, where a round consists of parallel appends to the memory. The randomized access to the memory thereby bounds the number of appends from Byzantine parties in each round. Algorithm 6 presents a possible implementation of Byzantine agreement on the DAG.

Algorithm 6 Byzantine Agreement with DAG

```
Input: append memory $M$, input value $\texttt{val}(v)$
1: $M$.read()
2: while there is no longest (heaviest) containing at least $k$ values do
3:     $M$.read()
4:     upon granted access to the memory do
5:         Let $C$ be the set of the last states of $M$, which do not have child nodes
6:         $M$.append($C$, $\texttt{val}(v)$)
7:     end upon
8: end while
9: Order the values of the DAG with respect to the longest chain
10: Decide on the sign of the sum of the first $k$ values in the ordering
```

The correctness of Algorithm 6 is based on one of the tie-breaking rules in Line 2, such as the heavies chain defined in the Ghost protocol [22] or simply the longest chain [14]. In this section, we are interested in the impact of the worst-case construction of the DAG for Byzantine agreement. In the previous section, it was noted that the worst-case construction of a chain is reached by letting correct nodes generate forks and the Byzantine nodes break the ties. When the nodes are building a DAG, such a construction does not work here, as there will be correct nodes which will include all forked values into the ordering at a later point in time.

The analysis of the DAG therefore focuses on two main issues: The first issue is the rate at which the nodes are appending values. If the rate is too large, the nodes will likely not have the same views when appending to the memory and therefore it will not be possible to determine the global order of the appends. If the rate is small enough, [14, 22] show that w.h.p. there will be a longest chain such that the nodes can decide on an identical view and thus on the ordering of the values in the DAG. The second issue is that Byzantine nodes have the possibility to alter the algorithm by not referencing all values they see in the DAG. While the views of the correct nodes can be identical, a Byzantine strategy can increase the number of Byzantine values among the first k values that are considered for decision.

Lemma 5.5. *In Algorithm 6, the views of the DAG upon decision may contain up to $\Omega(\log(n))$ additional Byzantine values with high probability.*

Proof. Note that all Byzantine parties can be controlled by one single adversary, and that they can withhold their values for a small period of time when the correct nodes are not appending. If the Byzantine nodes apply the strategy for their values among the first k appends, this strategy will not change the ratio between the correct and the Byzantine nodes among these first writes. Instead, the Byzantine nodes can append a chain of values in the last interval of size Δ just before the decision, thus prolonging the longest(or heaviest) chain and adding their own values to the first k values of the DAG.

We can bound the length of the time interval T during which no correct node appends a value to the memory by the Poisson distribution as follows:

$$\Pr\left[T > \Delta \cdot \log(n)\right] = \exp\left(-\frac{\lambda(n-t)}{t\Delta} \cdot \Delta \cdot \log(n)\right) \leq \frac{1}{n^{\lambda/2}}$$

That is, with high probability there will be an append by a correct node at the end of the interval T.

Next, we calculate the length of the chain that the Byzantine nodes can produce within the time interval T. The size of this chain corresponds to the number of values that Byzantine nodes can insert into the sequence of the first k appends, in addition to the values that are included in the sequence due to the Byzantine rate. Let X denote the Poisson random variable with rate $\mu = \frac{\lambda t}{n}\log(n) \leq \frac{1}{2}\lambda\log(n)$, which corresponds to the Byzantine rate inside the time interval T. We can use the Poisson tail in order to bound the number of Byzantine writes during this time interval:

$$\Pr\left[X \geq \mu + \lambda^2\log(n)\right] \leq \exp\left(\frac{\lambda\log^2(n)}{\mu + \lambda\log(n)}\right) \leq \exp\left(\frac{2}{3}\log^2(n)\right)$$

The above equation states that with high probability, the Byzantine nodes will add less than $2\lambda\log(n)$ values to the memory within a time interval T. □

Theorem 5.6. *Algorithm 6 satisfies all-same-validity, termination and agreement with high probability.*

Proof. Termination and agreement of Algorithm 6 are guaranteed at Line 2 and from the fact that there will be a longest or heavies chain as was shown in [14, 22].

In order to show validity, we consider the same probability distribution as in the proof of Theorem 5.2. Due to Lemma 5.5, the amount of correct writes has to be at least $2\lambda\log(n)$ in order for the correct nodes to satisfy validity:

$$\Pr\Big[\sum_{i=1}^{k} Y_i < 2\lambda\log(n)\Big] = \Pr\Big[\sum_{i=1}^{k} Y_i - \mu < 2\lambda\log(n) - \mu\Big]$$
$$\leq \exp\big(-\big(\sqrt{k}\big(\frac{n-2t}{n}\big) - \frac{1}{\sqrt{2k}}\lambda\log(n)\big)^2\big)$$

We next analyse when the above probability becomes exponentially small, which would imply validity with high probability. In the worst case, for $\#corr - \#byz = \Omega(1)$, the number of appends in the memory has to be at least $k = \Omega(\lambda n\log(n))$. In the case $\#corr - \#byz = \Omega(n)$, $k = \Omega(\lambda\log(n))$ values are sufficient. Note that other than in Theorem 5.2, the number of values that are needed for a decision with DAG also depends on the rate λ, which follows from Lemma 5.5. □

The proof shows that the resilience of Byzantine agreement with DAG is independent of the rate λ. Moreover, this analysis shows that the DAG can tolerate up to $t < n/2$ Byzantine nodes, which corresponds to the optimal bound for Byzantine agreement.

Finally, we would like to emphasize that Nakamoto consensus, unlike Byzantine agreement, does not require finality. In [22], the authors mention that the resilience of Nakamoto consensus on the DAG does not change if the nodes are temporarily asynchronous. The above analysis, in particular Lemma 5.5, shows that this result is not true for Byzantine agreement. In Algorithm 6, there is a predetermined number of appends, on which the nodes base their decision. In the case of a temporal asynchrony, the Byzantine nodes could make sure to add more Byzantine values into the set of the first k appends. Therefore, temporarily asynchronous nodes would reduce the resilience of Byzantine agreement on the DAG.

REFERENCES

[1] Marcos Kawazoe Aguilera and Sam Toueg. 1999. A simple bivalency proof that t-resilient consensus requires t+1 rounds. *Inform. Process. Lett.* 71, 3 (1999), 155 – 158.

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

[3] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. 1995. Sharing Memory Robustly in Message-Passing Systems. *J. ACM* 42, 1 (January 1995), 124–142.

[4] P. J. Courtois, F. Heymans, and D. L. Parnas. 1971. Concurrent Control with "Readers" and "Writers". *Commun. ACM* 14, 10 (October 1971), 667–668.

[5] Edsger W. Dijkstra. 1965. Solution of a Problem in Concurrent Programming Control. *Commun. ACM* 8, 9 (1965).

[6] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. 1987. On the Minimal Synchronism Needed for Distributed Consensus. *J. ACM* 34, 1 (January 1987), 77–97.

[7] D. Dolev and H. R. Strong. 1983. Authenticated Algorithms for Byzantine Agreement. *SIAM J. Comput.* 12, 4 (1983), 656–666.

[8] Michael J. Fischer and Nancy A. Lynch. 1982. A lower bound for the time to assure interactive consistency. *Inform. Process. Lett.* 14, 4 (1982), 183 – 186.

[9] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. 2015. The Bitcoin Backbone Protocol: Analysis and Applications. In *Advances in Cryptology - EUROCRYPT 2015*. 281–310.

[10] Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovič, and Dragos-Adrian Seredinschi. 2019. The Consensus Number of a Cryptocurrency. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing (PODC '19)*. 307–316.

[11] Saurabh Gupta. 2016. A Non-Consensus Based Decentralized Financial Transaction Processing Model with Support for Efficient Auditing.

[12] Maurice Herlihy. 1991. Wait-Free Synchronization. *ACM Trans. Program. Lang. Syst.* 13, 1 (January 1991), 124–149.

[13] Lucianna Kiffer, Rajmohan Rajaraman, and abhi shelat. 2018. A Better Method to Analyze Blockchain Consistency. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 729–744.

[14] Chenxing Li, Peilun Li, Dong Zhou, Wei Xu, Fan Long, and Andrew Yao. 2018. Scaling Nakamoto Consensus to Thousands of Transactions per Second. arXiv:cs.DC/1805.03870

[15] Michael C Loui and Hosame H Abu-Amara. 1987. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing research* 4, 163-183 (1987), 31.

[16] Dahlia Malkhi, Michael Merritt, Michael Reiter, and Gadi Taubenfeld. 2000. Objects Shared by Byzantine Processes. In *Distributed Computing*. 345–359.

[17] Satoshi Nakamoto and A Bitcoin. 2008. A peer-to-peer electronic cash system. (2008).

[18] Rafael Pass, Lior Seeman, and Abhi Shelat. 2017. Analysis of the Blockchain Protocol in Asynchronous Networks. In *Advances in Cryptology – EUROCRYPT*. 643–673.

[19] R. Pass and E. Shi. 2017. Rethinking Large-Scale Consensus. In *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*. 115–129.

[20] S. A. Plotkin. 1989. Sticky Bits and Universality of Consensus. In *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing (PODC)*. 159 – 175.

[21] Ling Ren. 2019. Analysis of Nakamoto Consensus. *IACR Cryptology ePrint Archive* (2019), 943.

[22] Yonatan Sompolinsky and Aviv Zohar. 2015. Secure High-Rate Transaction Processing in Bitcoin. In *Financial Cryptography and Data Security*. 507–527.