Authenticated Consensus in Synchronous Systems with Mixed Faults

Ittai Abraham[1], Danny Dolev *[2], Alon Kagan *[2], and Gilad Stern *[2]

[1]VMware Research
[2]The Hebrew University of Jerusalem

June 18, 2022

Abstract

Protocols solving authenticated consensus in synchronous networks with Byzantine faults have been widely researched and known to exists if and only if $n > 2f$ for f Byzantine faults. Similarly, protocols solving authenticated consensus in partially synchronous networks are known to exist if $n > 3f + 2k$ for f Byzantine faults and k crash faults. In this work we fill a natural gap in our knowledge by presenting MixSync, an authenticated consensus protocol in synchronous networks resilient to f Byzantine faults and k crash faults if $n > 2f + k$. As a basic building block, we first define and then construct a publicly verifiable crusader agreement protocol with the same resilience. The protocol uses a simple double-send round to guarantee non-equivocation, a technique later used in the MixSync protocol. We then discuss how to construct a state machine replication protocol using these ideas, and how they can be used in general to make such protocols resilient to crash faults. Finally, we prove lower bounds showing that $n > 2f + k$ is optimally resilient for consensus and state machine replication protocols.

1 Introduction

In recent years there has been a surge of interest in Byzantine Fault Tolerance (BFT) and Blockchain technologies. The security of both Bitcoin and later Ethereum's proof-of-work protocols depends on a synchronous model and obtains resilience against minority corruptions [16, 24]. Following this direction there have been several academic papers that advanced *authenticated* BFT protocols and systems in the synchronous model that use more traditional membership assumptions [1, 2, 4, 5, 10, 18, 21, 23, 28]. A major advantage of this model is that it can obtain resilience as long as $n > 2f$, which is qualitatively much better than protocols that assume partial synchrony (or asynchrony) that can only obtain resilience of $n > 3f$.

In this paper we continue this line of research into authenticated BFT protocols and merge it with yet another long line of research around *mixed-faults* [8, 11, 12, 13, 14, 17, 19, 20, 26, 27, 29, 30, 31]. In the mixed-faults model that we study in this paper, the adversary can corrupt up to f parties in a malicious manner and can crash up to k additional parties. One motivation behind this assumption is that it allows us to model a real world case where the non-faulty parties can detect some of the corrupted parties (via some side broadcast channel, say the internet or a secure messaging application) and publicly mark them as faulty, hence simulate a crash. Another motivation is to model the case that there is some trusted hardware that may cause some of the parties to crash if they become compromised. Another motivation, for a large set of parties, is that some of the honest parties may be offline for large periods of time, and hence can be modeled as crashed. Finally, there may be a need to model crash failures (due to hardware failures) as a separate parameter from Byzantine failures (due to an adversary).

To the best of our knowledge, this problem of authenticated BFT in synchrony with mixed-faults has not been systematically studied. It is well known that the best one can hope for in partial synchrony

*This work was supported in part by the HUJI Federnann Cyber Security Research Center in conjunction with the Israel National Cyber Directorate (INCD) in the Prime Minister's Office.

(or asynchrony) is security when $n > 3f + 2k$. This is where authenticated BFT in synchrony gives a major resilience advantage. The main result of this paper is that it is possible to get security *if and only if* $n > 2f + k$. We note that we do not find this bound very surprising, but we believe getting to this bound and proving tight upper and lower bounds provides new insights into how to design authenticated BFT protocols in the synchronous model.

Our Contributions – Upper Bounds: The main contribution of our paper is MixSync, an authenticated consensus protocol in a synchronous network resilient to f Byzantine faults and k crash faults if $n > 2f + k$. MixSync uses a simple technique for achieving authenticated non-equivocation in synchronous networks with mixed-faults. As far as we know, this is the first authenticated consensus protocol in a mixed-faults setting achieving a resilience of $n > 2f + k$ without limiting the power of the adversary. This is made possible by solving the task in a synchronous setting, as opposed to the partially synchronous setting which requires at least $n > 3f + 2k$ replicas. The protocol is oblivious to the number of crash faults, so it is possible to use it as long as some bound is known on the number of Byzantine replicas and at least $f + 1$ honest replicas are guaranteed to stay online.

To construct our new authenticated consensus protocol we decompose it into an outer protocol and an inner building block. We call this inner building block *Publicly Verifiable Crusader Agreement* (PVCA). We show how to construct a simple and efficient $PVCA$ protocol in a network with mixed faults in Section 3. In this task, there is a commonly known sender with an input x, and replicas are required to output some value v and a proof π. If the sender is honest, every honest replica that completes the protocol outputs x and a proof π, showing that it is their actual output from the protocol. If the sender is faulty, then there exists some value v such that every honest replica either outputs v or $\perp$ with an appropriate proof. Using these proofs, replicas can convince each other that the value they received is correct, or alternatively that the sender was faulty by producing a proof for $\perp$. This task formalizes a rather strong notion of a non-equivocation round. By the end of the round, every replica that hears a message from the sender, hears the same message, in addition to proving that a given value was actually received from the sender (or that the sender was faulty). The $PVCA$ protocol relies on a simple technique: forwarding a received message to all replicas, and then sending a second message immediately after that. A replica receiving the second message knows that if its sender was non-Byzantine, the first message has already been sent to all replicas.

Using the simple idea of a double-send, we then construct the MixSync protocol in Section 4. The protocol is based on the Sync HotStuff protocol, with slight adaptations made for it to solve the task of single-shot consensus. Our protocol is view and leader based, and just like Sync HotStuff, each view consists of a view change and a non-equivocation round. In order to make our protocol resilient to k crash faults, all that is needed is using the double-send technique from our $PVCA$ protocol. In fact, we could use the $PVCA$ protocol as a blackbox inside the Sync HotStuff protocol, only requiring the addition of a view-change protocol, but we open the blackbox in order to optimize the protocol. Finally, we discuss how to create a State Machine Replication (SMR) protocol using the same double-send idea. This can either be done generically by using our consensus protocol, or by adapting optimized protocols such the Sync HotStuff protocol. Thankfully, in the Sync HotStuff protocol replicas already send two messages to all replicas after receiving a value from the leader, meaning that the only change required is making sure that replicas send them in a specific order.

These constructions suggest a possible general approach to constructing consensus and SMR protocols in the authenticated synchronous mixed-fault scenario. If a protocol mainly consists of a non-equivocation round and a view change protocol, replace the non-equivocation round with our $PVCA$ or simply a double-send, yielding a crash-resilient protocol. Specific protocols might also require adapting other parts of the protocol if they rely on specific properties of the non-equivocation round.

Our Contributions – Lower Bounds: In order to complete the picture, we also provide tight lower bounds for consensus tasks in the presence of mixed-faults in the synchronous model. First, we show that consensus is impossible in a system with f Byzantine faults and k crash faults if $n \leq 2f + k$. Secondly, in order to prove a similar lower bound for the task of SMR, we first formalize the notion of a Write-Once Register. A Write-Once Register is a shared memory object to which clients can write only once. That means that once a client manages to write a value to the register, this value is final and cannot be changed. Unlike registers that allow clients to overwrite previous values, the Write-Once Register captures a specific idea of finality that is shared with consensus protocols. In SMR protocols, replicas are required to commit to a value v_s for each log position $s \in \mathbb{N}$, which clients can read by asking replicas to send their committed values. This means that SMR protocols actually implement infinitely

many Write-Once Registers. As such, we show that no protocol virtualizing a Write-Once Register exists in a system with f Byzantine faults and k crash faults if $n \leq 2f + k$, yielding a lower bound on SMR protocols as well.

Related Work: Consensus and State Machine Replication have been widely researched in networks with mixed faults. For example, classic results such as that of Siu, Chin and Yang [29] achieve consensus in synchronous networks in the face of f Byzantine faults and k "dormant" faults, consisting of omissions or slow delivery, if $n > 3f + k$. The works of Thambidurai and Park [30] and of Lincoln and Rushby [20] achieve broadcast in synchronous networks in the face of f Byzantine faults, s "symmetric" Byzantine faults, and k non-malicious faults if $n > 2f + 2s + k$. In the above, symmetric faults are ones in which the adversary chooses to send the same messages to all replicas, and non-malicious faults are faults such as omission, crash, and timing faults.

Another line of work focuses on such tasks in partially synchronous networks. For example, the Scrooge protocol by Serafini, Bokor, Dobre, Majuntke and Suri [26] is resilient to u total faults (crash or Byzantine), r of which are Byzantine if $n \geq 2u + 2r$. Setting $r = f$ and $u = f + k$ we get $n \geq 4f + 2k$. Similarly, the Upright protocol of Clement, Kapritsos, Lee, Wang, Alvisi, Dahlin and Riche [12] is resilient to u total faults and r Byzantine faults if $n \geq 2u + r + 1$. Again, setting $r = f$ and $u = f + k$, we get $n \geq 3f + 2k + 1$ faults, but their work allows for omission faults as well. The SBFT protocol of Gueta, Abraham, Grossman, Malkhi, Pinkas, Reiter, Seredinschi, Tamir and Tomescu, [17] also achieves resilience to f Byzantine faults and k crash faults if $n \geq 3f + 2k + 1$.

A different approach to increasing resilience is limiting the power of adversary. For example, the work of Correia, Lung, Neves and Verissimo [13] achieves broadcast for any number of Byzantine faults, assuming special hardware that reduces all faults to crash faults in critical sections. In this sense, the mixed faults don't manifest in both types of faults existing in the network *at the same time*, but different faults are allowed at different times. Following that, Correia, Neves and Verissimo [14] show how to achieve state machine replication with $n \geq 2f + 1$ by using similar hardware. Similar approaches are used in A2M by Chun, Maniatis, Shenker and Kubiatowicz [11], in TrInc by Levin, Douceur, Lorch and Moscibroda, [19], and in the work of Veronese, Correia, Bessani, Lung and Verissimo [31], increasing resilience by using specialised trusted hardware to limit the adversary. Other approaches can be found in the work of HeterTrust by Serafini and Suri [27] and BFT-SMART by Bessani, Sousa and Alchieri [8], implementing state machine replication in partial synchrony. The HeterTrust protocol separates nodes to two different roles, execution and coordination nodes. The protocol is then resilient to f Byzantine faults and k crash faults if $n \geq 2f + 2k + 2$, but assumes that coordination nodes can only crash and cannot be Byzantine. The BFT-SMART protocol on the other hand can be configured mid run to tolerate either f Byzantine faults or k crash nodes assuming that both $n \geq 3f + 1$ and $n \geq 2k + 1$. However, the protocol cannot tolerate both types of faults at the same time. Some of the works above are designed to be safe in asynchrony and live in synchrony, but do not explicitly refer to themselves as protocols for partially synchronous systems. For simplicity we call these works resilient in partial synchrony as well.

2 Model and Definitions

2.1 Communication and Corruption Model

Entities in the System: We consider a fully connected network with n entities called "replicas", for a commonly known n. Replicas can send messages to each other, which arrive in a FIFO order. This means that two messages sent over the same link keep their order. This can be implemented by adding sequence numbers to each message. In addition, we say a replica "multicasts" a message if it sends to all replicas. When dealing with state machine replication, we also consider a network with c clients. The clients cannot communicate with each other, but have links to all n replicas in the system.

Synchronous communication network: We assume a synchronous model, i.e. all replicas can exchange messages over reliable communication links, ensuring messages will always arrive at most after a known bounded time Δ to its destination. The value Δ is defined to also be long enough for any underlying clock-synchronization protocol to take place if required, and to take into account possible drift within the bounds allowed by the synchronization protocol.

Authentication and keys: In our model, we assume the existence of a public key infrastructure. Each replica has a signing key sk_i and knows the public key pk_j of every replica j. Replicas can compute

a signature σ for the message m using the $sign(sk_i, m)$ function, and can verify other replicas' messages using the $verify(pk_j, m, \sigma)$ function, which is defined to be True iff $sign(sk_j, m) == \sigma$. It is well known that such a function exists without needing to know sk_j. In this work, we assume a perfectly secure signature scheme, guaranteeing that the message was sent by the appropriate replica, as no signature could be forged (even by a Byzantine replica). Replicas can also forward messages with signatures from other replicas. A message m sent signed by replica i is noted as $\langle m \rangle_i = (m, \sigma, i)$ for $\sigma = sign(m, sk_i)$. In that case $verify(\langle m \rangle_i)$ is defined as $verify(pk_i, m, \sigma)$.

Adversary: This work deals with a static adversary that can choose replicas to corrupt in the beginning of the protocol. The adversary also controls the exact amount of time a message is delayed, and can choose any delay between 0 and Δ time for any message to arrive. When corrupting a replica, the adversary can choose one of the following types of faults:

- **Crash fault**. A crash faulty replica is a replica that the adversary can crash - meaning no more messages would be sent or received by the replica from that point on. In each round replicas may choose to send messages $m_1, \ldots, m_\ell$, with m_1 being the first message sent and m_ℓ being the last. When the adversary crashes a replica, it chooses an i such that the messages $m_1, \ldots, m_i$ are delivered but $m_{i+1}, \ldots, m_\ell$ are not (as well as any message in future rounds).

  The adversary can chose when to crash a crash faulty replica throughout the protocol or to not crash it at all.

- **Omission fault**. The adversary can choose any arbitrary message sent to or from the omission-faulty replicas or clients, and stop it from reaching its destination.

- **Byzantine fault**. A Byzantine faulty replica is a replica the adversary has complete control over. This means that the adversary can make it deviate arbitrarily from the protocol.

In this work, we say that a replica is **honest** if it is not crash, omission, or Byzantine faulty. On the other hand, we say that a replica is **non-Byzantine** if it is either honest, crash, or omission faulty. Note that we only consider omission faulty clients. The adversary cannot choose to make a client Byzantine, but can essentially make it "crash" from the viewpoint of the system by omitting all of its ingoing and outgoing messages.

In this work, k is the number of crash faulty replicas, f is the number of Byzantine faulty replicas, and t is the number of omission faulty replicas. Note that crash and omission faulty replicas act honestly based on the messages that they see, and they do not know that they are faulty, or which other replicas are faulty.

2.2 Definitions

We start by defining the task of Crusader Agreement, as formulated in [15]. In this task, a known sender has a value to send to all replicas. Every honest replica then outputs some value v or a special value $\perp$, indicating that the sender was faulty. Furthermore, if the sender is honest, every replica outputs its input. In a protocol solving Crusader Agreement, every non-Byzantine replica that outputs a non-$\perp$ value must output the same value. A stronger related task is Publicly Verifiable Crusader Agreement. In this setting, all replicas have access to an external validity oracle $validate()$ that takes non-$\perp$ values and outputs either True or False. In addition, replicas output proofs, showing that they output a correct value from the protocol. The values and proofs can be checked using a $check()$ function, defined as part of the protocol. It is assumed that the sender has an externally valid input, and non-Byzantine replicas must output externally valid values. This notion of an external validity function was suggested by Cachin, Kursawe, Petzold and Shoup [9], and used extensively in consensus protocols, for example in [3, 6, 9, 22]. More precisely, the task is defined as follows:

Definition 2.1. *Let I be the set of all possible inputs to the protocol and let $\perp \notin I$. In addition, let there be some proof space P. All replicas have access to an oracle function $validate : I \to \{True, False\}$. A Publicly Verifiable Crusader Agreement scheme consists of a protocol $PVCA$ and a function $check : (I \cup \{\perp\}) \times P \to \{True, False\}$. $PVCA$ has a designated sender s with some input $x \in I$ such that $validate(x) = True$. Every replica then outputs a pair $(v, \pi) \in (I \cup \{\perp\}) \times P$ such that v is a value and π is a proof. A scheme solving Publicly Verifiable Crusader Agreement has the following properties:*

- ***Correctness.*** *If two non-Byzantine replicas i, j output non-$\perp$ values v_i, v_j with proofs, then $v_i = v_j$.*

- ***Validity.*** *If s is honest with input x, every non-Byzantine replica that completes the protocol outputs x, and a proof; in addition, no replica can produce a pair v, π such that $v \neq x$ and $check(v, \pi) = True$ (including for $v = \perp$). Furthermore if s is non-Byzantine (but is possibly crash-faulty), and some replica produces a pair v, π such that $v \neq \perp$ and $check(v, \pi) = True$, then $v = x$.*

- ***Termination.*** *Every honest replica outputs some pair v, π and completes the protocol.*

- ***External Validity.*** *If some non-Byzantine replica outputs the pair v, π, then either $validate(v) = True$ or $v = \perp$.*

- ***Verifiability.*** *If a non-Byzantine replica outputs v, π, then $check(v, \pi) = True$. Furthermore, if some non-Byzantine replica outputs v, π such that $v \neq \perp$, then it is impossible for any replica to produce a pair v', π' such that $v' \notin \{v, \perp\}$ and $check(v', \pi') = True$.*

A Crusader Agreement protocol, described by the first three properties of the definition above (ignoring the proofs), formalizes the idea of a non-equivocation round. In such a round, a sender can choose a single message to send to whichever replicas it wants, but it cannot send different messages to different replicas. The stronger Publicly Verifiable Crusader Agreement protocol also allows replicas to prove that their output is a correct output from the protocol. Importantly, if the sender is non-Byzantine, Byzantine replicas can only prove that their output was either the sender's input x or $\perp$. Furthermore, if the sender is honest, the only value for which a proof can be produced is x. This means that replicas can prove to each other that a given sender is faulty, without worrying that a Byzantine replica will be able to convince them that an honest sender was actually faulty. Note that Verifiability implies Correctness, but the Correctness property is included for clarity.

Using ideas from of the Publicly Verifiable Crusader Agreement protocol, we construct consensus and state machine replication protocols.

Definition 2.2. *In a consensus protocol, each replica i has an input x_i, and needs to output some value v_i. A protocol solving consensus has the following properties:*

- ***Correctness.*** *If two honest replicas i, j output the values v_i, v_j respectively, then $v_i = v_j$.*

- ***Validity.*** *If all honest replicas have the same input x, then all honest replicas output x.*

- ***Termination.*** *All honest replicas complete the protocol and output some value.*

It is also possible to formulate the task of Validated Consensus, in which all replicas have externally valid inputs. Then, a Validated Consensus protocol has the following additional external validity property:

Definition 2.3. ***External Validity.*** *If some non-Byzantine replica outputs the value v, then necessarily $validate(v) = True$.*

The task of State Machine Replication (SMR), is highly related to that of consensus. In this task, c clients are required to agree on a log of values, sometimes thought of as actions in a state machine. More precisely, for every "log position" $s \in \mathbb{N}$, they must agree on a value v_s. Usually there is also an additional requirement that the values are agreed upon in the correct order, i.e. first v_1 is agreed upon, then v_2, etc. For example, one formulation of an SMR protocol, as presented by Abraham *et al.* [2] is as follows:

Definition 2.4. *A protocol solving State Machine Replication has the following properties:*

- ***Safety.*** *Non-Byzantine replicas do not commit different values at the same log position.*

- ***Liveness.*** *Every honest client's request is eventually committed by all honest replicas.*

Clients can then ask replicas to inform them of the committed value for each log position, and consider a value val as committed if they receive $f + 1$ responses with the value val.

3 Publicly Verifiable Crusader Agreement

This section describes $PVCA$, a Publicly Verifiable Crusader Agreement protocol, resilient to f Byzantine faults and k crash faults for $n > 2f + k$. This protocol is later used as a building block, conceptually playing the role of a non-equivocation round. A simplified version of the $PVCA$ protocol is presented, assuming all replicas start it at the same time. A more general version of the protocol is used in the MixSync consensus protocol of Section 4. In this version, different replicas might start the protocol T time apart. If that is the case, every replica must wait for T time in the beginning of the protocol, listening to messages from other replicas. As can be seen in the proof provided in Section 4, the $PVCA$ protocol works in this case as well.

Protocol $PVCA$:

- **Round 1:** The sender sends a $\langle \mathsf{propose}, value \rangle_s$ message to all replicas, with $value$ being its input to the protocol.

  If a replica r did not get its first propose message signed by the leader by time $t = \Delta$, send a $\langle \mathsf{blame} \rangle_r$ message to all replicas.

- **Round 2:** Upon receiving blame messages $b_1, \dots, b_{f+1}$ from $f+1$ different replicas, or a $\mathsf{blame-cert}$ message containing blame messages $b_1, \dots, b_{f+1}$ from $f + 1$ different replicas, send the message $\langle \mathsf{blame-cert}, \{b_1, \dots, b_{f+1}\} \rangle_{self}$ to all replicas and proceed to output calculation. Alternatively, upon receiving a $\langle \mathsf{propose}, value \rangle_s$ message signed by the leader, if no propose message has been forwarded with the same value previously, send the message to all replicas. Then, if $validate(value) = True$ and no propose message has been received signed by the sender with a different value, send a $\langle \mathsf{vote}, value \rangle_{self}$ message to all replicas, start a timer of 2Δ time and continue handling messages as described above during that time. If either of those conditions don't hold, proceed to output calculation immediately.

- **Output Calculation** After the 2Δ time, or proceeding to this step from Round 2, do the following:
  - If a $p_1 = \langle \mathsf{propose}, value \rangle_s$ message was received such that $validate(value) = False$, output $(\bot, \{p_1\})$.
  - If at any point two propose messages $p_1 = \langle \mathsf{propose}, value \rangle_s, p_2 = \langle \mathsf{propose}, value' \rangle_s$ were received such that $value \neq value'$, output $(\bot, \{p_1, p_2\})$.
  - If at any point blame messages $b_1, \dots, b_{f+1}$ were received from $f + 1$ different replicas or a $\mathsf{blame-cert}$ message is received with blame messages $b_1, \dots, b_{f+1}$ signed by $f + 1$ different replicas, multicast $\langle \mathsf{blame-cert}, \{b_1, \dots, b_{f+1}\} \rangle_{self}$, output $(\bot, \{b_1, ..., b_{f+1}\})$.
  - Otherwise, if a $\langle \mathsf{propose}, value \rangle_s$ message was received, output $(value, votes)$, with $votes$ being all signed $\langle \mathsf{vote}, value \rangle$ messages received from replicas.

  When outputting a value complete the protocol and terminate.

Given inputs v, π, $check(v, \pi)$ acts as follows:

- If $v = \bot$, and π contains a single message $p_1 = \langle \mathsf{propose}, value \rangle_s$ signed by the sender such that $validate(value) = False$, output $True$.
- If $v = \bot$, and π contains two messages $p_1 = \langle \mathsf{propose}, value \rangle_s, p_2 = \langle \mathsf{propose}, value' \rangle_s$ signed by the sender such that $value \neq value'$, output $True$.
- If $v = \bot$, and π contains blame messages $b_1, \dots, b_{f+1}$ signed by $f + 1$ different replicas such that $value \neq value'$, output $True$.
- If $v \neq \bot$, and π contains $\langle \mathsf{vote}, v \rangle$ messages with the same value v signed by at least $f + 1$ replicas, output $True$.
- In all other cases, output $False$.

The following theorem is proven in Appendix A:

Theorem 3.1. *The pair $(PVCA, check)$ as described above are a Publicly Verifiable Crusader Agreement scheme for $n > 2f + k$.*

4 Uniform Consensus with Mixed Faults

The MixSync protocol presented in this section is a leader-based protocol for agreeing on a single value. MixSync is an adaptation of the Sync HotStuff [2] protocol to the task of single-shot consensus. In order to make the protocol resilient to crash faults as well, replicas send a second message indicating that information was spread to all replicas. This is the exact same mechanism used in the $PVCA$ protocol, in which replicas send propose messages, immediately followed by a vote message. A replica receiving the verification (vote) message can be sure that if it was sent by a non-Byzantine replica, the preceding proposal was forwarded to all replicas. That is because the sending replica did not crash before sending the verification message. Thus, it is enough to collect $f+1$ verification messages to be sure equivocation was detected, even if some of those verification messages were sent by crash faulty replicas.

As is standard for leader-based protocols, the protocol presented in Algorithm 1 actually achieves a weaker validity property, as formulated in the following definition:

Definition 4.1. ***Weak Validity.*** *If all replicas are non-Byzantine, all replicas output a value which is an input of some replica.*

On the other hand, the protocol achieves a stronger correctness property: uniform correctness.

Definition 4.2. ***Uniform Correctness.*** *If two non-Byzantine replicas i, j output the values v_i, v_j respectively, then $v_i = v_j$.*

Note that in a protocol achieving the regular correctness property, a crashed replica might have output any value before crashing. However, in a protocol achieving uniform consensus, if a crashed replica outputs a value before crashing, it must output the same value as all other non-Byzantine replicas. Finally, for simplicity, in this protocol replicas eventually output values by "committing to a value", but do not actually terminate. This is captured in the definition of the following property:

Definition 4.3. ***Eventual Commitment.*** *Every honest replica eventually outputs a value.*

Using standard transformations, it is possible to turn a protocol with eventual commitment to a terminating protocol. For example, this can be done by replicas sending $\langle \mathsf{commit}, val \rangle$ messages, indicating that they committed to val. Then, after receiving $f+1$ such messages, they forward all of them in one message and terminate. Combining these three properties, as well as the External Validity property defined in Section 2.2, in this section we will prove the following theorem:

Theorem 4.4. *There exists a protocol achieving Uniform Correctness, Weak Validity, External Validity and Eventual Commitment if $n > k + 2f$.*

4.1 Basic Terms

MixSync, described in Algorithm 1, is a leader based protocol which proceeds in "views". Each view has a set leader which is responsible for leading the whole system to consensus. If a failure was detected, the protocol will advance to change a view, setting a new leader. There are two types of failures. Crash faulty leaders might not send messages, and thus stall, requiring replicas to change a view later. On the other hand, Byzantine leaders can equivocate by sending different values to different replicas, in addition to stalling by not sending messages. Each replica is constantly listening for messages, and for every incoming message it starts a thread that handles the message. We assume network delay is much larger than computing delay, and so we practically assume running each command takes 0 time.

There also exists a $sleep(t)$ command. The sleep command pauses the main process from running for t time, and also pauses the message handling mechanism. Any message that arrives while the replica is sleeping is not handled, but enters a queue. After t time, all messages from the queue are handled by order of arrival, and only then the process proceeds. A $wait(t)$ command exists as well but does not freeze message handling, only the thread that called the $wait(t)$ command.

In the protocol, replicas construct certificates and proofs. A certificate is considered valid if it contains vote messages from $f+1$ different replicas, from the same view V with the same value val. If that is the case, we say that the certificate certifies val. In the proof of the protocol, we will show how certificates are used to guarantee that when parties commit to a value no other value may be proposed in future views. A proof P is considered valid if contains certificate messages sent by $f+1$ replicas in the same view, containing valid certificates with respect to some value and view.

4.2 Protocol Description

As mentioned earlier, the protocol consists of views. Each view has a determined leader known to all replicas. The leader is chosen in a round robin manner. For example, this can be done by setting the leader of view V to be replica $(V \mod n) + 1$. Replicas ignore any message received from views other than the one they are currently in, except for vote messages, as described in the protocol. Lastly, for every time we say "a replica forwards the message", it does so only if it hasn't forwarded the same message previously. This is to avoid loops of of forwarding messages back and forth.

Let V be the current view number and replica L the leader of current view. Each view proceeds in 5 conceptual rounds, as follows.

Protocol MixSync:

- **Round 0**: When starting a view, replicas increment their *view* variable and sleep for Δ time in order to collect vote messages and to allow for other replicas to join the view. While sleeping, accept vote messages from the previous view, and all messages from current view. After sleeping for Δ time, first handle vote messages from the previous view. If at least $f+1$ vote messages were received with the same value in the previous view, construct a certificate from them. Afterwards, send a certificate message as described in the next round, then handle all other messages by order of arrival. **Do not ignore vote messages from last view.**

- **Round 1**: After sleeping for Δ time, replicas send a $\langle \mathsf{certificate}, V, cert \rangle$ message to the leader, with *cert* being their most recently collected valid certificate. Certificates are explained more fully in round 4. In rounds 2 and 3, when mentioning that replicas wait some time we mean that they start waiting after sending the certificate message.

- **Round 2**: The leader L sleeps for an additional 2Δ time, guaranteeing that all certificate messages arrive. It then constructs a proof P from all valid certificates it received, and chooses the certificate and the value it certifies, *cert* and *value* respectively, from the most recent view. If all certificates received are empty certificates, i.e. no valid certificate has been built yet by any replica, the leader sets the proposed *value* to be its input and *cert* to be $\perp$.

  Finally the leader sends a $\langle \mathsf{propose}, V, value, cert, P \rangle_L$ message to all replicas.

- **Round 3**: All replicas wait for 4Δ time. If they receive a $\langle \mathsf{propose}, V, value, cert, P \rangle_L$ message during that time, they start by forwarding the message by multicasting it to all replicas, if they haven't forwarded the same propose message previously. They then check for leader faults by making sure that the following conditions hold:

  - The proof P contains at least $f+1$ valid certificates.
  - The proposed *value* is the one contained in the most recent certificate in P, if at least one of the certificates isn't empty. If all of the certificates are empty, check that $validate(value) = True$.
  - No $\langle \mathsf{propose}, V, value', cert', P' \rangle_L$ message was received previously in this view such that $(value', cert', P') \neq (value, cert, P)$, i.e. the leader did not equivocate.

  If all of those conditions hold, multicast $\langle \mathsf{vote}, V, value \rangle$, and start a timer of 2Δ. Note that a vote message is sent after forwarding the propose message to all replicas. If any of these conditions don't hold, start Round 0 (implemented by calling the view_change method).

- **Round 4**: If a replica received a $\langle \mathsf{propose}, V, value, cert, P \rangle_L$ and it did not change view while waiting, it commits to *value*.

Additionally, replicas check in the background that the leader is not stalling. If a replica enters a view and does not receive any propose message in 4Δ time, it sends $\langle \mathsf{blame}, view \rangle_r$ to all replicas. If a replica receives blame messages $b_1, ..., b_{f+1}$ from $f+1$ replicas from the current view, it forwards them all at once in a $\langle \mathsf{blame-cert}, \{b_1, ..., b_{f+1}\} \rangle$ message and calls view_change, proceeding to Round 0. When receiving a blame − cert message with $f+1$ blame messages from current view, forward the message by multicasting it and then change view.

The exact pseudo-code describing the MixSync protocol is provided in Algorithm 1 in Appendix C. Note that the entire $PVCA$ protocol actually takes place in rounds 2 and 3 of the MixSync protocol

described above. The $PVCA$ protocol could be used as a black-box by waiting before changing view, processing vote messages and then proceeding with the next view (this is similar to how it is currently written, but sets the time in which replicas move to the next view as after waiting, not before). However, the MixSync is presented in a white-box manner in order to show the similarity to the Sync HotStuff protocol and to show that replicas actually implicitly receive proofs of faulty leaders from each other in the $PVCA$ protocol without the need to forward such proofs again. We now proceed to prove that the protocol indeed solves consensus.

4.3 Proof of the Protocol

Note that the MixSync protocol described above consists of two conceptual parts. Rounds 2 and 3, are simply a non equivocation round nearly identical to the one described in the $PVCA$ protocol. Slight additions are made to these rounds, in order to make sure the leader sends proposals that are consistent with previous views. The rest of the protocol consist of a blaming mechanism used to proceed through views, and a view-change protocol in rounds 0 and 1. Even when proceeding through views, replicas either explicitly forward a proof that the leader was faulty using a blame − cert message, or implicitly do so by forwarding messages containing equivocation, invalid values, or invalid proofs. These are the exact proofs produced by the $PVCA$ protocol for a replica outputting $\perp$, when thinking of the validity of proofs as part of the external validity of a propose message.

Seeing as the protocol uses ideas from the $PVCA$ protocol, large parts of the proofs of the two protocols are similar. Both proofs are provided for completeness. First we prove some helpful Lemmas. Lemma 4.5 shows that all non-Byzantine replicas are "synchronized in views", with no more than Δ delay between replicas joining a given view. Lemmas 4.6 and 4.7 are used to show that if a replica commits to a value, then no other value can be committed by any non-Byzantine replica. Finally, Lemma 4.8 shows that replicas proceed in views until they manage to commit to a value, which guarantees that they eventually reach a view with an honest leader. In all proofs in this section assume that there are at most f Byzantine faults and k crash faults with a total of $n > 2f + k$ replicas.

In Lemma 4.5, we use the fact that replicas wait for Δ time in the beginning of each view in order to "let other replicas catch up" and enter their view. This is done in order to make sure that they don't ignore messages intended to make them advance to the next view. Then, if some non-Byzantine replica advances to a given view, every non-Byzantine replica that hasn't crashed yet does so too.

Lemma 4.5. *Once a non-Byzantine replica in view $V-1$ enters view V, all non-Byzantine replicas that haven't crashed Δ time later will enter view V by that time.*

Proof. We will prove the lemma by induction on the view number V.

All non-Byzantine replicas enter view 1 at the same time, meaning that the lemma holds for $V = 1$. Assume the claim holds for view $V \geq 1$. Let i be a non-Byzantine replica that entered view $V+1$ at time t_{V+1}. Before doing so, it had entered view V at some time t_V. Before handling any message, i sleeps in view V for Δ time, and doesn't send any messages while it sleeps. By the induction assumption all other non-Byzantine replicas which haven't crashed would be in view V by time $t_V + \Delta$, in which i starts sending messages in view V.

Replica i would only leave the view if it either hears $f+1$ blame messages or a blame − cert message; if it receives a propose message such that the proof is not valid or not compatible with the value proposed; or if it heard two different propose messages in that view. Before leaving view V, replica i forwards all blame − cert and propose messages and sends a blame − cert message containing $f+1$ blame messages if it receives them. All other replicas receive those messages at time $t_V + \Delta$ or earlier, and any non-Byzantine replica still in view V then sees that the same conditions hold and start view $V+1$, if it hasn't done so earlier. □

Lemma 4.6 is proven by showing that before committing to a value, a non-Byzantine replica waits in order to guarantee that the first proposal received by any non-Byzantine replica contains the same value. This means that non-Byzantine replicas only send votes for that value, from which the rest of the proof follows.

Lemma 4.6. *If a non-Byzantine replica i commits a value val_i at view V, then (i) any message containing a valid certificate from view V certifies val_i, and (ii) every non-Byzantine replica has a valid certificate for it from view V by the time i commits, if it hasn't crashed by that time.*

Proof. Observe some non-Byzantine replica i that commits a value val_i in view V at time t.

In order to show (i), we will actually show that no non-Byzantine replica sent a vote message containing any $val' \neq val_i$ in this view. Any valid certificate certifying val' must contain a signed vote message with the value val' from $f+1$ replicas. At least one of those replicas must be non-Byzantine, and thus showing the above proves that it is impossible to construct a valid certificate for val' from view V.

Since i committed at time t, it sent a propose and a vote message containing it at time $t-2\Delta$. We know that i does not send any message in view V before it sleeps for Δ time, and thus it entered the view by time $t-3\Delta$. Let j be another non-Byzantine replica. By Lemma 4.5, j entered the view by time $t-2\Delta$, and thus it received the propose message by time $t-\Delta$, if it hasn't crashed earlier. Since i committed at time t, it did not hear any equivocating propose message from j by that time. This means that j did not send a propose message or a vote message for any value $val' \neq val_i$ before time $t-\Delta$. As noted above, j received the propose message sent by i by time $t-\Delta$, and thus it won't send a vote message for $val' \neq val_i$ at time $t-\Delta$ or later, because equivocation would be detected.

We now turn to show (ii). No non-Byzantine replica j left view V before time $t-\Delta$, otherwise by Lemma 4.5 i would have left view V by time t, contradicting the assumption that it commits at time t in view V. Therefore, every honest replica j receives i's propose while still in view V and sends a vote message containing val_i by time $t-\Delta$. Every non-Byzantine replica receives those vote messages from every honest replica by time t, if it hasn't crashed earlier, and constructs a certificate from them. □

Lemma 4.7 is proven using the certificates that replicas collect. If some non-Byzantine replica commits to some value, then from that point on, all replicas will have recent certificates for that value. In addition, only earlier certificates might exist for other values. These two facts are then combined to show that any accepted proposal from the leader must be for the committed value.

Lemma 4.7. *Let i be the first non-Byzantine replica to commit a value val_i at view V. Then any* vote *message sent by a non-Byzantine replica votes for the value val_i in any view $V' \geq V$.*

Proof. We will now prove by induction that the following three statements are true for any view $V' \geq V$: (i) every non-Byzantine replica's certificate message in view $V'+1$ contains a certificate from a view V'' such that $V' \geq V'' \geq V$ which certifies the value val_i, (ii) any certificate message containing a valid certificate from view V' certifies val_i, (iii) any vote message sent by a non-Byzantine replica in view V' votes for the value val_i. Note that statement (i) refers to $V'+1$ and statement (ii) and (iii) refers to V'.

Start by observing view $V'=V$, and assume i committed in view V at time t. Under close inspection, the proof of Lemma 4.6 shows that (iii) holds for $V'=V$, and that no non-Byzantine replica $j \neq i$ leaves view V before time $t-\Delta$. Since replicas wait Δ wait when entering a new view, non-Byzantine replicas send certificate message in view $V+1$ at time t or later. By Lemma 4.6, every non-Byzantine replica constructs a certificate for val_i from view V by time t, and thus their certificate message contains a certificate from view V certifying val_i. In addition, from Lemma 4.6, no message contains a valid certificate for any other value in view V.

Observe some view $V' > V$, and assume the claim holds for $V'-1$. By assumption, every non-Byzantine replica's certificate message in view V' contains a certificate certifying val_i from some view $V'-1 \geq V'' \geq V$. In addition, by our assumption a propose message sent by the dealer can't contain a certificate for any other value from any view $V'-1 \geq V'' \geq V$. Before sending a $\langle \mathsf{vote}, V', val' \rangle_r$ message, non-Byzantine replicas must receive a $\langle \mathsf{propose}, V', val', cert, P \rangle_L$ such that P contains certificates from at least $f+1$ replicas with val' being the value from the most recent certificate. Such a proof contains a certificate from at least one non-Byzantine party. Therefore, the most recent certificate in the proof must be from at least view V, and as stated above it must certify val_i. Combining these observations, if some non-Byzantine replica sends a $\langle \mathsf{vote}, V', val' \rangle_r$ message, then $val' = val_i$, and we get (iii). As a certificate consists of at least one vote from a non-Byzantine replica, we get that any certificate from view V' must certify val_i. Hence we get that no message contains a certificate from view V' certifying any $val' \neq val_i$, proving that (ii) is true. In addition, if a non-Byzantine replica updates its most recent certificate in view V', it updates it to a certificate such that $V' \geq V$ certifying val_i, and thus (i) remains true as well. □

Lemma 4.8 is proven by showing that if an honest replica does not commit in a view, it must have either seen a fault, or heard nothing from the leader. If it hears nothing from the leader without committing, then other replicas must have also heard nothing, allowing all replicas to advance to the next view by constructing a $\mathsf{blame-cert}$.

Lemma 4.8. *If an honest replica does not commit in a view, it will eventually advance to the next view.*

Proof. Let i be a replica that arrives at view V at time t and does not commit in V. Assume by way of contradiction that i does not leave V. From Lemma 4.5, every other honest replica reaches view V by time $t + \Delta$. By assumption, i never reaches view $V + 1$ or commits. There must be some honest replica j that did not send a blame message in view $V + 1$, because otherwise i would have received $f + 1$ such messages and started view $V + 1$. It is important to note that if an honest replica sent a blame message, it must have done so after waiting for Δ time in the beginning of the protocol. From Lemma 4.5, i enters view V by that time, and receives any such blame message. In addition, j never starts view $V + 1$, because if it would, i would start it at most Δ time later. This must mean that j received some propose message and forwarded it to all replicas, otherwise it would have sent a blame in view V. Eventually i receives that message, and starts a timer of 2Δ time. After that timer finishes, i commits to a value, in contradiction. □

Using the lemmas above, we are now ready to prove the main theorem of this section.

Theorem 4.9. *The MixSync protocol described in Algorithm 1 achieves Uniform Correctness, Weak Validity, and Eventual Commitment if $n > k + 2f$.*

Proof. **Uniform Correctness.**
Let i be the first non-Byzantine replica to commit to a value val_i, and let view V be the view in which it commits. Any non-Byzantine replica that commits to a value val' in any view $V' \geq V$ first sends a vote message with that value. From Lemma 4.7, $val' = val_i$, and hence that replica commits to val_i.

Weak Validity.
If no replica is Byzantine, leaders only send propose messages for their input, or for the highest certified value they received in certificate messages. Note that the first certificate constructed by any honest replica is for some replica's input. That is because before such a certificate is constructed the leader must have sent only a propose with its input, and thus all replicas only send vote messages with that input. Using a simple inductive argument, every other certificate must certify some replica's input, because in the view when it is constructed, the leader could have only sent propose messages with some replica's input: either because it is its input, or because it received a certificate for a certified value, which at that point can only be some other replica's input. Finally, replicas only output values for which they received propose messages from the leader, completing the proof.

External Validity.
Non-Byzantine replicas only accept propose messages for a value v if they either contain a proof P with a valid certificate for that value, or if all certificates in P are empty and $validate(v) = true$. Using similar arguments to the ones made in the proof of the Weak Validity property, any valid certificate produced in the protocol must certify a value v such that $validate(v) = true$. Replicas only output values for which they accepted propose messages, completing the proof.

Eventual Commitment.
We would like to show that every honest replica eventually commits to a value. If every honest replica eventually commits, we are done. Otherwise, there exists some honest replica i that never commits. By Lemma 4.8, if there exists some honest replica i that never commits, it must reach every view $V \in \mathbb{N}$. The leaders of the views are changed in a round-robin manner, meaning that there is an honest leader after at most $k + f + 1$ views. Let V be a view with an honest leader L. As shown above, i starts that view at some time t. From Lemma 4.5, every honest replica starts that view by time $t + \Delta$. The leader L sleeps for 3Δ when entering a new view. Again, after at most Δ time, all honest replicas start view V as well. They then sleep for Δ time, and then send L a certificate message. That message arrives at most Δ time later. Honest replicas only save valid certificates containing at least $f + 1$ vote messages, or initiate their first certificate to the valid "empty" certificate. Therefore, the leader receives at least $f + 1$ certificates from the honest replicas. It then construct a valid proof P from the valid certificates it heard. If at least one of those certificates is not empty, L chooses the most recent valid certificate it heard, $cert$, which certifies the value val. Otherwise, it sets val to be its input, which is assumed to be externally valid. Finally, L sends a $\langle \mathsf{propose}, V, val, cert, P \rangle_L$ to all replicas. Every non-Byzantine replica i hears that message at most Δ time later. For similar reasons to the ones above, every non-Byzantine replica that hasn't crashed receives this message at most 5Δ time after starting view V, and sees that it contains a valid proof. This means that none of the non-Byzantine replicas send blame messages which also means

that no blame − cert message contains more than f signed blame messages. Therefore i doesn't leave the view due hearing $f + 1$ blame messages or a blame − cert message. In addition, the aforementioned propose message is the only propose message signed by L, so i doesn't leave the view due to hearing equivocation, an invalid proof, or a value which isn't externally valid. Finally, as stated above i receives the propose message, and commits to val after 2Δ time, completing the proof. □

Next, we will also analyze the required number of messages until all honest replicas commit to some value.

Theorem 4.10. *The protocol described in Algorithm 1 requires $O(n^2(f + k))$ messages to be sent before all honest replicas commit.*

Proof. As shown in the proof of the Eventual Commitment property, every honest replica commits in the first view with an honest leader, if it hasn't done so earlier. The leaders are changed in a round-robin manner, so by the $f + k + 1$'th view, there must be at least one honest leader, meaning that $O(f + k)$ views are required.

Observe a single view. In the beginning of a a view, all non-Byzantine replicas send the new leader certificate messages, totalling in $O(n)$ messages. Then, a propose message is sent to all replicas, requiring another $O(n)$ messages. Lastly, propose messages are forwarded by all replicas to all replicas, along with sending vote messages. This is an all to all round of $O(n^2)$ messages. Note that no honest replica sends more than 2 propose messages, because it forwards any message at most once, and it switches views after receiving two propose messages in a given view. Additionally, each replica might send a single blame and a single blame − cert message to all other parties, totalling in $O(n^2)$ more messages. Therefore, the total number of messages required until all honest replicas commit to some value is $O(n^2(f + k))$, as required. □

For completeness, we also provide a lower bound for consensus with mixed-faults, showing that if there are f Byzantine faults and k crash faults, at least $n > 2f + k$ replicas are needed. The following theorem is proven in Appendix B:

Theorem 4.11. *There is no protocol solving consensus in a synchronous network with k crash faults and f Byzantine faults if $k + 2f \geq n > k + f$ and $f > 0$.*

5 State Machine Replication

As discussed in Section 2.2, a common extension of the task of consensus is the task of State Machine Replication (SMR). In the task of SMR, replicas are required to agree on a value v_s for every log position $s \in \mathbb{N}$, and every client's request must eventually be committed by all honest replicas. It is well known that one way to implement such a protocol is by repeatedly solving the task of consensus, once for each log position. When replicas hear a client's request for a given log position, they forward it to each other and then reach consensus on the input. Using the MixSync protocol then immediately yields a state machine replication protocol. Note that the proof of the MixSync protocol shows that it works even if replicas start each view up to Δ time apart. This is true for the first view as well. Since replicas forward client requests before starting the consensus protocol, they are guaranteed to start the first view up to Δ time apart, and thus all of the properties of the consensus protocol hold. In addition, using an external validity function $validate()$ that returns true on a request only if it is signed by a client yields a highly desirable property: each log position contains a request from one of the clients.

On the other hand, we might want to use more optimized SMR protocols even in a mixed-faults model. For example, the Sync HotStuff protocol presented in [2] is highly optimized for the task of state machine replication, without the overhead of executing many separate consensus instances. The Sync Hotstuff protocol consists of a view-change protocol, and a steady state protocol. Similarly to the MixSync protocol, the steady state protocol of Sync HotStuff proceeds in views, each having a designated leader. In the view, the leader repeatedly sends propose messages, and replicas then forward the propose message and send vote messages. If no equivocation is detected, replicas then commit the newly suggested value. When changing views, replicas wait to hear votes from the last round in order to construct a certificate. Afterwards, replicas send their most recent certificate to the leader, it chooses the most recent certified value, and proceeds with the protocol. Observing the MixSync, modelled after

the Sync HotStuff protocol, we can see that these protocols are nearly identical, except for changes made for it to solve the task of single-shot consensus. Making Sync HotStuff resilient to crash faults as well only requires the use of a crash-resilient non-equivocation round. This means that if replicas make sure to send propose messages before sending the corresponding vote messages, we get a crash-fault resilient version of the Sync HotStuff protocol as well.

5.1 Lower Bound for State Machine Replication

A natural question to ask is whether lower bounds for consensus also imply lower bounds on SMR protocols. In order to show that a similar lower bound can be formulated for SMR protocols, we start by formulating the notion of Write-Once Register.

In a protocol implementing a Write-Once Register, in addition to the replicas, there are c clients. Clients can send two types of commands to the system - READ() and WRITE(v). Each command can eventually terminate on the client's end. In addition, the READ() command can return $\perp$ or a value written by one of the clients. Intuitively, a protocol implements a write-once register, if once a value "is written to the register", every party that reads from it receives that value and cannot be changed afterwards. When observing SMR protocols, we can see that they are required to implement a single Write-Once Register for each log position. Therefore, a lower bound on Write-Once Registers immediately translates to a lower bound on SMR.

Our formulation actually considers a very weak variant of a Write-Once Register, which strengthens the lower bound. The following definition considers a register that is not required to work if several writers are trying to access it at the same time. However, if a write command is completed before any other write command starts, it must succeed. In general, "starting" and "completing" either READ() or WRITE(v) commands are events entirely local to the clients. By that we mean that both commands start when the client locally calls the READ() or WRITE(v) protocols. They are then considered completed when the client receives the read value in case of a READ() command, or when it receives a signal from the network that it has been completed in the case of a WRITE(v) command. This can be modeled as the client receiving a $DONE$ signal, instead of the stronger requirement of receiving either a $SUCCESS$ or a $FAIL$ signal.

Definition 5.1. *A protocol implementing a Write-Once Register has the following properties*

- ***Liveness.*** *Every command by an honest client eventually terminates.*
- ***Validity.*** *If an execution has just one WRITE(x) command that completes at time t then any READ() command that starts after time t returns the value x.*
- ***Correctness.*** *If two READ() commands complete with outputs v and v' such that $v \neq \perp, v' \neq \perp$, then $v = v'$.*

Note that the correctness property above is strongly related to the notion of Uniform Correctness, since omission faulty clients are also required to output the same value from a READ() command if they output any value. We can now show a lower bound similar to the one proven in Theorem 4.11. The lower bound shows that a similar relationship between faulty and honest replicas is required, even if the adversary may only cause replicas to experience omission faults instead of Byzantine faults. We choose to deal with a write-once register instead of multi-write variants, such as the ones described in [7, 25], because a write-once register captures an important property we desire: finality. Like in consensus and SMR protocols, once a value is agreed upon for the register, that value is final and binding (unlike in multi-write registers).

We prove the following theorem in Appendix B, showing that implementing a Write-Once Register, and thus solving SMR is impossible if $n \leq 2t+k$ in a system with n replicas experincing t omission faults and k crash faults. This of course yields an immediate lower of bound $n > 2f + k$ for f Byzantine faults as well.

Theorem 5.2. *There is no protocol implementing a Write-Once Register in a synchronous network with k crash faults, t omission faults and c clients if $k + 2t \geq n > k + t$, $t > 0$ and $c > 1$.*

References

[1] Ittai Abraham, Srinivas Devadas, Danny Dolev, Kartik Nayak, and Ling Ren. Synchronous byzantine agreement with expected $o(1)$ rounds, expected $o(n^2)$ communication, and optimal resilience. In *International Conference on Financial Cryptography and Data Security*, pages 320–334. Springer, 2019. 1

[2] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. Sync hotstuff: Simple and practical synchronous state machine replication. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 106–118. IEEE, 2020. 1, 5, 7, 12

[3] Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. Asymptotically optimal validated asynchronous byzantine agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, pages 337–346, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3293611.3331612. 4

[4] Ittai Abraham, Kartik Nayak, Ling Ren, and Zhuolun Xiang. Byzantine agreement, broadcast and state machine replication with near-optimal good-case latency. *arXiv preprint arXiv:2003.13155*, 2020. 1

[5] Ittai Abraham, Kartik Nayak, Ling Ren, and Zhuolun Xiang. Good-case latency of byzantine broadcast: A complete categorization. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, pages 331–341, 2021. 1

[6] Ittai Abraham and Gilad Stern. Information theoretic hotstuff. In *OPODIS*, volume 184 of *LIPIcs*, pages 11:1–11:16, Dagstuhl, Germany, 2020. Schloss Dagstuhl - Leibniz-Zentrum für Informatik. 4

[7] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM (JACM)*, 42(1):124–142, 1995. 13

[8] Alysson Bessani, João Sousa, and Eduardo E.P. Alchieri. State machine replication for the masses with bft-smart. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 355–362, 2014. doi:10.1109/DSN.2014.43. 1, 3

[9] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '01, pages 524–541, Berlin, Heidelberg, 2001. Springer-Verlag. 4

[10] TH Hubert Chan, Rafael Pass, and Elaine Shi. Pili: An extremely simple synchronous blockchain. *Cryptology ePrint Archive*, 2018. 1

[11] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, page 189–204, New York, NY, USA, 2007. Association for Computing Machinery. doi:10.1145/1294261.1294280. 1, 3

[12] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. Upright cluster services. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 277–290, 2009. 1, 3

[13] M. Correia, Lau Cheuk Lung, N.F. Neves, and P. Verissimo. Efficient byzantine-resilient reliable multicast on a hybrid failure model. In *21st IEEE Symposium on Reliable Distributed Systems, 2002. Proceedings.*, pages 2–11, 2002. doi:10.1109/RELDIS.2002.1180168. 1, 3

[14] M. Correia, N.F. Neves, and P. Verissimo. How to tolerate half less one byzantine nodes in practical distributed systems. In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, 2004.*, pages 174–183, 2004. doi:10.1109/RELDIS.2004.1353018. 1, 3

[15] Danny Dolev. The byzantine generals strike again. *Journal of algorithms*, 3(1):14–30, 1982. 4

[16] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 281–310. Springer, 2015. 1

[17] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. Sbft: A scalable and decentralized trust infrastructure. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 568–580, 2019. doi:10.1109/DSN.2019.00063. 1, 3

[18] Timo Hanke, Mahnush Movahedi, and Dominic Williams. Dfinity technology overview series, consensus system, 2018. URL: https://arxiv.org/abs/1805.04548, doi:10.48550/ARXIV.1805.04548. 1

[19] Dave Levin, John R Douceur, Jacob R Lorch, and Thomas Moscibroda. Trinc: Small trusted hardware for large distributed systems. In *NSDI*, volume 9, pages 1–14, 2009. 1, 3

[20] Patrick Lincoln and John Rushby. A formally verified algorithm for interactive consistency under a hybrid fault model. In *FTCS-23 The Twenty-Third International Symposium on Fault-Tolerant Computing*, pages 402–411. IEEE, 1993. 1, 3

[21] Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, and Marko Vukolić. Xft: Practical fault tolerance beyond crashes. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 485–500, 2016. 1

[22] Yuan Lu, Zhenliang Lu, Qiang Tang, and Guiling Wang. Dumbo-mvba: Optimal multi-valued validated asynchronous byzantine agreement, revisited. In *Proceedings of the 39th Symposium on Principles of Distributed Computing*, PODC '20, page 129–138, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3382734.3405707. 4

[23] Silvio Micali and Vinod Vaikuntanathan. Optimal and player-replaceable consensus with an honest majority. 2017. 1

[24] Rafael Pass, Lior Seeman, and Abhi Shelat. Analysis of the blockchain protocol in asynchronous networks. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 643–673. Springer, 2017. 1

[25] Gary L. Peterson and James E. Burns. Concurrent reading while writing ii: The multi-writer case. In *28th Annual Symposium on Foundations of Computer Science (sfcs 1987)*, pages 383–392, 1987. doi:10.1109/SFCS.1987.15. 13

[26] 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 *2010 IEEE/IFIP International Conference on Dependable Systems Networks (DSN)*, pages 353–362, 2010. doi:10.1109/DSN.2010.5544295. 1, 3

[27] Marco Serafini and Neeraj Suri. The fail-heterogeneous architectural model. In *2007 26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007)*, pages 103–113, 2007. doi:10.1109/SRDS.2007.33. 1, 3

[28] Nibesh Shrestha, Ittai Abraham, Ling Ren, and Kartik Nayak. On the optimality of optimistic responsiveness. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 839–857, 2020. 1

[29] Hin-Sing Siu, Yeh-Hao Chin, and Wei-Pang Yang. Byzantine agreement in the presence of mixed faults on processors and links. *IEEE Transactions on Parallel and Distributed Systems*, 9(4):335–345, 1998. doi:10.1109/71.667895. 1, 3

[30] Philip Thambidurai and You-Keun Park. Interactive consistency with multiple failure modes. In *Proceedings Seventh Symposium on Reliable Distributed Systems*, pages 93–94. IEEE Computer Society, 1988. 1, 3

[31] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Verissimo. Efficient byzantine fault-tolerance. *IEEE Transactions on Computers*, 62(1):16–30, 2013. doi:10.1109/TC.2011.221. 1, 3

A Proof of Publicly Verifiable Crusader Broadcast

Theorem 3.1. *The pair* $(PVCA, check)$ *as described above are a Publicly Verifiable Crusader Agreement scheme for* $n > 2f + k$.

Proof. We prove each property individually.

Verifiability.

Let i be a non-Byzantine replica that outputs val_i, π_i. We first prove that if $val_i \neq \bot$, it is not possible for a replica to produce a pair val', π' s.t. $val' \notin \{val_i, \bot\}$ and $check(val', \pi') = True$. We will actually show that no non-Byzantine replica sent a vote message containing any $val' \neq val$. If such π' exists, it must contain a signed vote message with the value val' from $f+1$ replicas. At least one of those replicas must be non-Byzantine, and thus showing the above proves that it is impossible to construct π'.

Assume i output val_i, π_i at time t. This means i sent a $\langle \mathsf{propose}, val_i \rangle_L$ and a $\langle \mathsf{vote}, val_i \rangle_i$ message at time $t - 2\Delta$ to all replicas, which arrived by time $t - \Delta$. Let j be a non-Byzantine replica. It receives i's messages at most by time $t - \Delta$, if it hasn't crashed earlier. Since i decided at time t, it did not hear any equivocating $\mathsf{propose}$ message from j by that time. This means j did not send a $\mathsf{propose}$ message for any value $val' \neq val_i$ by time $t - \Delta$, as i would have received it before outputting val_i and detected equivocation. This means that j didn't send a vote for that value before time $t - \Delta$ either. As noted above, j receives $\langle \mathsf{propose}, val_i \rangle_L$ from i by time $t - \Delta$, and thus it won't send a vote message for $val' \neq val_i$ after time $t - \Delta$, as it would detect equivocation and proceed immediately to compute its output.

Next we show that $check(val_i, \pi_i) = True$. We first deal with the case that $val_i \neq \bot$. Assume by way of contradiction that some honest replica j completed the protocol before time $t - \Delta$. If it completed it as a result of receiving two different $\mathsf{propose}$ message, a $\mathsf{propose}$ message with an invalid value, blame messages from $f + 1$ different replicas or a $\mathsf{blame-cert}$ message. Before doing so, it forwards $\mathsf{propose}$ messages it received or sends a $\mathsf{blame-cert}$ message. Replica i receives those messages before time t and completes the protocol, contradicting the fact that it completed it at time t. Alternatively, j completed the protocol before time $t - \Delta$ as a result of receiving a $\mathsf{propose}$ message before time $t - 3\Delta$. In that case, it forwards the message and i receives it before time $t - 2\Delta$. Then, i completes the protocol before time t, either as a result of the 2Δ timer finishing or after receiving another message. This is also a contradiction, and thus j does not complete the protocol before time $t - \Delta$. As stated above, every honest replica j receives a $\langle \mathsf{propose}, val_i \rangle_L$ message from i by time $t - \Delta$. Also, j did not detect equivocation by that time, so it sends a vote message containing val_i. We get that all honest replicas send $\langle \mathsf{vote}, val_i \rangle$ by time $t - \Delta$. Replica i receives those message by time t, meaning these votes will enter his proof π_i. As there are $n - f - k > f$ honest replicas, i would have at least $f + 1$ votes in his proof π_i, and we get $check(val_i, \pi_i) = True$. On the other hand, if $val_i = \bot$, then one of the following things must have happened:

- i received a $p_1 = \langle \mathsf{propose}, value \rangle_s$ message such that $validate(value) = False$ and it output $\bot, p_1$;
- i received two messages $p_1 = \langle \mathsf{propose}, value \rangle_s, p_2 = \langle \mathsf{propose}, value' \rangle_s$ such that $value \neq value'$ and it output $\bot, \{p_1, p_2\}$;
- or i received blame messages $b_1, \ldots, b_{f+1}$ from $f + 1$ different replicas or a $\mathsf{blame-cert}$ message containing such messages and output $\bot, \{b_1, \ldots, b_{f+1}\}$.

In all three cases, $check(\bot, \pi_i) = True$, completing the proof.

Termination.

We will split the proof into cases. First assume some honest replica terminates as a result of receiving a $\langle \mathsf{propose}, val \rangle_s$ message such that $validate(val) = False$ or $\langle \mathsf{propose}, val \rangle_s$ and $\langle \mathsf{propose}, val' \rangle_s$ messages such that $val \neq val'$. In that case, it forwards the $\mathsf{propose}$ messages to all replicas. They receive those messages up to Δ time later and terminate for the same reason if they haven't done so earlier. From this point on, assume no honest replica terminates for either of those reasons.

Otherwise, assume no honest replica received a $\mathsf{propose}$ message within Δ time from starting the protocol. Before Δ time has passed, no non-Byzantine replica completes the protocol: no non-Byzantine replica detects an invalid propose by assumption and no non-Byzantine replica sends a blame message. This means that no non-Byzantine replica completes the protocol as a result of receiving blame messages from $f + 1$ different replicas or as a result of receiving a $\mathsf{blame-cert}$ message with $f + 1$ such blame

messages until Δ time has passed. If that is the case, all honest replicas stay in the protocol for Δ time and send a blame message. As there are $n - f - k > f$ non-faulty replicas, they all receive $f + 1$ blame messages, and output $(\bot, \{b_1, ..., b_{f+1}\})$ where $\{b_1, ..., b_{f+1}\}$ are all the blame messages received, and quit the protocol.

Alternatively, at least one honest replica i receives a propose message signed by the leader within Δ time from starting the protocol. i then forwards the propose message to all replicas and start a timer, completing the protocol within 2Δ time at the latest. As i is honest, it will successfully forward the propose received to all other honest replicas. Each such replica starts a timer, if it hasn't done so earlier, and complete the protocol at most 2Δ time later.

Validity.

Let s be an honest sender with the input x such that $validate(x) = True$. s starts by sending a $\langle \text{propose}, x \rangle_s$ to all replicas. Every non-Byzantine replica that hasn't crashed receives that message by time Δ, and thus they don't send blame messages. This means that overall, no non-Byzantine replica proceeds to the "output calculation" step of the protocol as a result of receiving blame messages from $f + 1$ different replicas or as a result of receiving a $\text{blame} - \text{cert}$ message containing $f + 1$ such blame messages. Since that is the only signed propose message sent by the dealer, no non-Byzantine replica proceeds to output calculation after detecting equivocation. Finally, by assumption $validate(x) = True$, so no non-Byzantine replica proceeds to output calculation as a result of receiving an invalid proposal. Therefore, 2Δ after receiving the propose message, all non-Byzantine replicas that haven't crashed yet proceed to output calculation and output x, π for some proof π.

Next, we show that no val, π can be produced by any replica such that $check(val, \pi) = True$ unless $val = x$. If $val = \bot$, then the proof must either be a propose message from the dealer with an invalid x', two equivocating propose messages from the dealer, or blame messages from $f + 1$ different replicas. As shown above, no non-Byzantine replica sends a blame message, so the proof cannot consist of $f + 1$ such messages. In addition, the only propose message sent by the leader contains its externally valid input x. This means that the proof cannot contain an invalid x' or two equivocating messages from the dealer. Combining these observations, if a replica produces some val, π such that $check(val, \pi) = True$, $val \neq \bot$. On the other hand, if $val \neq \bot$, then a proof for it must contain $\langle \text{vote}, val \rangle$ messages signed by $f + 1$ replicas, which contains at least one message from a non-Byzantine replica. Before sending a $\langle \text{vote}, val \rangle$ message, a non-Byzantine replica must receive a $\langle \text{propose}, val \rangle_s$ message signed by the sender. The only propose message sent by the sender is the $\langle \text{propose}, x \rangle_s$ message, meaning that if $check(val, \pi) = True$ for $val = x$. Following a similar argument, a non-Byzantine but faulty sender never sends a $\langle \text{propose}, val \rangle_s$ message with any value but x, meaning that if $check(val, \pi) = True$ and $val \neq \bot$, then $val = x$.

External Validity.

If some non-Byzantine replica i outputs a pair val, π such that $val \neq \bot$, then it first received a $\langle \text{propose}, val \rangle_s$ message. If $validate(val) = False$, then i would have immediately output $\bot$ upon receiving the propose message instead.

Correctness. Let i, j be two non-Byzantine replicas that output v_i, π_i and v_j, π_j respectively such that $v_i \neq \bot$ and $v_j \neq \bot$. From the Verifiability property, $check(v_i, \pi_i) = check(v_j, \pi_j) = True$. In addition, since v_i is not Byzantine, no replica can produce v', π' such that $v' \notin \{v_i, \bot\}$ and $check(v', \pi') = True$. This includes j, meaning that it must be the case that $v_i = v_j$. □

B Lower Bounds

In this section we prove both lower bounds discussed above. The first lower bound deals with the task of consensus with mixed faults.

Theorem 4.11. *There is no protocol solving consensus in a synchronous network with k crash faults and f Byzantine faults if $k + 2f \geq n > k + f$ and $f > 0$.*

Proof. Assume by way of contradiction that there exists a protocol solving consensus in the model we have described above, which is resilient to k crash faults and f Byzantine faults for $k + 2f \geq n > k + f$ and $f > 0$.

Next, we shall divide all replicas in the model into 3 groups:

- A, a group of k replicas.

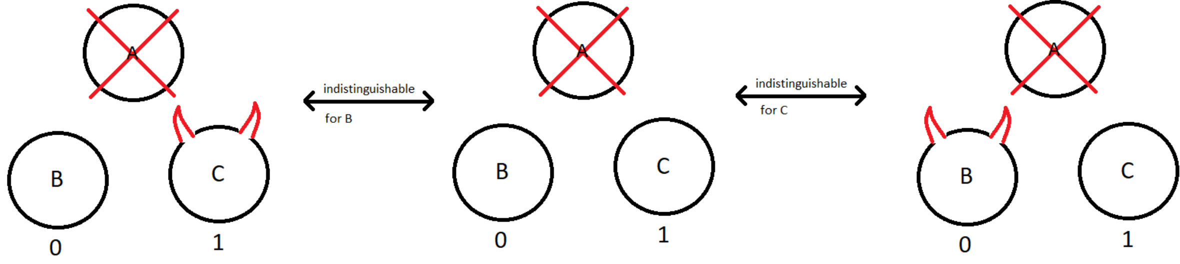

Figure 1: Theorem 4.11, Indistinguishabilty Argument

- B, a group of f replicas.
- C, the rest of the replicas, having $n-k-f$ replicas. Note that by the assumption that $n > k+f$, we get $|C| = n-k-f > 0$.

Next, we describe the following 3 worlds:

- **World 0**: All replicas in A are crash faulty, all replicas in C are Byzantine, all replicas in B are honest. All replicas have 0 as input. The adversary immediately crashes all replicas in A. It then directs all Byzantine replicas in C to act as honest replicas with input 1.
- **World 1**: All replicas in A are crash faulty, all replicas in B are Byzantine, all replicas in C are honest. All replicas have 1 as input. The adversary immediately crashes all replicas in A. It then directs all Byzantine replicas in B to act as honest replicas with input 0.
- **Hybrid World**: All replicas in A are crash faulty, the rest of the replicas ($B \cup C$) are honest. The adversary crashes all replicas in A immediately. All replicas in B have the input 0, and all replicas in C have the input 1.

We notice that both World 0 and World 1 are possible as the groups B and C's sizes are each f or less and A's size is k. This means that the adversary may corrupt all replicas in either B or C and crash all replicas in A, and thus all three worlds are possible. In world 0, all honest replicas have the input 0, so from the Validity and Termination properties of the protocol, they must all terminate and output 0. Similarly, in world 1, all honest replicas have the same input of 1, and hence they must terminate and output 1.

In all three worlds, replicas in A crash in the beginning of the run, replicas in B act as honest replicas with the input 0 and all replicas in C act as honest replicas with the input 1. Therefore, replicas in B cannot distinguish between World 0 and Hybrid World, and must output 0 in both. Similarly, replicas in C cannot distinguish between World 1 and Hybrid World, and must output 1 in both. All in all, in the hybrid world in which all replicas in $B \cup C$ are honest, the non-empty group C must decide 1 while the non-empty group B must decide 0, breaking Correctness and reaching a contradiction.

□

Using similar ideas, we can prove a lower bound for implementing Write-Once Registers as well

Theorem 5.2. *There is no protocol implementing a Write-Once Register in a synchronous network with k crash faults, t omission faults and c clients if $k + 2t \geq n > k + t$, $t > 0$ and $c > 1$.*

Proof. Assume by way of contradiction that there exists a protocol solving consensus in the model we have described above, which is resilient to k crash faults and t omission faults for $k + 2t \geq n > k + t$, $t > 0$ and $m > 1$.

Next, we shall divide all replicas in the model into 3 groups:

- A, a group of k replicas.
- B, a group of t replicas.

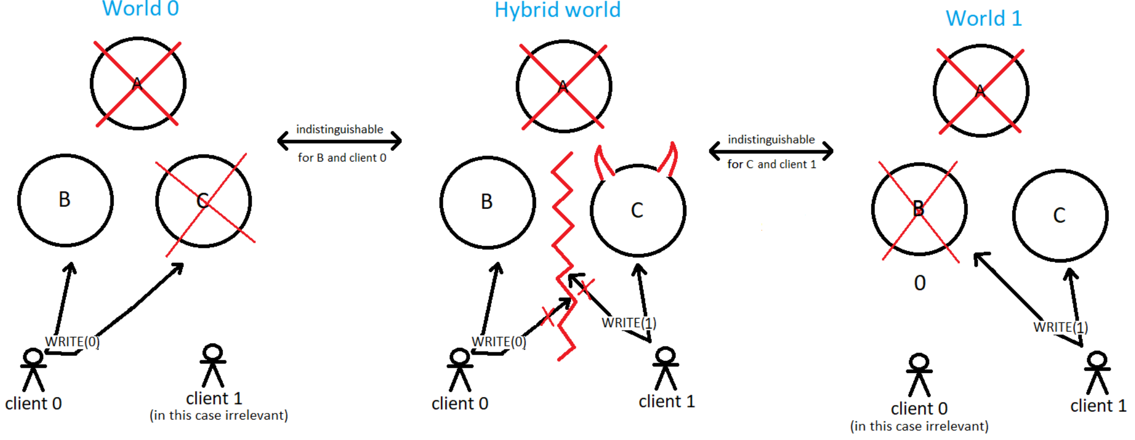

Figure 2: Theorem 5.2, Indistinguishabilty Argument For the Write once Model. WRITE commands to replicas in A omitted for simplicity.

- C, the rest of the replicas, having $n-k-t$ replicas. Note that by the assumption that $n > k+f$, we get $|C| = n-k-t > 0$.

Additionally, we have two clients using the system, client 0 and client 1. Note that if there is a single client, it can trivially implement a write-once register, by locally performing all actions, without communicating with any of the replicas.

Next, we describe the following 3 worlds:

- **World 0**: All replicas in A are crash faulty, all replicas in C are omission faulty, all replicas in B are honest. In addition, client 1 is omission faulty, and client 0 is honest. The adversary immediately crashes all replicas in A. All messages to and from client 1 and replicas in C are omitted. Client 0 sends a $WRITE(0)$ command to the system. After completing the command, client 0 sends a $READ()$ command and waits for it to terminate.

- **World 1**: All replicas in A are crash faulty, all replicas in B are omission faulty, all replicas in C are honest. In addition, client 0 is omission faulty and client 1 is honest. The adversary immediately crashes all replicas in A. All messages to and from client 0 and replicas in B are omitted. Client 1 sends a $WRITE(1)$ command to the system. After completing the command, client 1 sends a $READ()$ command and waits for it to terminate.

- **Hybrid World**: All replicas in A are crash faulty, the replicas in B are honest, and the replicas from C are omission faulty. In addition, both client 0 and client 1 are omission faulty. The adversary crashes all replicas in A immediately. Throughout the run of the protocol, messages between different replicas in B and messages between replicas in B and client 0 are delivered. Similarly, all messages between different replicas in C and messages between replicas in C and client 1 are delivered. All other messages are omitted, either due to replicas in C being faulty, or due to the clients being faulty. Client 0 sends a $WRITE(0)$ command and client 1 sends a $WRITE(1)$ command. After completing their respective $WRITE$ commands, both clients send a $READ()$ command and wait to complete those commands.

We notice that both World 0 and World 1 are possible as the groups B and C's sizes are each t or less and A's size is k. This means that the adversary may crash all replicas in A and cause all replicas in either B or C to be omission faulty all replicas, and thus all three worlds are possible. In world 0, client 0 is honest, and it sends a $WRITE(0)$ command. From the Liveness property of the protocol, the command must eventually terminate. Following that, client 0 sends a $READ()$ command, which must also terminate for the same reason. At that time, client 0 completed the $WRITE(0)$ command, and no other $WRITE(x)$ call has been made. Therefore, from the Validity property of the protocol, the $READ()$ command must return 0. Similarly, in world 1, both the $WRITE(1)$ and $READ()$ commands made by client 1 must terminate, and the read command must return 1.

In all three worlds, replicas in A crash in the beginning of the run. In addition, in world 0 and in hybrid world, client 0 calls the same commands and all messages between different replicas in B and between replicas in B and client 0 are delivered. No other message is delivered to replicas in B or to client 0 in both worlds. This means that replicas in B and client 0 have identical views of the system throughout the runs of the protocol in World 0 and in Hybrid World. Therefore, replicas in B and client 0 must act identically in both worlds, which means that both of client 0's commands terminate and its $READ()$ command outputs 0 in Hybrid World as well. Using identical arguments with regard to replicas in C and client 1, their views must be identical in World 1 and Hybrid World. Therefore, client 1 must act identically in both worlds, so both its commands terminate and its $READ()$ command outputs 1 in both world. In other words, two different clients output two different non-$\perp$ values from their $READ()$ commands, contradicting the Correctness property of the protocol.

□

C MixSync Pseudocode

The pseudocode in Algorithms 1 and 2 describes the exact behavior of the MixSync protocol. Every replica runs MixSync by calling the run function with its own input. When multicasting a message, the multicast() function will check if the replica already has multicasted the message to all replicas. It will not forward a message that has already been sent.

Algorithm 1 MixSync

```
1: def run(input)
2:     self.input = input
3:     self.view = 0
4:     self.certificate = ⊥
5:     self.proof = {}
6:     self.proposal = ⊥
7:     self.votes = {}
8:     view_change(), blame_thread()                              ▷ in a new thread
9:     while True do
10:        m = receive_message()        ▷ if received while sleeping, handle after "waking up"
11:        if verify(m) == True then
12:            handle_new_message(m)                                  ▷ In a new thread
13:        end if
14:    end while
15:
16: def view_change()
17:     self.view++
18:     sleep(Δ)               ▷ Handle all messages received during sleep when command ends.
19:     if self.votes has at least f + 1 vote messages for the same value val then
20:         self.certificate = {m—m is a vote message for val}  ▷ Choose any such val if more than one
    exists.
21:     end if
22:     self.proposal = ⊥, self.votes={}
23:     new_leader = get_leader_of_view(self.view)
24:     send_to(new_leader, ⟨certificate, self.view, self.certificate⟩_r)   ▷ self.cert may be ⊥
25:     self.blames = {}
26:     self.proof = {}
27:     self.time_entered_view = get_current_time()
28:     if self == next_leader then
29:         sleep(2Δ)
30:         cert = the valid certificate received in a certificate message s.t. cert.view is maximal
31:         if cert == ⊥ then
32:             multicast(⟨propose, self.view, self.input, self.proof⟩_r)
33:         else
34:             multicast(⟨propose, self.view, cert.value, cert, self.proof⟩_r)
35:         end if
36:     end if
37:
38: def blame_thread()
39:     while True do
40:         if get_current_time()-self.time_entered_view¿3Δ and self.proposal == ⊥ then
41:             multicast(⟨blame, self.view⟩_r)                    ▷ send only once per view
42:         end if
43:     end while
```

Algorithm 2 Mixed-Fault Consensus - pseudo code for replica r

```
1:  def handle_new_message(m)
2:      if m.view ≠ self.view then
3:          if m.type ≠ vote or get_current_time()-self.time_entered_view¿Δ or m.view¡self.view-1 then
4:              return
5:          end if
6:      end if
7:      if m.type == propose and m.signature is of get_leader(m.view) then
8:          multicast(m)
9:          if m.proof is contains an invalid certificate or fewer than f + 1 certificates then
10:             view_change() and return
11:         else if the most recent non-⊥ certificate in m.proof doesn't certify m.value then
12:             view_change() and return
13:         else if all certificates in m.proof are ⊥ and validate(m.value) = False then
14:             view_change() and return
15:         else if self.proposal≠ ⊥ and self.proposal≠ m then
16:             view_change() and return
17:         end if
18:         self.proposal=m
19:         multicast(⟨vote, m.value, m.view⟩_r)
20:         wait(2Δ)
21:         if view hasn't changed while waiting then
22:             commit(m.value)
23:         end if
24:     end if
25:     if m.type == blame then
26:         self.blames.add(m)
27:         if —self.blames— == f + 1 then
28:             multicast(⟨blame, self.blames⟩_r)
29:             view_change()
30:         end if
31:     end if
32:     if m.type == blame − cert then
33:         if m.blame_cert contains blame messages from self.view signed by f + 1 different replicas then
34:             multicast(m)
35:             view_change()
36:         end if
37:     end if
38:     if m.type == vote and m.view ≤ self.view then
39:         if received a propose message with same view and value from same replica then
40:             self.votes.add(m)
41:         end if
42:     end if
43:     if m.type == certificate then
44:         if self == get_leader(m.view) and m.certificate is valid then
45:             self.proof.add(m)
46:         end if
47:     end if
```