DAMYSUS: Streamlined BFT Consensus Leveraging Trusted Components

Jérémie Decouchant*
j.decouchant@tudelft.nl
Delft University of Technology, The Netherlands

David Kozhaya*
david.kozhaya@ch.abb.com
ABB Research, Switzerland

Vincent Rahli*
vincent.rahli@gmail.com
University of Birmingham, UK

Jiangshan Yu*
J.Yu.Research@gmail.com
Monash University, Australia

Abstract

Recently, *streamlined* Byzantine Fault Tolerant (BFT) consensus protocols, such as HotStuff, have been proposed as a means to circumvent the inefficient view-changes of traditional BFT protocols, such as PBFT. Several works have detailed trusted components, and BFT protocols that leverage them to tolerate a minority of faulty nodes and use a reduced number of communication rounds.

Inspired by these works we identify two basic trusted services, respectively called the Checker and Accumulator services, which can be leveraged by streamlined protocols. Based on these services, we design `Damysus`, a streamlined protocol that improves upon HotStuff's resilience and uses less communication rounds. In addition, we show how the Checker and Accumulator services can be adapted to develop `Chained-Damysus`, a *chained* version of `Damysus` where operations are pipelined for efficiency. We prove the correctness of `Damysus` and `Chained-Damysus`, and evaluate their performance showcasing their superiority compared to previous protocols.

1 Introduction

Consensus is a crucial building block of many distributed systems including state machine replication and blockchains. Such systems comprise multiple software and hardware components that are bound to eventually fail, potentially causing system malfunction or unavailability. To this end, a distributed system relies on a consensus protocol to agree on actions critical to the system's correct operation.

In particular, Byzantine Fault Tolerant (BFT) consensus protocols allow systems to withstand arbitrary failures [1, 2, 3]. However, being able to tolerate arbitrary (Byzantine) component failures, induces a non-negligible system overhead, namely in the required number of nodes that should be present in the system as well as the communication complexity. For example, traditional BFT protocols, such as PBFT [4], typically require $3f+1$ nodes to tolerate up to f Byzantine faulty nodes. Moreover, albeit being efficient in the normal case, i.e., when the current leader is correct, traditional BFT protocols use complex "view-change" operations to replace possibly malicious (compromised) leaders and often require the nodes to "synchronize" and transfer their states.

In order to circumvent such complex view changes, recent "streamlined" BFT protocols [5][1] such as PiLi [6], PaLa [7], Tendermint [8], HotStuff [9], and Streamlet [10] rotate the leader on each command. Instead of performing a state transfer when switching to another leader, they utilize a simpler locking mechanism combined with an additional communication phase (compared to PBFT-like protocols) to guarantee both safety and liveness. Hence, streamlined protocols are conceptually simpler. An advantage of such a leader rotation mechanism, used for example in [11, 12, 13, 14, 15], is that it can be fairer to clients by preventing a fixed leader from favoring some clients over others for an extended period of time. Another typical feature of streamlined protocols is their linear message complexity as opposed to the quadratic all-to-all communication of traditional solutions. Nevertheless, streamlined solutions still require at least $3f+1$ nodes to tolerate f Byzantine nodes, and they introduce an additional communication phase that inflicts higher system latency.

For some applications being able to tolerate more than one-third of Byzantine faults is crucial [16]. Traditional BFT protocols have typically relied on *hybrid* solutions to reduce the number of nodes required to ensure system safety as well as to decrease system latency. Those solutions rely on secure elements that execute key functionalities in trusted execution environments such as inside Intel SGX enclaves [17]. For example, MinBFT [18] relies on a trusted message counter to increase the resilience of PBFT, permitting a system of

*Corresponding Authors

[1] A streamlined protocol is defined in [5] as one that follows a unified propose-vote paradigm, and that integrates view-changes in the normal case operation via a rotating leader.

EuroSys '22, April 5–8, 2022, RENNES, France
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9162-7/22/04.
https://doi.org/10.1145/3492321.3519568

size N to tolerate $\lfloor \frac{N-1}{2} \rfloor$ faulty nodes while simultaneously reducing the number of communication rounds in normal case operations from 3 in PBFT to 2 in MinBFT.

While hybrid solutions have been largely explored in the context of traditional BFT protocols focusing on minimizing the functionalities dedicated to trusted components, they have not yet been explored to the same extent in the context of streamlined protocols. To this end, this paper investigates the design of hybrid streamlined solutions focusing on HotStuff-like protocols. Interestingly, we show that the previous way of using trusted message counters (simplest known trusted components used in traditional BFT protocols) [18, 19, 20] is not sufficient for simultaneously making HotStuff-like protocols more resilient and faster (see §4).

Still adhering to a minimalist approach when designing trusted components, we introduce two simple trusted services called CHECKER and ACCUMULATOR, which contribute respectively to increasing a streamlined protocol's resilience and to decreasing its latency. Roughly speaking, our CHECKER service helps ensure that nodes cannot vote for conflicting blocks, or lie about the blocks they have already voted for, while our ACCUMULATOR service guarantees that leaders can only propose blocks that are consistent with previous votes. We then present Damysus,[2] a streamlined protocol, based on *basic* HotStuff, which makes use of the CHECKER and ACCUMULATOR services. Damysus requires $2f+1$ nodes to tolerate f Byzantine failures and can terminate in 2 communication phases as opposed to 3 in basic HotStuff.

In this sense, Damysus simultaneously tolerates a minority of Byzantine nodes and reduces the number of communication phases to its known minimum. The introduced trusted services are simple and do not require changes to HotStuff's overall behavior, preserving linear view change and optimistic responsiveness [9] where nodes reach consensus at the pace of the actual (vs. maximum) network delay.

We also generalize our results showcasing how we can seamlessly transfer Damysus to Chained-Damysus using the exact same trusted services to also support a *chained* version of HotStuff, where operations are pipelined so that multiple blocks are concurrently processed.

In short, our contribution can be summarized as follows.

1. We introduce two simple trusted services, which when applied to HotStuff (without changing its overall behavior), result in Damysus, a streamlined consensus protocol with improved resilience and latency. Damysus requires $2f+1$ nodes and terminates in 2 phases.
2. We also introduce Chained-Damysus, a chained version of Damysus for higher throughput.
3. We provide theoretical proofs that all protocols are safe and live.
4. We show the performance superiority of our protocols compared to baseline HotStuff-like protocols. For example, using 256B payload messages, Damysus has an average throughput increase of 87.5% and an average latency decrease of 45% compared to basic HotStuff. Chained-Damysus has a throughput increase of 50.5% and latency decrease of 32.1% compared to chained HotStuff.

The rest of the paper is organized as follows. §2 reviews the related work. §3 provides background on HotStuff. §4 presents the functionality of our CHECKER and ACCUMULATOR trusted services. §5 describes our system model. §6 introduces Damysus and gives the intuition behind its design principles. §7 presents Chained-Damysus. §8 provides a performance evaluation. §9 concludes this paper.

2 Related Work

We summarize our analysis of the related work in Table 1 and elaborate further in the following.[3]

Hybrid BFT protocols, which leverage the use of trusted hardware components to reinforce different aspects of BFT protocols, ranging from reconfiguration [23] and proactive recovery [24] to the fault-tolerance and performance [25, 26, 18, 19, 20, 27], have been studied in the context of leader-based BFT protocols. While few studies have also been focusing on hybrid leaderless BFT protocols [28], the focus of this section is on leader-based hybrid BFT protocols that aim at improving fault-tolerance and/or performance.

Previous literature explored techniques such as attested append-only memory (A2M) [25], and trusted incrementer (TrInc) [26] where multiple trusted monotonic counters are used to prevent equivocation and tolerate more faults. Later, MinBFT [18] presented an approach that provides the same properties using a single trusted monotonic counter while additionally reducing the number of phases in PBFT from 3 to 2 in the normal case operation. CheapBFT [19], as well as its generalization ReBFT [29], further improved this result by enabling the system to "optimistically" operate with as little as $f+1$ active replicas, while f other replicas stay passive during the normal case operation. Hybster [20] also explored the possibility to parallelize instances with TrIncX, a TrInc-like trusted monotonic counter. FastBFT [21] is similar to CheapBFT in the sense that it is both optimistic (making use of $f+1$ active nodes in the normal case operation) and relies on a trusted monotonic counter. In addition, FastBFT relies on a TEE-based secret sharing mechanism, which compounded with trusted monotonic counters, allows reducing the message complexity of the protocol. Recently, TBFT [27] was proposed to improve performance by requiring backups to communicate only with the leader (with complexity $O(N)$ as in HotStuff) rather than an all-to-all communication pattern (with complexity $O(N^2)$ as in PBFT). In addition to relying on trusted monotonic counters, TBFT uses a trusted

[2] Damysus was the fastest of all the Giants in the Greek mythology.

[3] In Table 1, the communication steps exclude the interaction with clients, which are common to all protocols: requests sent by clients to replicas, and replies sent by replicas to clients.

Table 1. Comparative analysis of Damysus and Chained-Damysus with the related work. View-change communication steps are indicated between parentheses when applicable. Message counts include self-messages.

	# replicas	Commun. steps	#Msgs normal case	#Msgs view change	Optimistic exec.?	Trusted component & Storage complexity
PBFT [4]	$3f$+1	3 (+2)	$18f^2$+$15f$+3	$9f^2+6f+1$	No	-
FastBFT [21]	f+1 act. & f pass.	5 (+3)	$6f$+5	$8f^2+8f+2$	Yes	Secret generation - Constant
MinBFT [18]	$2f+1$	2 (+3)	$4f^2$+$6f$+2	$8f^2$+$6f$+1	No	Trusted counter - Constant
CheapBFT [19]	f+1 act. & f pass.	3 (+3)	$2f^2$+$4f$+2	$8f^2$+$6f$+1	Yes	Trusted counter - Constant
HotStuff [9]	$3f+1$	8	$24f$+8	-	No	-
HotStuff-M [22]	$2f+1$	11	$(24+9d)f+(8+3d)$	-	No	Append-only logs - Linear with # msgs
Damysus	$2f+1$	6	$12f+6$	-	No	Checker & Accumulator - Constant
Chained-Damysus	$2f+1$	6	$12f+6$	-	No	Checker & Accumulator - Constant

message sharing mechanism to generate quorum certificates representing messages collected from f+1 replicas.

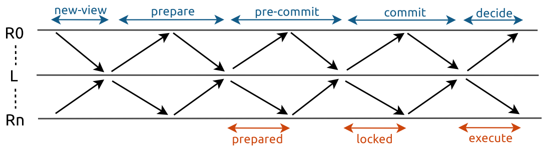
In the context of streamlined BFT protocols, hybrid solutions were first mentioned for LibraBFT [30], a protocol built on HotStuff, to possibly reduce the attack surface. Fast-HotStuff [31] is a version of HotStuff with one less communication phase, which does not rely on trusted components. Instead, leaders send proofs that the blocks they extend are the highest received blocks. This requires larger messages (containing an aggregated vector of $2f$+1 quorum certificates) but improves latency and prevents from creating forks.

Later, HotStuff-M and VABA-M [22] proposed the use of trusted components in the form of trusted logs to improve fault tolerance without worsening the communication complexity of the underlying BFT protocol: one trusted log is used for each protocol phase with an additional log for tracking views. This approach also requires changes to the underlying BFT protocol: the protocol would need to use and maintain expander graphs to diffuse messages. Such graphs introduce more network traffic, extra storage overhead and their impact on throughput/latency remains open.

A related approach consists in transforming crash fault tolerant protocols to BFT ones using trusted components, such as [32], which essentially works by running all state machines inside a trusted environment, leading to large enclaves; and [33], which relies on smaller trusted components, that however need to sign the entire history of sent and received messages, leading to large messages.

Simultaneously achieving improved communication complexity with optimal resilience of streamlined BFT protocols remains an open challenge. This drives us to propose the first hybrid streamlined BFT system, which tolerates a minority of Byzantine faults while reducing the number of communication phases to improve the latency and throughput of the underlying algorithm. In addition, unlike HotStuff-M, our approach does not change the overall behavior of the underlying protocol, retaining its conceptual simplicity and does not introduce extra network traffic in any phase. It also arguably requires minimal storage (including only a counter representing the current view/phase and some information regarding the locally prepared and locked blocks).

Figure 1 Communication phases in HotStuff.



3 HotStuff in a Nutshell

HotStuff [9] is a BFT protocol whose communication complexity grows linearly with the total number N of nodes, making it remarkably efficient. HotStuff requires $N \geq 3f+1$ nodes to tolerate f Byzantine faults. Nodes build a chain of blocks by voting for extensions, which are proposed by the leaders of *views* (successive rounds).

HotStuff comes in two versions: (1) *Basic* HotStuff, where nodes vote on a single block per view, and (2) *chained* (or *pipelined*) HotStuff that allows pipelined votes to simultaneous progress on several blocks per view.

Communication phases To execute a block, HotStuff employs several *phases* per view v, which Fig. 1 illustrates. In that figure, time progresses from left to right, and L is the leader of the view v among the nodes $R_0, \ldots, R_n$. HotStuff is referred to as a 3-phase protocol, as it has 3 *core* phases: prepare, pre-commit, and commit to agree on blocks, complemented by 2 additional half phases: new-view to submit latest prepared blocks, and decide to execute blocks once it has be judged safe to do so.

(1) In the **new-view** phase[4] each node sends its latest prepared block and view number to the leader;
(2) In the **prepare** phase, after having received $2f$+1 pairs of a block and a view number, the leader creates a new block by extending the block with the largest view number among all received pairs. The leader proposes a block by sending it to all other nodes who will vote for it if it is correct.
(3) In the **pre-commit** phase, the leader gathers $2f$+1 votes for its proposal and marks the proposed block as being *prepared* (marked in orange in Fig. 1). The collection of $2f$+1

[4] HotStuff has a new-view phase, which is not explicitly referred to as such in [9]. For presentation purposes we group new-view messages in a new-view phase here.

votes forms a quorum certificate indicating that the proposed block is prepared. The leader forwards this certificate to the other nodes, which mark the proposed block as being prepared after verifying the quorum certificate. Once a node has prepared a block, it votes for it in this phase.
(4) In the **commit** phase, the leader gathers a quorum certificate on a prepared block, *locks* it (marked in orange in Fig. 1) and forwards the quorum certificate to the other nodes so that they lock the same block. Once a node has locked a block, it votes for it in this phase.
(5) In the **decide** phase, the leader gathers a quorum certificate of $2f+1$ votes on a locked block, *executes* that block (marked in orange in Fig. 1) and forwards the certificate to the other nodes so that they also execute the locked block.

Basic HotStuff uses a different leader in each view, and each view is in charge of proposing a single extension to the current blockchain.

Locking scheme A new-view message only contains the certificate of what a node believes to be the latest prepared block, rather than all certificates for all prepare messages since the last checkpoint as in traditional protocols such as PBFT [4] and BFT-SMaRt [34]. While this makes view-changes more efficient in terms of communication complexity and message size, as explained in [9, §4.4], HotStuff requires a "locking" phase between the prepare and execution phase to guarantee safety and liveness, where blocks are locked only after they are prepared (for liveness) and before they are executed (for safety). In addition, as the messages sent to extend a block in the prepare phase do not prove the non-existence of a conflicting extension on the same block, it is possible that different subsets of nodes have prepared and locked conflicting extensions of the same block. In that case, only one of those extensions will be accepted (safety), and nodes that locked on a conflicting block have to be allowed to replace it (liveness). To achieve this, a node accepts a block proposed by a leader if it extends the latest block it locked (necessary for safety), or if it extends a block that was prepared at a view higher than the view of the latest block it locked (necessary for liveness)—the conjunction of those two predicates is referred to as the SafeNode predicate [9].

Liveness HotStuff guarantees liveness after GST, the Global Stabilization Time (see §5). This is done through a mechanism that allows correct nodes to eventually be in the same view for long enough to make progress. This mechanism can, for example, be an exponential backoff mechanism where nodes set a timer when starting a new view; exponentially increase their timeout when a time out occurs; and cancel the timer and linearly decrease the timeout when views succeed. In addition, a rotating leader election scheme ensures that there is a single leader per view, and that a correct leader is always eventually elected. HotStuff also guarantees responsiveness – after GST, with a correct leader, nodes make progress by waiting for $N-f$ messages.

4 Using Trusted Components

4.1 Traditional Use of Trusted Services

Previous efforts in the literature presented BFT protocols that increased the resilience of PBFT using increasingly simpler trust assumptions starting from tamperproof distributed components connected through dedicated channels separated from the payload network [16], moving to trusted append-only logs [25], and finally reaching the minimal trusted counters [26, 18]. In this sense, a reader might wonder whether the simplest established trusted components, i.e., a single trusted counter, can be used to increase the resilience of streamlined protocols. We answer this question by the negative in the following.

First let us recall how these simple trusted counters are used. On each client request the leader calls its trusted counter, which generates a unique number along with a signed certificate that proves that the number is assigned to that request. The other nodes go through a similar process when replying to a message from the leader. This way, all correct replicas can safely consider that all messages with a given identifier are identical. Modifying PBFT to allow each node to leverage a trusted counter increases the resilience of PBFT from $\lfloor \frac{N-1}{3} \rfloor$ to $\lfloor \frac{N-1}{2} \rfloor$, where N is the total number of nodes, and reduces the number of phases from 3 to 2. This is achieved as new-view messages contain view-change certificates that require even Byzantine nodes to include the history of messages they have prepared. However, contrary to PBFT, HotStuff does not need or use view-change certificates. It is up to the new leader to chose which prepared block to extend. This prepared block will have a certificate, but a Byzantine leader could produce an old certificate, and the backups would not have a way to verify whether the leader correctly picked the latest prepared block. A simple trusted message counter is therefore not sufficient to lower HotStuff's number of replicas from $3f+1$ to $2f+1$, i.e., to increase its resilience, let alone to reduce its number of phases.

Let us consider a version of HotStuff with $2f+1$ replicas only and the same number of phases as basic HotStuff, where each node is augmented with a trusted message counter. We demonstrate the following non-safe scenario, where $f = 1$, with nodes i, j, k. Therefore, votes from 2 nodes are enough to execute a block. Assume that node i is Byzantine. All nodes initialize their counters to 0. During the first view (view 0), helped by node i, node j executes some block b, which extends the genesis block. Assume that j is the leader of that first view. Therefore, i's counter is now equal to 4 because it has sent 4 messages (new-view, prepare, pre-commit, commit), while j's counter is now 5 because it has sent 5 messages (new-view, prepare, pre-commit, commit, decide). Assume that k is lagging behind and did not receive b. Its counter is still 0. Assume that node i is the leader of the next view (view 1). It receives new-view messages from nodes j and k, and only one of those messages is necessary. It will here use k's vote. Node i might decide to extend the genesis block with some block b', which conflicts with block b, instead of extending b, which is unsafe. Nodes i and k might then execute b'. Node i's counter is then equal to 9 (i.e., 4 + the 5 new increments), while k's counter is now equal to 4. The problem is that the first 4 messages sent by i were sent to j (view 0's leader) and not to k. So k cannot be expected to receive those messages. Therefore, when it receives messages in view 1 from i, starting with a counter value of 4, then even though it has not received the messages with lower counter values (which were meant to j and not k), it should still act upon them. Node k will then not be able to detect that block b was locked and even executed in view 0. This is due to the message pattern used by such protocols. We solve this problem in our protocol by storing some information about prepared/locked blocks in trusted components, while the message pattern remains the same.

To this end, our approach relies on augmenting trusted counters with additional secure storage to guarantee liveness and safety, which later leads to our CHECKER service. In order to increase the resilience, the additional secure storage would need to persist both prepared and locked blocks. The latter are needed as otherwise a Byzantine node could lead f correct but late nodes to lock and then further execute conflicting blocks. However, using only this simple yet additional storage is not enough to reduce the number of communication rounds but merely to increase resilience.

In order to also reduce the number of communication rounds (for better latency) a leader should certify that it has selected the block with the highest view from the new-view messages it has received. This functionality is captured by our ACCUMULATOR service, which forces a leader to chose the latest prepared block among the received ones. This is valid with quorums of size f+1, as using the CHECKER nodes are forced to relay the latest prepared block (as they store it). Interestingly, when using this ACCUMULATOR function, the original suggested trusted counter with augmented storage (to store prepared and locked blocks) becomes slightly simpler: there is no need to store the locked blocks anymore, leading to our current CHECKER.

Therefore, our approach relies on CHECKER and ACCUMULATOR as two local trusted components, which must be available at all nodes, to achieve respectively two goals: (1) increased Byzantine resilience, and (2) reduced latency. Each component is identified by a unique identifier stored with the component. Both components perform key operations, and produce *certificates* to guarantee that those operations have been performed. These certificates are signed messages, which are produced and verified using *private/public keys* stored within the services (§5 provides further details on signatures and §6 shows the certificates used by Damysus). Private keys need to be kept confidential to prevent hosts of trusted components from forging commitments.

4.2 Our CHECKER and ACCUMULATOR Services

4.2.1 CHECKER.

The CHECKER trusted service is used by Damysus for increased Byzantine resilience, allowing to tolerate more Byzantine nodes. In each execution of Damysus, all replicas interact with their local CHECKER. The intuition behind CHECKER is best conveyed through the two main purposes it serves: (1) it assigns to each message a unique identifier thanks to a monotonic counter, which is incremented each time a message is signed, to prevent nodes from equivocating; and (2) it stores information about relevant blocks, namely prepared and locked blocks, to guarantee that nodes cannot lie about the blocks they have last prepared and locked.

Thus, a Byzantine leader cannot send multiple valid proposals in a given view but rather only a single valid, since CHECKER maintains a monotonic counter whose value is equal to the view number and it stamps the proposal of a leader with the signed value of that monotonic counter. Also when prompted by the algorithm to report the last prepared/locked block, Byzantine node can no longer lie about that value, e.g., by reporting an old block, since CHECKER saves the values of the last prepared/locked block and a node is forced to get that value signed from CHECKER when reporting it to the rest of the network.

To this end, CHECKER maintains the following state, which needs to be securely stored in memory, and can only be modified by the service:

- a *monotonically increasing counter* keeping track of the current view and phase
- some information regarding the *latest prepared and locked blocks*. As explained in §6.3, for the purpose of Damysus, we store a view number and a hash value per prepared and locked block.

Furthermore, it provides the following interface:

- TEEprepare(*block*, *certificate*): this function takes a block b (we see below that blocks can often be replaced by their hash values) and a certificate, which includes a block b', and a proof that b' is the latest prepared block according to the node that generated the certificate. It then returns a new certificate that contains the current value of the monotonic counter, certifying that b is a safe (w.r.t. SAFENODE) proposal extending b', after which this counter is increased.
- TEEstore(*certificate*): this function takes a certificate that some block b extends some other block b' as generated by the CHECKER service, verifies it, stores b as the latest prepared or locked block in the state, depending on the phase, and generates a new certificate that this operation has been executed.
- TEEsign(): this function generates a certificate for the stored latest prepared block.

4.2.2 Accumulator.

The accumulator trusted service is used by Damysus to reduce latency by reducing the number of communication phases. In each execution of Damysus, only the leader interacts with its accumulator. Its goal is to certify that some block has the highest view among a given set of blocks. This is used in Damysus to ensure that a leader at anytime proposes the latest prepared block in the system (if it exists).

In a nutshell the intuition behind accumulator is as follows: In a given view, a leader in streamlined protocols typically asks replicas for their latest prepared blocks. Upon receiving them the leader selects the block with the highest view and disseminates it back to the rest of the network. The purpose of accumulator is to prevent a Byzantine leader from deviating, e.g., by not sending the prepared block with the actual highest view. accumulator prevents this malicious act by processing itself the recived messages and generating a signed response that contains the latest prepared block with the highest view. Hence replicas would only process valid responses produced by accumulator.

Similar to checker, accumulator also generates certificates, however, the ones it generates are not tied to a monotonic counter, but instead indicate how many messages have been accumulated so far. We therefore call such a certificate an *accumulator* here.

This service does not require further storage, and provides the following interface:

- TEEstart(*certificate*): this function takes a certificate that some block b is the latest prepared block (see above), and generates an initial *accumulator*, indicating that only one block, namely b, was used so far to generate that accumulator.
- TEEaccum(*accumulator*, *certificate*): this function takes an *accumulator* certifying that some block b has the highest view v among k blocks signed by k different nodes denoted K here, as well as a certificate signed by j that some block b' is the latest prepared block, prepared in view v', and generates a new accumulator certifying that b has the highest view v among k+1 blocks if $v \geq v'$ and $j \notin K$.
- TEEfinalize(*accumulator*). The accumulators considered so far have to keep track of the nodes that have signed the certificates used to generate those accumulators. Once a given *threshold* has been reached, TEEfinalize replaces the set of node ids $\{i_i, \dots, i_k\}$ stored in such an accumulator by their number k

This service is intended to be used by leaders as follows. (1) Before calling the service, collect C, a list of certificates signed by different nodes, and select from C the certificate c with the highest view (here the view at which the block certified by that certificate was prepared). (2) Call TEEstart(c) to create an initial accumulator. (3) Iterate TEEaccum on the remaining certificates in C, expanding the accumulator with the ids of the nodes that signed those certificates. (4) Call TEEfinalize on that accumulator.

4.2.3 On the Use of checker and accumulator.

For space limitations, we use in this paper the checker and accumulator combined, since their combination leads to the most efficient hybrid protocol. However, we point the reader to Appx.C and D for the individual hybrid protocols that are based solely on either the checker (to increase resilience) or the accumulator service (to decrease latency). Remark that when used in combination with the accumulator service (§4.2.2), the checker service does not require to store locked blocks (leading to simpler TEEprepare and TEEstore functions) because leaders are constrained to extend the highest latest prepared block out of a quorum of such messages received in new-view messages.

5 System Model

Replicas and leader We consider a static system consisting of N replicas out of which at most f can be faulty (i.e., Byzantine). Depending on the protocol, N will either be $3f$+1 or $2f$+1. For simplicity, we refer to a replica by using a unique id. We assume that each view has a unique leader, which is chosen deterministically and known to all nodes.

Trusted components Each replica executes trusted components that provide the checker and accumulator services, resulting in a *hybrid* fault model, where at each faulty node all components can be tampered with except the ones providing these trusted services.

Communications We assume that replicas communicate by exchanging messages over a fully connected communication network. Communications are reliable (i.e., messages are not lost). We adopt the partial synchrony model, where there is a known bound Δ and an unknown Global Stabilization Time (GST), such that after GST, all communications arrive within time Δ [35].

Signatures Replicas and trusted components rely on an asymmetric signature scheme. A digital signature σ is generated via the SIGN function, and verified using the VERIFY function. We assume that a digital signature contains the identity of the signing replica or component, which is obtained using σ.id.

Blocks A block b contains transactions submitted by clients. Our protocols work at the block level, and we therefore leave abstract the internal details of transactions, which are mostly application specific. We assume a cryptographic secure hash function $\mathbf{H}$ that is used to hash blocks. We write h for the hash value of a block. We write $b \succ h$ when b is a direct extension of a block with hash value h. We also write $b_1 \succ b_2$ for $b_1 \succ \mathbf{H}(b_2)$. The relation $\succ$ can easily be checked if, e.g., blocks store the hash values of the blocks they extend. We write $\succ^+$ for its transitive closure, and $\succ^*$ for its reflexive and transitive closure. When $b_2 \succ^* b_1$, we say that b_2 is a *descendant* of b_1 and that b_1 is an *ancestor* of b_2. We say that a block b_1 *conflicts with* a different block b_2 if $\neg b_2 \succ^{+} b_1$ and $\neg b_1 \succ^{+} b_2$. We also assume a createLeaf function that creates a new block extending a parent block (or simply its hash value) with client transactions. Let G stand for the genesis block.

6 Damysus

This section presents Damysus, a 2-phase protocol[5] that tolerates $\lfloor \frac{N-1}{2} \rfloor$ Byzantine replicas (f out of $2f$+1 replicas). Damysus requires each node to be equipped with an instance of each of our trusted services described in §4 to achieve respectively two goals: (1) an instance of the CHECKER service for increased resilience against Byzantine nodes; and (2) an instance of the ACCUMULATOR service for reduced latency.

We next present an overview of Damysus and discuss how the CHECKER and ACCUMULATOR components are used.

6.1 Overview

At the beginning of every view its leader awaits to receive f+1 new-view messages from other processes. A new-view message from process p contains a quorum certificate of the latest prepared block at p. A leader then selects from these new-view messages, the certificate c for the prepared block with the highest view (this view being the view at which the block was prepared), and submits it to its ACCUMULATOR component, along with the other f new-view messages. If c was indeed a certificate for the highest prepared blocks among the f+1 submitted certificates, and those f+1 certificates have been produced by distinct nodes, then ACCUMULATOR generates a certificate of its own (a signed message), called an *accumulator* here, guaranteeing that the above has been checked. A leader then uses this *accumulator* to build its proposal. A leader's proposal is first submitted to its local CHECKER component, which validates a single proposal per view, and then it is sent out to the other replicas in a prepare message. All other nodes only accept proposals validated by the current leader's CHECKER.

Upon receiving the leader's proposal b, every replica uses a modified SAFENODE predicate to check whether it should accept b. This SAFENODE checks that b extends the prepared block certified by the ACCUMULATOR. If the leader's proposal is accepted, the replicas also use their CHECKER components to verify the *accumulator*, and in turn sign the proposal. Those signatures are partially signed prepare votes, which the replicas send to the leader.

When the leader receives f+1 prepare votes for its proposal, it combines them into a certificate, which it broadcasts in the pre-commit phase. Upon receipt of a prepare certificate for b from the leader, the nodes use their CHECKER components to store the view at which b is prepared, i.e., the current view, and for code simplicity b's hash value too. Storing only views would require trusted components to perform further computations to check the "correctness" of those blocks, which can be omitted by simply storing the hash values of those blocks. Prepared blocks are stored in CHECKER to ensure that nodes (even Byzantine ones) relay their prepared messages in new-view messages. Once a CHECKER has stored a prepared block, it emits a signature guaranteeing that this operation has been done. Such a signature is a partially signed pre-commit vote, which is sent to the leader

When the leader receives f+1 pre-commit votes, it combines them into a certificate and sends it in a decide message to all other replicas. Upon receiving a decide message, a replica executes the transactions, increments its view number, and starts the next view.

6.2 Definitions

In what follows, we introduce some definitions that we use later to formally describe Damysus.

Phases Messages generated by trusted components include a tag $ph \in \{\texttt{nv\_p}, \texttt{prep\_p}, \texttt{pcom\_p}\}$ indicating the phase in which they were generated: Those are shorthands for new-view, prepare, and pre-commit respectively.

Steps A step (v, ph) is a pair of a view v and a phase ph. To progress through views and phases, nodes increment steps as follows: $(v, \texttt{nv\_p})$++ $= (v, \texttt{prep\_p})$, $(v, \texttt{prep\_p})$++ $= (v, \texttt{pcom\_p})$, and $(v, \texttt{pcom\_p})$++ $= (v{+}1, \texttt{nv\_p})$.

Commitments A commitment ϕ (generated by CHECKER components) is of the form $\langle h, v, h', v', ph, \vec{\sigma}^n \rangle$, where h and h' are hash values of blocks, v and v' are view numbers, ph is a phase, and $\vec{\sigma}^n = [\sigma_1, \ldots, \sigma_n]$ is a list of signatures on data (h, v, h', v', ph). We simply write $\vec{\sigma}$ when the number of signatures in the list is irrelevant.

We refer to h, v as the "proposed (hashed) block/view" of the commitment, and h', v' as the "justification (hashed) block/view" of the commitment.

We sometimes refer to a commitment as an n-commitment if it contains n signatures. In the case of a 1-commitment, we write $\langle h, v, h', v', ph, \sigma \rangle$ for $\langle h, v, h', v', ph, [\sigma] \rangle$. We define $\texttt{C-combine}([\langle h, v, h', v', ph, \sigma_1 \rangle, \ldots, \langle h, v, h', v', ph, \sigma_n \rangle])$ to be $\langle h, v, h', v', ph, [\sigma_1, \ldots, \sigma_n] \rangle$. This is used to create quorum certificates out of partial votes. Given a list $\vec{\phi}$ of the form $[\langle h_1, v_1, h'_1, v'_1, ph_1, \sigma_1 \rangle, \ldots, \langle h_n, v_n, h'_n, v'_n ph_n, \sigma_n \rangle]$, we define the following operation to check whether enough messages of a certain kind have been received: let $\texttt{C-match}(\vec{\phi}, k, h, v, ph)$ be true iff: (1) $n = k$; (2) all n signatures have been created by different nodes; and (3) $\forall i \in \{1, \ldots, n\}. h = h_i \wedge v = v_i \wedge ph = ph_i$. Because not all fields are always necessary in commitments, we use $\perp$ to indicate that a field (more precisely a hash value or view field) is not filled out. The symbol $\perp$ is considered syntactically different from all hash values and view numbers. Given a commitment ϕ of the form $\langle h, v, h', v', ph, \vec{\sigma}^n \rangle$, let $\phi_{\texttt{.Hprep}}$ be h; $\phi_{\texttt{.Vprep}}$ be v; $\phi_{\texttt{.Hjust}}$ be h'; $\phi_{\texttt{.Vjust}}$ be v; $\phi_{\texttt{.phase}}$ be ph; and $\phi_{\texttt{.sign}}$ be $\vec{\sigma}^n$.

[5] As mentioned in §3, HotStuff is a 3-phase protocol, as it has 3 *core* phases. Similarly, Damysus is referred to as a 2-phase protocol as it has 2 core phases.

Accumulator An accumulator *acc* (generated by the ACCUMULATOR service) is built out of CHECKER-generated commitments and has one of the following forms: (1) $\langle v, v', h, \vec{i}, \sigma \rangle$, where h is the hash value of a block prepared at view v', and $\vec{i}$ is a list of node ids that have prepared blocks at most as high as h, including a node that has prepared h at view v'; (2) $\langle v, v', h, n, \sigma \rangle$, indicating that h is the hash value of the highest prepared block among n (a natural number) checked certificates used to generate the accumulator. Let $\vec{i}_1 @ \vec{i}_2$ be the list $\vec{i}_2$ appended to the list $\vec{i}_1$. Let $acc_{\texttt{.hash}}$ be h. Let the size of an accumulator be defined as follows: $|\langle v, v', h, \vec{i}, \sigma \rangle| = |\vec{i}|$, and $|\langle v, v', h, n, \sigma \rangle| = n$, i.e., the size of an accumulator *acc* is the number of nodes that have contributed the commitments that were used to generate *acc*.

Message Message exchanged by nodes are either of the the form ϕ or $\langle b, acc, \sigma \rangle$.

6.3 Instances of the Trusted Services

We now describe the instances of our CHECKER and ACCUMULATOR trusted services (presented in §4) used by `Damysus`.

CHECKER instance As mentioned above `Damysus`'s instance of the CHECKER trusted service stores both the view and hash value of the latest prepared block. Also, for convenience, the monotonic counter is split into a view and a phase. More precisely, the CHECKER component maintains a state that comprises the following (where the public keys are those of the other checker components in the system):

prepv	the view of the last prepared block
preph	the hash value of the last prepared block
view	the current view
phase	the current phase
$priv_c$	the trusted component's private key
pubs	public keys

The *certificates* produced by the CHECKER service are all 1-commitments, and the ones they act upon are accumulators for `TEEprepare`, and $(f{+}1)$-commitments for `TEEstore`.

`TEEsign` creates a 1-commitment using the prepared block stored in the trusted component. Replicas are required to use this function to generate new-view certificates so that they cannot lie about the last block they have prepared. It can also be used by late nodes to increase their view and phase number without going through all the steps. As no phase number is checked when calling `TEEsign`, note that crucially, this function generates a commitment ϕ where the proposed hash value $\phi_{\texttt{.Hprep}}$ is $\perp$ so that ϕ can only be used as a commitment for a `nv_p` phase.

`TEEprepare` takes the pair of the hash value h of a block proposed by a leader and an accumulator (generated by the leader) corresponding to the justification of that block. If the accumulator was generated by the current leader, i.e., for the current, view then `TEEprepare` generates a 1-commitment, which stands for a partially signed prepare vote.

`TEEstore` takes a $(f{+}1)$-commitment, and checks whether that commitment is for a block that has been prepared in the current view by a quorum of nodes, in which case a 1-commitment is generated, which stands for a partially signed pre-commit vote.

ACCUMULATOR instance The ACCUMULATOR component maintains a private key, as well as public keys, which are shared with those of the CHECKER component.

The `TEEstart` function takes a 1-commitment ϕ, and turns it into an accumulator *acc* if ϕ's signature can be verified, in particular registering in *acc* the id of the node that signed ϕ. This initial commitment is meant to be the one for the prepared block with the highest view number among a collection of $f{+}1$ such 1-commitments received by the leader at the beginning of the prepare phase. The leader then iterates over the remaining f commitments, each time calling `TEEaccum` to check that those commitments are for lower view numbers and signed by different nodes, and updating *acc* accordingly, i.e., recording the ids of the f signers. Finally, `TEEfinalize` is called to turn *acc* of the form $\langle v, v', h, [i_1, \ldots, i_{f+1}], \sigma \rangle$ into an acc' of the form $\langle v, v', h, f{+}1, \sigma \rangle$.

6.4 The `Damysus` Algorithm

Fig. 2 presents `Damysus`'s pseudocode. In the `prepare` phase, the leader accumulates $f{+}1$ new-view messages by calling `accumList` (see Fig. 2a, ln. 7), i.e., the leader selects the new-view message for the block prepared at the highest view among all $f{+}1$ received messages, and certifies that it is indeed the highest using its ACCUMULATOR. The leader then extends that highest prepared block with a new block b (Fig. 2a, ln. 8), and prepares b using `TEEprepare` (Fig. 2a, ln. 9) of the CHECKER. This guarantees that the algorithm advances to the next step, and generates as well a signature of the newly proposed block (its hash value), which stands for a partially signed prepare vote.

The leader then sends the new proposal b, along with *acc*, the accumulator it has generated and used to create b, and the signature of its prepared commitment, to all other nodes. Then, the other nodes check that b extends the block contained in *acc* (Fig. 2a, ln. 16), and also prepare the block proposed by the leader using their CHECKER (Fig. 2a, ln. 17).

In the next `pre-commit` phase, correct nodes call `TEEstore` to store b in their CHECKER component. Nodes are forced to store b during this phase to generate a partially signed pre-commit vote. This guarantees that if a block is executed, $f{+}1$ nodes (possibly Byzantine) have stored it in their CHECKER components, and will relay it in their new-view messages.

Thanks to the ACCUMULATOR, nodes do not need to lock blocks as done in HotStuff: no `commit` phase is needed and in the next `decide` phase, the leader gathers $f{+}1$ `pre-commit` commitments, and sends a `pre-commit` $(f{+}1)$-commitment to all nodes, which use it to execute b. No locking is needed because the ACCUMULATOR service guarantees that leaders only propose blocks that extend the highest prepared block.

Figure 2 Damysus ($2f$+1 replicas and 2 phases)

(a) Non-trusted code at replica i

```
1:  $view = 0$ // current view (duplicates the one in the TEE)
2:  $pubs$ // public keys (duplicate the ones in the TEE)
3:
4:  // prepare phase
5:  as a leader
6:      wait for $\vec{\phi}$ s.t. C-match$(\vec{\phi}, f{+}1, \bot, view, $nv_p$)$
7:      $acc :=$ accumList$(\vec{\phi})$
8:      $b :=$ createLeaf$(acc.$hash$,$ client transactions$)$
9:      $\phi_{prep} :=$ TEEprepare$(\mathbf{H}(b), acc)$ // also sent to itself
10:     send $\langle b, acc, \phi_{prep}.$sign$\rangle$ to backups
11:
12: as a backup
13:     wait for $\langle b, \langle view, v', h', f{+}1, \sigma\rangle, \sigma'\rangle$ from the leader
14:     $acc := \langle view, v', h', f{+}1, \sigma\rangle$
15:     $\phi_{prep} := \langle \mathbf{H}(b), view, h', v', $prep_p$, \sigma'\rangle$
16:     abort if $\neg($VERIFY$(\phi_{prep})_{pubs} \land b > h')$
17:     send $\phi' :=$ TEEprepare$(\mathbf{H}(b), acc)$ to leader
18:
19: // pre-commit phase
20: as a leader
21:     wait for $\vec{\phi}$ s.t. C-match$(\vec{\phi}, f{+}1, h, view, $prep_p$)$
22:     send $\phi :=$ C-combine$(\vec{\phi})$ to all replicas
23:
24: all replicas
25:     wait for $\langle h, view, h', v', $prep_p$, \vec{\sigma}^{f+1}\rangle$ from leader
26:     send $\phi :=$ TEEstore$(\langle h, view, h', v', $prep_p$, \vec{\sigma}^{f+1}\rangle)$ to leader
27:
28: // decide phase
29: as a leader
30:     wait for $\vec{\phi}$ s.t. C-match$(\vec{\phi}, f{+}1, h, view, $com_p$)$
31:     send $\phi :=$ C-combine$(\vec{\phi})$ to all replicas
32:
33: all replicas
34:     wait for $\langle h, view, \bot, \bot, $pcom_p$, \vec{\sigma}^{f+1}\rangle$ from leader
35:     abort if $\neg$VERIFY$(\langle h, view, \bot, \bot, $pcom_p$, \vec{\sigma}^{f+1}\rangle)_{pubs}$
36:     execute $b$ corresponding to $h$ & reply to clients
37:
38: // new-view phase
39: all replicas
40:     when executed or timeout
41:     // call TEEsign until the node can generate
42:     // a new-view-phase commitment for the next view
43:     $(v, ph) := (view, $prep_p$)$; $view$++
44:     while $(v, ph) \neq (view, $nv_p$)$ do
45:         $\phi :=$ TEEsign(); $(v, ph) := (\phi.$vprep$, \phi.$phase$)$
46:     end while
47:     send $\phi$ to $view$'s leader
48:
49: function accumList$(\vec{\phi})$
50:     $\phi_0 :=$ commitment $\phi \in \vec{\phi}$ with highest $\phi.$vjust
51:     $acc :=$ TEEstart$(\phi_0)$
52:     for $\phi \in \vec{\phi} \setminus \{\phi_0\}$ do $acc :=$ TEEaccum$(acc, \phi)$
53:     return TEEfinalize$(acc)$
```

(b) TEE code

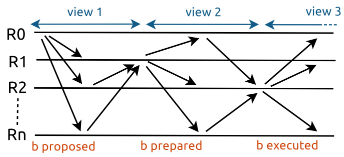
```
6:  $(prepv, preph) = (0, \mathbf{H}(G))$ // latest prepared block
7:  $(view, phase) = (0, $nv_p$)$ // current step
8:  $priv_c, pubs$ // private (confidential) and public keys
9:
10: function createUniqueSign$(h, h', v')$
11:     $\phi := \langle h, view, h', v', phase, \sigma\rangle$ where $\sigma :=$ SIGN$(h, view, h', v', phase)_{priv_c}$
12:     $(view, phase)$++ // increased to avoid equivocation
13:     return $\phi$
14:
15: function TEEsign()
16:     return $\phi :=$ createUniqueSign$(\bot, preph, prepv)$
17:
18: function TEEprepare$(h, acc)$ where $acc$ is $\langle v, v', h', f{+}1, \sigma\rangle$
19:     if VERIFY$(acc)_{pubs} \land view{=}v \land h \neq \bot$ then
20:         return $\phi :=$ createUniqueSign$(h, h', v')$
21:     end if
22:
23: function TEEstore$(\phi)$ where $\phi$ is $\langle h, v, h', v', ph, \vec{\sigma}^{f+1}\rangle$
24:     if VERIFY$(\phi)_{pubs} \land view = v \land ph = $prep_p then
25:         $preph := h$; $prepv := v$
26:         return $\phi' :=$ createUniqueSign$(h, \bot, \bot)$
27:     end if
28:
29: function TEEstart$(\phi)$ where $\phi$ is $\langle \bot, v, h', v', $nv_p$, \sigma\rangle$
30:     if VERIFY$(\phi)_{pubs}$ then
31:         $\sigma' :=$ SIGN$(v, v', h', [\sigma.$id$])_{priv_c}$
32:         return $acc := \langle v, v', h', [\sigma.$id$], \sigma'\rangle)$
33:     end if
34:
35:  function TEEaccum$(acc, \phi)$ where $acc$ is $\langle v_1, v'_1, h_1, \vec{i}, \sigma\rangle$
                                and $\phi$ is $\langle \bot, v_2, h_2, v'_2, $nv_p$, \sigma_2\rangle$
36:     if $\left(\begin{array}{l} v_1{=}v_2 \land v'_1 \geq v'_2 \land \sigma_2.\text{id} \notin \vec{i} \\ \land \text{VERIFY}(acc)_{pubs} \land \text{VERIFY}(\phi)_{pubs} \end{array}\right)$ then
37:         $\sigma' :=$ SIGN$(v_1, v'_1, h_1, \vec{i}$@$[\sigma_2.$id$])_{priv_c}$
38:         return $acc' := \langle v_1, v'_1, h_1, \vec{i}$@$[\sigma_2.$id$], \sigma'\rangle$
39:     end if
40:
41: function TEEfinalize$(acc)$ where $acc$ is $\langle v, v', h, \vec{i}, \sigma\rangle$
42:     if VERIFY$(acc)_{pubs}$ then
43:         return $acc' := \langle v, v', h, |\vec{i}|, \sigma\rangle$ where $\sigma :=$ SIGN$(v, v', h, |\vec{i}|)_{priv_c}$
44:     end if
```

Nodes rely on timers, which they start at the beginning of each view, to move on to the next view when the current one does not succeed fast enough, for example, due to a faulty leader. The new-view phase, is then executed either once the current view has succeeded, i.e., the decide phase has completed, or the timer started at the beginning of the view expired. In that phase, nodes increment their view, and submit their vote to the leader of this new view. As nodes store the current view number both outside and within their trusted component, during this phase nodes ensure that the view is incremented both outside (Fig. 2a, ln. 43) and within the trusted component (Fig. 2a, ll. 44-45).

6.5 Proof of Correctness - Overview

We provide our security discussion here and leave the formal verification as future work. Existing effort to formally verify HotStuff by using model checking (such as TLA+/TLC [36])

Figure 3 Communication pattern in `Chained-Damysus`.



or automated theorem proving (such as Ivy and TLAPS [37]) can potentially be adapted to formally verify Damysus.

The intuition behind `Damysus`'s correctness stems from the following two points (see Appx. A for a detailed proof).

(1) The proof relies on demonstrating that a Byzantine leader cannot propose various valid proposals for a single view, or even alter what has been proposed over the course of the different phases within a single view. This is guaranteed by the CHECKER that only certifies one proposed value as valid per view (all messages are assigned a unique counter/step using `createUniqueSign`). All other nodes only accept valid messages certified by the CHECKER. Hence any vote in any phase of a given view is on the same proposal.

In addition, prepared blocks are also stored in CHECKER for both liveness and safety, otherwise leaders might only propose blocks based on f+1 new-view messages from late or Byzantine nodes. `TEEsign` in particular enforces that nodes, even Byzantine ones, send their latest prepared blocks.

(2) In basic HotStuff, every node locks a block b locally once it knows that it has been prepared by a quorum of nodes so that it will later accept another block b' only if it extends b or if b' extends a block that was prepared for a higher view than b's (possibly because b did not get executed, but the system still managed to make some progress on a conflicting block). Without locking, nodes would be allowed to accept any new block, even blocks that are in direct conflict (at the next view, but not extending) with the ones they have executed. Thanks to the ACCUMULATOR, locking is unnecessary because when receiving a proposal from the leader with a valid accumulator, nodes are then guaranteed, that this proposal must extend the highest prepared block, and therefore, for example, the block that they have last executed.

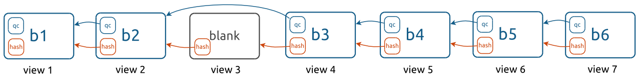
7 Chained-Damysus

This section presents `Chained-Damysus`, which concurrently processes blocks to further improve performance.

7.1 Algorithm

`Chained-Damysus` improves on `Damysus` by allowing nodes to pre-commit the predecessor block b_0 of a block b when preparing b, while at the same time deciding the predecessor b_1 of b_0 (`Damysus` does not have a commit phase). Fig. 3 illustrates `Chained-Damysus`'s communication pattern, while Fig. 5 provides its pseudocode. The leader R_0 of view 1 extends the genesis block with a new block b, which is proposed to all nodes in this view 1. All nodes vote for b by sending their votes to the leader R_1 of view 2. In case R_1 receives f+1 votes (prepare messages) for b, those votes form a new certificate for block b from view 1. The leader R_1 of view 2 can then use this certificate when creating a new block b'. In that case, R_1 does not need to call its trusted accumulator (the condition ln. 8 in Fig. 5a is then false) as it has a certificate from the previous view, and does not need to justify that choice of certificate. However, in case R_1 does not receive f+1 votes for b, it needs to obtain the latest prepare certificate. To achieve this, nodes always send new-view messages along with their prepare messages (ll. 18, 19, 32, and 33 in Fig. 5a).[6] Once R_1 has collected f+1 new-view messages, it selects the latest prepare certificate, and calls its trusted accumulator to certify that the latest prepare certificate was indeed selected. Once again, R_1 makes use of this certificate when creating a new block b'. In both cases, R_1 sends its new block to the other nodes. This process is then reiterated in the subsequent views.

Figure 4 Example of a chain of blocks with a blank block.



In this chained (or pipelined) version, while some other block b' is being proposed in view 2, b is being prepared, and while some other block is being proposed in view 3, b is executed and b' is being prepared. `Chained-Damysus`, like `Damysus`, requires one phase less than chained HotStuff and has a better resilience: it tolerates less that half of the system being corrupt (compared to one-third).

We now detail how we adapt some of the concepts introduced previously to handle a chained version.

Blocks As in chained HotStuff [9, §5], we now replace `createLeaf` by `createChain`, which generates blank blocks to fill gaps in the chain. Precisely, the leader of view v that builds a proposal extending a block proposed in view v' invokes `createChain` to create blank blocks for all views in $[v'+1, v-1]$. We assume that a block stores: (1) as before, a pointer to its parent block (a hash value), accessible using b`.parent`; (2) its justification accessible using b`.just`, which is a certificate as defined below.

Fig. 4 depicts a chain with one blank block between b_2 and b_3, where the top blue arrows capture b`.just`, while the bottom orange arrows capture b`.parent`.

In chained HotStuff, the oldest block in a chain of 4 consecutive blocks without blanks gets executed. In the above example, b_3 gets executed in view 7, when b_6 is proposed. As `Chained-Damysus` requires one phase less, only 3 consecutive blocks are necessary for a block to be executed. Therefore, b_3 gets executed in view 6, when b_5 is proposed. This derives from the fact that 2 phases, namely `prepare` and

[6] In the pseudo-code, nodes send two messages for simplicity, while in practice those two messages can be combined.

pre-commit, are required in Damysus before a block gets executed in the decide phase (therefore a total of 3 consecutive phases).

Certificates Certificates (stored within blocks and accessible via $b.\texttt{just}$) are required to be of size f+1, and nodes reject blocks such that $|b.\texttt{just}| \neq f$+1. A certificate is now either a *quorum certificate* or an accumulator. For simplicity, we write *qc* for both. A *quorum certificate qc* is of the form $\langle v, h, \vec{\sigma} \rangle$, where h is the hash value of a block prepared in view v, and $\vec{\sigma}$ is a list of partial signatures on the prepared block at view v. If a certificate *qc* is of the form $\langle view, h, \vec{\sigma}^n \rangle$, then let $qc.\texttt{cview} = qc.\texttt{view} = view$, $qc.\texttt{hash} = h$, and $|qc| = n$. If *qc* is an accumulator of the form $\langle view, v, h, n, \sigma \rangle$, then let $qc.\texttt{cview} = view$, $qc.\texttt{view} = v$, $qc.\texttt{hash} = h$, and $|qc| = n$. Given a certificate *qc*, $qc.\texttt{view}$ refers to the view at which the certificate was created, while $qc.\texttt{cview}$ refers to the view at which $qc.\texttt{hash}$ was certified. For example, an accumulator of the form $\langle v{+}1, v, h, n, \sigma \rangle$ certifies in view v+1 that h was the hash value of the highest block elected in view v.

Nodes store a certificate qc_{prep}, which, when acting as leaders, they update once they obtain a quorum of votes at the end of the previous view, which act as new-view messages. The certificate qc_{prep} is initialized with $\perp$, which is a special certificate for view 0, that does not contain any signature.

Steps Our chained version has only two types of phases: prepare and view-change identified by the tag nv_p and prep_p, respectively. Consequently, the increment function is now defined by $(v, \texttt{prep\_p})\text{++} = (v, \texttt{nv\_p})$ and $(v, \texttt{nv\_p})\text{++} = (v{+}1, \texttt{prep\_p})$. Steps are incremented in a way that guarantees that a commitment generated at a view v is tagged with v−1, regardless of the origin of the commitment, i.e., whether it came from prepare votes or from new-view messages. Due to this step reordering, nodes now start at view 1.

Commitments A commitment ϕ is now either a new-view commitment of the form $\langle \perp, view, h, v, \texttt{nv\_p}, \sigma \rangle$ used by nodes to relay their latest prepared block (its hash value), where *view* is the current view, and h is the hash value of the latest prepared block at view v; or it is a prepare commitment of the form $\langle h, view, \perp, \perp, \texttt{prep\_p}, \vec{\sigma}^n \rangle$ used by nodes to vote on blocks in the prepare phase, where h is the hash value of the block currently being prepared in view *view*. In both cases, we write $\phi.\texttt{view}$ for *view*. In the first case we write $\phi.\texttt{Hcomm}$ for h and $\phi.\texttt{Vcomm}$ for v; and in the second case we write $\phi.\texttt{Hcomm}$ for h and $\phi.\texttt{Vcomm}$ for *view*. Note that $\phi.\texttt{Hcomm}$ is the hash value of a block that received f+1 votes in view $\phi.\texttt{Vcomm}$, while $\phi.\texttt{view}$ is the view in which this information is signed and sent.

7.2 Proof of Correctness - Overview

We prove Chained-Damysus's safety and liveness in Appx. B, and provide here a high-level view of the safety proof, i.e., that correct nodes do not execute conflicting blocks.

A block is executed if it is the oldest in a chain of three consecutive blocks, i.e., if it is certified in three consecutive views. For safety, we must prove that the executed oldest block b_2 of any such chain C_2 extends the executed oldest block b_1 of any other such chain C_1 (assuming b_2's view is higher than b_1's). If the two chains overlap, then they cannot be inconsistent because at most one block can be certified per view, as nodes can only vote once per view. If the two chains do not overlap, then b_2's view is higher than the views of C_1's blocks. We can then derive that b_2 is a descendant of the middle block b'_1 of C_1, and therefore that $b_2 \succ^* b'_1 \succ b_1$. We differentiate two cases: (1) the leaders of the views between C_1 and C_2 have created certificates out of the prepare messages they received, and therefore did not need to call their trusted accumulator, in which case $b_2 \succ^* b'_1$; and (2) a leader L of a view between C_1 and C_2 extended a different certificate than the one created at the end of the previous view. In that case, because leaders (even Byzantine ones) have to justify that they have selected the latest prepare certificate using their trusted accumulators, and because nodes (even Byzantine ones) must relay their latest prepare certificates in new-view messages, we are again guaranteed that L will only be able to accumulate f+1 votes for its proposal if that proposal extends b'_1.

8 Evaluation

Implemented protocols This section evaluates the performance of Damysus (§6) and Chained-Damysus (§7) and respectively compares them to those of basic and chained HotStuff. In order to better quantify the impact of the CHECKER and ACCUMULATOR trusted services on performance we also implement two non-chained additional versions of Damysus. One where nodes are only equipped with a CHECKER, but not an ACCUMULATOR, called Damysus-C, and one where nodes are only equipped with an ACCUMULATOR, but not a CHECKER, called Damysus-A. We recall from §4.2.3 that by using a single component, Damysus-C merely improves the resilience while Damysus-A only reduces the communication phases. We summarize in what follows the protocols we evaluate (C. stands for CHECKER and A. for ACCUMULATOR).

name	nodes	phases	trusted comp.
basic HotStuff	$3f + 1$	3	none
Damysus-C	$2f + 1$	3	C.
Damysus-A	$3f + 1$	2	A.
Damysus	$2f + 1$	2	C. & A.
chained HotStuff	$3f + 1$	3	none
Chained-Damysus	$2f + 1$	2	C. & A.

Development environment All 6 protocols are implemented in C++[7], and whenever applicable use Intel SGX enclaves [17] to run trusted services. We use SGX because of its Linux SDK that provides a convenient development environment. Although our trusted services are generic enough

[7] The code can be found here: https://github.com/vrahli/damysus.

Figure 5 Chained-Damysus ($2f$+1 replicas and 2 phases).

(a) Non-trusted code at replica i

```
$view = 1$ // current view (duplicates the one in the TEE)
$pubs$ // public keys (duplicate the ones in the TEE)
$qc_{prep} := \perp$ // latest prepared certificate
$blocks$ // mapping from views to proposed blocks

// prepare phase
as a leader
    if $qc_{prep}.\mathtt{cview} \neq view-1$ then
        // we don't have the latest certificate
        wait for $\vec{\phi}$ s.t. $\mathtt{C\text{-}match}(\vec{\phi}, f+1, \perp, view-1, \mathtt{nv\_p})$
        $qc_{prep} := \mathtt{accumList}(\vec{\phi})$
    end if
    $b := \mathtt{createChain}(qc_{prep},$ client transactions$)$
    $blocks[view] := b$
    $b_0 := blocks[b.\mathtt{just}.\mathtt{view}]$ if it is defined, else abort
    abort if $\mathbf{H}(blocks[b.\mathtt{just}.\mathtt{view}]) \neq b.\mathtt{just}.\mathtt{hash}$
    $\phi_{prep} := \mathtt{TEEprepare}(b, b_0)$
    send $\langle b, \phi_{prep}.\mathtt{sign} \rangle$ to backups
    send $\phi_{nv} := \mathtt{TEEsign}()$ to leader of view $view$+1

all replicas
    wait for $\langle b, \sigma' \rangle$ from the leader
    abort if $view \neq b.\mathtt{just}.\mathtt{cview}+1$
    $b_0 := blocks[b.\mathtt{just}.\mathtt{view}]$ if it is defined, else abort
    abort if $\mathbf{H}(blocks[b.\mathtt{just}.\mathtt{view}]) \neq b.\mathtt{just}.\mathtt{hash}$
    $b_1 := blocks[b_0.\mathtt{just}.\mathtt{view}]$ if it is defined, else abort
    abort if $\mathbf{H}(blocks[b_0.\mathtt{just}.\mathtt{view}]) \neq b_0.\mathtt{just}.\mathtt{hash}$
    if $\neg$ leader of $view$ then
        $\phi_{prep} := \langle \mathbf{H}(b), view, \perp, \perp, \mathtt{prep\_p}, \sigma' \rangle$
        abort if $\neg(\mathtt{VERIFY}(\phi_{prep})_{pubs} \wedge b \succ^{+} b.\mathtt{just}.\mathtt{hash})$
        $blocks[view] := b$
        send $\phi' := \mathtt{TEEprepare}(b, b_0)$ to leader of view $view$+1
        send $\phi_{nv} := \mathtt{TEEsign}()$ to leader of view $view$+1
    end if
    if $b.\mathtt{parent} = \mathbf{H}(b_0) \wedge b_0.\mathtt{parent} = \mathbf{H}(b_1)$ then
        execute $b_1$ (and previous blocks) & reply to clients
    end if
    if $\neg$ leader of $view$+1 then $view$++

as a leader of next view $view$+1
    wait for $\vec{\phi}$ s.t. $\mathtt{C\text{-}match}(\vec{\phi}, f+1, h, view, \mathtt{prep\_p})$
    $\langle h, view, \perp, \perp, \mathtt{prep\_p}, \vec{\sigma}^{f+1} \rangle := \mathtt{C\text{-}combine}(\vec{\phi})$
    $qc_{prep} := \langle view, h, \vec{\sigma}^{f+1} \rangle$; $view$++

// new-view phase
upon timeout// executed by all replicas
    $(v, ph) := (0, \mathtt{prep\_p})$; $view$++
    while $(v, ph) \neq (view, \mathtt{nv\_p})$ do
        $\phi := \mathtt{TEEsign}()$; $(v, ph) := (\phi.\mathtt{view}, \phi.\mathtt{phase})$
    end while
    send $\phi$ to $view$'s leader

function $\mathtt{accumList}(\vec{\phi})$
    $\phi_0 :=$ message $\phi \in \vec{\phi}$ with highest $\phi.\mathtt{Vcomm}$
    $acc := \mathtt{TEEstart}(\phi_0)$
    for $\phi \in \vec{\phi} \setminus \{\phi_0\}$ do $acc := \mathtt{TEEaccum}(acc, \phi)$
    return $\mathtt{TEEfinalize}(acc)$
```

(b) TEE code

```
$(prepv, preph) = (0, \mathbf{H}(G))$ // latest prepared block
$(view, phase) = (0, \mathtt{nv\_p})$ // current step
$priv_c, pubs$ // private (confidential) and public keys

functions $\mathtt{createUniqueSign}(h, h', v')$, $\mathtt{TEEsign}()$
    // Same as in Fig. 2b

function $\mathtt{TEEprepare}(b, b_0)$
    $qc := b.\mathtt{just}$
    if $\left(\begin{array}{l} \mathtt{VERIFY}(qc)_{pubs} \wedge view{=}qc.\mathtt{cview}+1 \\ \wedge qc.\mathtt{hash} = \mathbf{H}(b_0) \end{array}\right)$ then
        if $b.\mathtt{parent}{=}\mathbf{H}(b_0)$ then
            $preph := qc.\mathtt{hash}$; $prepv := qc.\mathtt{view}$
        end if
        return $\phi' := \mathtt{createUniqueSign}(\mathbf{H}(b), \perp, \perp)$
    end if

functions $\mathtt{TEEstart}(\phi)$, $\mathtt{TEEaccum}(acc, \phi)$, $\mathtt{TEEfinalize}(acc)$
    // Same as in Fig. 2b
```

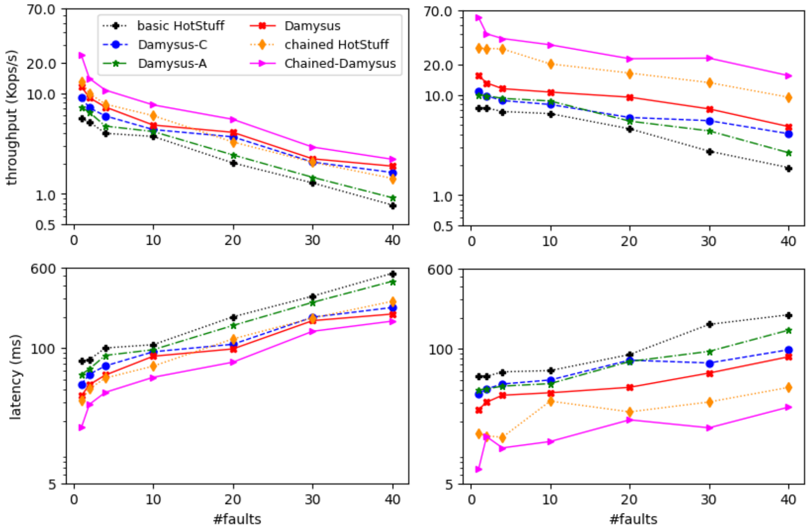
to be potentially implemented in any trusted execution environment, we acknowledge that SGX security vulnerabilities have been described in the literature [38]. Replicas use ECDSA signatures with prime256v1 elliptic curves (available in OpenSSL [39]), and are connected using Salticidae [40].

Note that basic and chained HotStuff do not rely on trusted components, while all the other protocols do and make use of SGX to implement them.

Deployment settings We deploy our protocols on AWS EC2 machines with one t2.micro instance per node instance to reproduce a real world environment. Each figure presents the average of 100 repetitions, where each repetition involves 30 views. We vary the fault threshold $f \in \{1, 2, 4, 10, 20, 30, 40\}$ (i.e. up to 121 nodes) with blocks of 400 transactions, and use payloads of 0B and 256B. In addition to the payload, a transaction contains 2×4B for metadata (a client id, and a transaction id), as well as the hash value of the previous block of size 32B, thereby adding 40B to each transaction in addition to its payload. For example, the experiments with payloads of 0B involve blocks of size 400×40B=15.6KB. Payloads of 0B have been selected to evaluate the protocols' overhead, while payloads of 256B have been selected to observe the trend when increasing the size of blocks. With 256B payloads, blocks are then of size $400 \times (256 + 40)$B=115.6KB, which as shown below allows observing a significant latency increase: about one order of magnitude for our world-wide deployment.

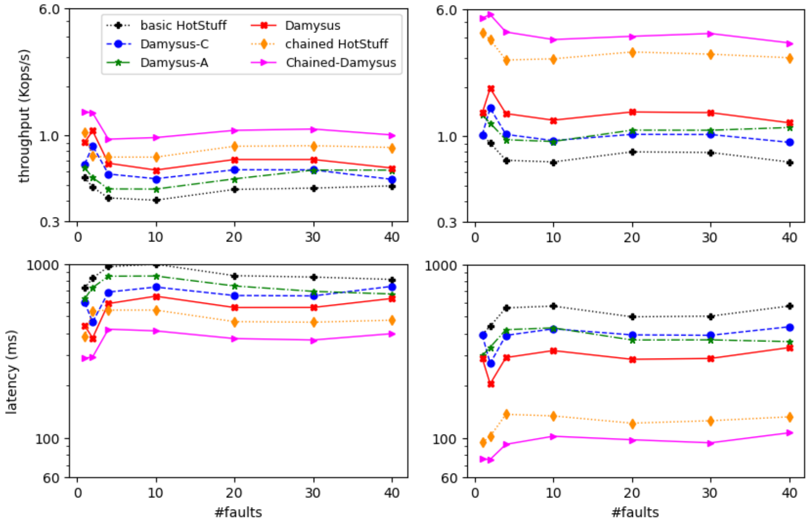
Regional deployment Fig. 6a and 6b show experiment with nodes deployed across 4 regions in Europe (Ireland, London, Paris, and Frankfurt), comparing the throughput and latency (measured by the replicas) of the 6 protocols listed above. For the experiments reported in Fig. 6a, payloads are of size 256B. As we see there, compared to basic HotStuff, Damysus-C has an average throughput increase

Figure 6 Throughput (top Figs., in Kops/s with log scale) and latency (bottom Figs., ms) EU regions, varying the payloads.



(a) EU regs. & 256B pl. (AWS) **(b)** EU regs. & 0B pl. (AWS)

Figure 7 Throughput (top Figs., in Kops/s with log scale) and latency (bottom Figs., ms) World regions, varying the payloads.



(a) World regs. & 256B pl. (AWS) **(b)** World regs. & 0B pl. (AWS)

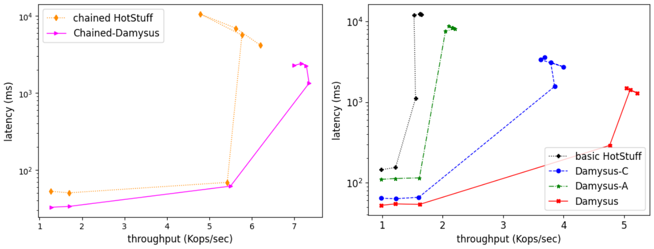
of 59.7%, Damysus-A of 19.3%, and Damysus of 87.5%. Moreover, Damysus-C has an average latency decrease of 35.9%, Damysus-A of 16.6%, and Damysus of 45%. In addition, compared to chained HotStuff, Chained-Damysus has a throughput increase of 50.5% and latency decrease of 32.1%. For the experiments reported in Fig. 6b, payloads are of size 0B. As we see there, compared to basic HotStuff, Damysus-C has an average throughput increase of 54.6%, Damysus-A of 36.7%, and Damysus of 107.1%. Moreover, Damysus-C has an average latency decrease of 31.8%, Damysus-A of 27.4%, and Damysus of 50.6%. In addition, compared to chained HotStuff, Chained-Damysus has a throughput increase of 57.4% and latency decrease of 33.1%. We observe from those experiments that each of the trusted components has a substantial effect on increasing the throughput and decreasing the latency.

Cross continent deployment Fig. 7a and 7b show experiments with nodes deployed across 11 regions across the world (4 in the US in North Virginia, Ohio, North California, and Oregon; 4 in Europe in Ireland, London, Paris, and Frankfurt; 2 in Asia in Singapore, and Sydney; and 1 in Canada Central), comparing the 6 protocols' throughput and latency (measured by the replicas). For the experiments reported in Fig. 7a, payloads of size 256B. We observe that compared to basic HotStuff Damysus-C has an average throughput increase of 35.1%, Damysus-A of 18.4%, and Damysus of 61.6%.

Figure 8 Comparison for $N = 61$ of the throughput/latency improvement over HotStuff and chained HotStuff

	Damysus-C	Damysus-A	Damysus	Chained-Damysus
Fig. 6a	1.88%/0.83%	28.04%/−37.83%	9.88%/8.10%	−10.96%/−18.44%
Fig. 6b	20.59%/17.04%	−4.68%/−7.31%	57.94%/33.70%	40.86%/29.81%
Fig. 7a	31.63%/23.41%	31.25%/18.69%	52.30%/34.26%	27.44%/21.53%
Fig. 7b	27.72%/21.71%	35.63%/26.27%	73.79%/42.36%	29.70%/22.88%

Figure 9 Max. throughput (Kops/s) vs. latency (ms).



Note that Damysus performs better than a simple combination of CHECKER (as in Damysus-C) and ACCUMULATOR (as in Damysus-A), as it eliminates the need of CHECKER storing locked blocks, as explained in §4. Moreover, Damysus-C has an average latency decrease of 24.2%, Damysus-A of 14.0%, and Damysus of 36.6%. In addition, compared to chained HotStuff, Chained-Damysus has a throughput increase of 35.2% and latency decrease of 24.8%. For the experiments reported in Fig. 7b, payloads are of size 0B. As we see there, compared to basic HotStuff, Damysus-C has an average throughput increase of 33.1%, Damysus-A of 38.2%, and Damysus of 78.6%. Moreover, Damysus-C has an average latency decrease of 23.3%, Damysus-A of 27.0%, and Damysus of 43.0%. In addition, compared to chained HotStuff, Chained-Damysus has a throughput increase of 32.2% and latency decrease of 23.7%.

In all cases, the hybrid protocols have a higher throughput and lower latency than the corresponding non-hybrid protocols. The most efficient protocol is always Damysus, which combines the advantages of Damysus-C and Damysus-A. The same conclusions apply to the chained versions.

Comparison with a given N In addition to comparing the protocols for a given number of faults f, we also compare them depending on the number of nodes N. The only comparable values are $f_1 = 20$ for the non-hybrid protocols and $f_2 = 30$ for the hybrid protocols, in which case $3f_1 + 1 = 61 = 2f_2 + 1$. As one can observe in the above experiments, for a system of size $N = 61$, Damysus and Chained-Damysus tolerate more faults than basic and chained HotStuff ($\lfloor \frac{N-1}{2} \rfloor = 30$ compared to $\lfloor \frac{N-1}{3} \rfloor = 20$), and also perform better overall. For example, for $N = 61$, Chained-Damysus and chained HotStuff have comparable throughput and latency in Fig. 6a, while Chained-Damysus has a 23Kops/sec throughput and a 17ms latency compared to chained HotStuff's 16Kops/sec throughput and 24ms latency in Fig. 6b. Fig. 8 summarizes the improvement of the throughput (left value) and latency (right value) of Damysus-C, Damysus-A, and Damysus over HotStuff, and of Chained-Damysus over chained HotStuff.

Throughput vs. latency Fig. 9 evaluates all protocols while increasing the throughput until system saturation. For this experiment, we configure the protocols using $f = 1$, blocks of 400 transactions, 0B payloads, and 4 regions in Europe. We increase the rate at which clients submit transactions, until we saturate the systems. The throughput and latency are here measured by the clients. In the left figure, we evaluate the chained protocols using 10 clients, which send a total of 250K transactions each at the following rates (going left to right in the figure): 700, 500, 100, 50, 10, 5, and $0\mu s$. In the right figure, we evaluate the basic protocols using 6 clients, which send a total of 50K transactions each at the following rates (going left to right in the figure): 900, 700, 500, 100, 50, 10, 5, and $0\mu s$. Note that to the right of the figures the lines move left to right because of small variations resulting from the unstable nature of such systems.

We observe that Chained-Damysus has a lower latency than HotStuff, and a higher maximum throughput. Also, all basic hybrid versions have a lower latency and higher maximum throughput than HotStuff. Damysus performs better than Damysus-C, which performs better than Damysus-A.

9 Conclusion

This paper explored the design space of hybrid solutions that leverage arguably minimalist trusted components to improve the performance and resilience of HotStuff-like protocols. We showcased that the simplest known trusted components in hybrid solutions for traditional BFT protocols are not sufficient to deliver the same guarantees for streamlined BFT protocols. To this end, we introduced two simple trusted components whose services are locally required by every node in the system. Using these trusted components, which we called CHECKER and ACCUMULATOR, we introduced Damysus a hybrid variant of basic HotStuff. Damysus simultaneously improved resilience, reduced communication complexity and latency, and over-performed HotStuff in terms of throughput. Damysus preserves HotStuff's overall behavior to retain linear view change and optimistic responsiveness. We finally extended Damysus to Chained-Damysus, which supports chained operations.

Acknowledgement

This work was partially supported by the Australian Research Council (ARC) under project DE210100019.

References

[1] L. Lamport, R. E. Shostak, and M. C. Pease. 1982. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4, 3, 382–401.

[2] C. Cachin and M. Vukolic. 2017. Blockchain consensus protocols in the wild (keynote talk). In *DISC* (LIPIcs). Volume 91. Schloss Dagstuhl - Leibniz-Zentrum fur Informatik.

[3] C. Natoli, J. Yu, V. Gramoli, and P. J. E. Verissimo. 2019. Deconstructing blockchains: A comprehensive survey on consensus, membership and structure. *CoRR*, abs/1908.08316. arXiv: 1908.08316.

[4] M. Castro and B. Liskov. 1999. Practical byzantine fault tolerance. In *OSDI*. USENIX Association.

[5] E. Shi. 2019. Streamlined blockchains: a simple and elegant approach (a tutorial and survey). In *ASIACRYPT* (LNCS). Volume 11921. Springer.

[6] T.-H. H. Chan, R. Pass, and E. Shi. 2018. Pili: an extremely simple synchronous blockchain. *IACR Cryptol. ePrint Arch.*, 980.

[7] T.-H. H. Chan, R. Pass, and E. Shi. 2018. Pala: A simple partially synchronous blockchain. *IACR Cryptol. ePrint Arch.*, 981.

[8] E. Buchman, J. Kwon, and Z. Milosevic. 2018. The latest gossip on BFT consensus. *CoRR*, abs/1807.04938. arXiv: 1807.04938.

[9] M. Yin, D. Malkhi, M. K. Reiter, G. Golan-Gueta, and I. Abraham. 2019. Hotstuff: BFT consensus with linearity and responsiveness. In *PODC*. ACM.

[10] B. Y. Chan and E. Shi. 2020. Streamlet: textbook streamlined blockchains. In *AFT*. ACM.

[11] T. D. Chandra and S. Toueg. 1996. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43, 2, 225–267.

[12] Y. Mao, F. P. Junqueira, and K. Marzullo. 2008. Mencius: building efficient replicated state machine for wans. In *OSDI*. USENIX Association.

[13] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung. 2009. Spin one's wheels? byzantine fault tolerance with a spinning primary. In *SRDS*. IEEE Computer Society.

[14] A. Clement, E. L. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. 2009. Making byzantine fault tolerant systems tolerate byzantine faults. In *NSDI*. USENIX Association.

[15] G. S. Veronese. 2010. *Intrusion Tolerance in Large Scale Networks*. PhD thesis. Universidade de Lisboa.

[16] M. Correia, N. F. Neves, and P. Veríssimo. 2004. How to tolerate half less one byzantine nodes in practical distributed systems. In *SRDS*. IEEE Computer Society.

[17] [n. d.] SGX. https://software.intel.com/en-us/sgx.

[18] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Veríssimo. 2013. Efficient byzantine fault-tolerance. *IEEE Trans. Computers*, 62, 1, 16–30.

[19] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel. 2012. Cheapbft: resource-efficient byzantine fault tolerance. In *EuroSys*. ACM.

[20] J. Behl, T. Distler, and R. Kapitza. 2017. Hybrids on steroids: SGX-based high performance BFT. In *EuroSys*. ACM.

[21] J. Liu, W. Li, G. O. Karame, and N. Asokan. 2019. Scalable byzantine consensus via hardware-assisted secret sharing. *IEEE Trans. Computers*, 68, 1, 139–151.

[22] S. Yandamuri, I. Abraham, K. Nayak, and M. K. Reiter. 2021. Brief announcement: communication-efficient BFT using small trusted hardware to tolerate minority corruption. In *DISC* (LIPIcs). Volume 209. Schloss Dagstuhl - Leibniz-Zentrum fur Informatik.

[23] R. Rodrigues, B. Liskov, K. Chen, M. Liskov, and D. A. Schultz. 2012. Automatic reconfiguration for large-scale reliable storage systems. *IEEE Trans. Dependable Sec. Comput.*, 9, 2, 145–158.

[24] M. Castro. 2001. *Practical Byzantine Fault Tolerance*. Ph.D. MIT, (January 2001). Also as Technical Report MIT-LCS-TR-817.

[25] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. 2007. Attested append-only memory: making adversaries stick to their word. In *SOSP*. ACM.

[26] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda. 2009. TrInc: small trusted hardware for large distributed systems. In *NSDI*. USENIX Association.

[27] J. Zhang, J. Gao, K. Wang, Z. Wu, Y. Lan, Z. Guan, and Z. Chen. 2021. TBFT: understandable and efficient byzantine fault tolerance using trusted execution environment. *CoRR*, abs/2102.01970. arXiv: 2102.01970.

[28] B. Arun. 2019. *Revamping Byzantine Fault Tolerant State Machine Replicationwith Decentralization, Trusted Execution, and PracticalTransformations*. PhD thesis. Blacksburg, Virginia.

[29] T. Distler, C. Cachin, and R. Kapitza. 2016. Resource-efficient Byzantine fault tolerance. *IEEE Trans. Computers*, 65, 9, 2807–2819.

[30] S. Bano, M. Baudet, A. Ching, A. Chursin, G. Danezis, F. Garillot, Z. Li, D. Malkhi, O. Naor, D. Perelman, and A. Sonnino. State machine replication in the libra blockchain. (2019).

[31] M. M. Jalalzai, J. Niu, and C. Feng. 2020. Fast-HotStuff: A fast and resilient HotStuff protocol. *CoRR*, abs/2010.11454. arXiv: 2010.11454.

[32] M. F. Madsen, M. Gaub, M. E. Kirkbro, and S. Debois. 2019. Transforming byzantine faults using a trusted execution environment. In *EDCC*. IEEE.

[33] A. Clement, F. Junqueira, A. Kate, and R. Rodrigues. 2012. On the (limited) power of non-equivocation. In *PODC*. ACM.

[34] A. N. Bessani, J. Sousa, and E. A. P. Alchieri. 2014. State machine replication for the masses with BFT-SMART. In *DSN*. IEEE.

[35] C. Dwork, N. A. Lynch, and L. J. Stockmeyer. 1988. Consensus in the presence of partial synchrony. *J. ACM*, 35, 2, 288–323.

[36] V. Kukharenko, K. Ziborov, R. Sadykov, and R. Rezin. 2021. Verification of hotstuff bft consensus protocol with TLA+/TLC in an industrial setting. *International Scientific Conference on New Industrialization and Digitalization (NID 2020)*, 93.

[37] L. Jehl. 2021. Formal verification of hotstuff. In *FORTE* (LNCS). Volume 12719. Springer.

[38] K. Murdock, D. F. Oswald, F. D. Garcia, J. V. Bulck, D. Gruss, and F. Piessens. 2020. Plundervolt: software-based fault injection attacks against intel SGX. In *SP*. IEEE.

[39] [n. d.] Openssl. Retrieved 02/25/2020 from https://www.openssl.org/.

[40] [n. d.] Salticidae. Retrieved 03/31/2021 from https://github.com/Determinant/salticidae.

A Damysus's Correctness

Lemma 1 (Safe Prepared blocks)**.** *Let v be a view and let $v' \leq v$ be the latest view in which a block b was stored as being prepared by $f+1$ trusted components (the pair $(\boldsymbol{H}(b), v')$ is prepared in those components). Then, any block/view pair (b', v'') prepared in a trusted component such that $v'' \geq v'$, is such that $b' \succ^* b$.*

Proof Sketch 1. *We prove this invariant by induction on the view number v.*

Base Case. *For the base case, i.e., the initial view 0, only the genesis block is stored as being prepared by correct nodes, and only an extension of the genesis block can be proposed, which satisfies this property.*

Inductive Case. *Let us now prove the inductive case. Let us assume that this property holds about view v (we call this hypothesis IH), and let us prove that it holds about view v+1. Let $v' \leq v$ be the latest view in which a block b was stored as being prepared by $f+1$ trusted components (we call C that set of $f+1$ trusted components). Let L be the leader of view v+1. The property holds straightforwardly for blocks b' prepared in views $v'' \leq v$. Therefore, let us now consider the case where b' gets prepared in a trusted component in view v+1.*

For b' to get prepared in a trusted component in view v+1, it must be through `TEEstore` (Fig. 2b, ln. 23). Therefore it must be signed by $f+1$ trusted components, and more precisely by the `TEEprepare` function (we call B that set of $f+1$ nodes), and b' must have been proposed by the leader as an extension of some other block. Let us assume that L extends a block b_1 with b'. We know that $b' \succ b_1$ otherwise it would not be signed by $f+1$ nodes in the prepare phase, and in particular no correct node would sign it. We will now show that it is also the case that $b_1 \succ^* b$, which gives us that $b' \succ^* b$.

Now, because L has to use the trusted accumulator presented in Fig. 2b to select b_1, and because at least one correct node has checked that the accumulator L created is correct, we know that b_1 is the block prepared at the highest view out of $f+1$ prepare certificates used to build the accumulator. We call A those $f+1$ certificates accumulated by the leader (see Fig. 2a, ln. 7).

There is one (possibly Byzantine) node i at the intersection between A and C. Either L used b received from i to create the accumulator, or i has stored in its trusted component another prepared block at a higher view and sent it in its new-view message (signed using `TEEsign`), which by IH must be an extension of b. In both cases, it must be that L has used an extension of b to create its accumulator. However, that extension might not be the block prepared at the highest view among the blocks accumulated by L. In case another block b_2 was prepared at a higher view than b, then it will be selected by the accumulator. By IH, we know that $b_2 \succ^* b$. Therefore, the block b_1 selected by the trusted accumulator is such that $b_1 \succ^* b$. In case b is still stored as being prepared in $f+1$ trusted components in view v+1, then we conclude straightforwardly.

If some other block b'' (different from b) is stored as being prepared in f+1 trusted components in view v+1, then we will show that $b'' = b'$, from which we deduce trivially that $b' \succ^* b''$.

It remains to prove that $b'' = b'$. Because b'' is stored as being prepared by at least one trusted component, then it was signed by the trusted components of f+1 nodes, which we call D. Because there is one (possibly Byzantine) node at the intersection of B and D, and because nodes (even Byzantine) can only signs one proposal from the leader in each view, then we deduce that $b'' = b'$, which concludes our proof.

Lemma 2 (Safety). *Correct nodes never execute conflicting blocks.*

Proof Sketch 2. *Let us assume that a correct nodes i_1 executed a block b_1 in view v_1. Because b_1 was executed in view v_1, then b_1 was stored as being prepared by the trusted components of f+1 nodes in that view (see the* `pre-commit` *phase in Fig. 2a). We prove by (complete) induction on the number of views $v_2 - v_1$ between v_1 and v_2 that for any view $v_2 \geq v_1$, if b_2 gets stored as being prepared by the trusted components of f+1 nodes in view v_2, then $b_2 \succ^* b_1$. Once this is proved, we derive that if a correct node i_2 executes a block b_2 in view $v_2 \geq v_1$, then b_2 is stored as being prepared by the trusted components of f+1 nodes in view v_2, and therefore $b_2 \succ^* b_1$, which in particular implies that b_1 and b_2 are not conflicting. Let us go back to our inductive proof now. We assume that it holds for any view strictly less than v_2, from which we prove that it holds about v_2. Assume b_2 is stored as being prepared in the trusted components of f+1 nodes in view v_2. Let b be a block that was last stored as being prepared in the trusted components of f+1 nodes in a view v such that $v_1 \leq v < v_2$. By our induction hypothesis, $b \succ^* b_1$, and by Lem. 1, $b_2 \succ^* b$. Therefore, $b_2 \succ^* b_1$.*

As in HotStuff, Damysus makes use of the exponential backoff and leader election mechanisms discussed in §3, which guarantee that after GST there will eventually be a view with a correct leader such that all correct nodes stay in that view long enough to make some progress. We now prove the progress part of this statement.

Lemma 3 (Liveness). *After GST, there exists a bounded time period T such that if all correct nodes remain in view v during T and v's leader is correct, then a block is executed.*

Proof Sketch 3. *The leader L of view v collects f+1 new-view messages, and generates an accumulator from those messages. It must be that this accumulator is for the highest prepared block among the ones received. L then extends the accumulator with a block b, and sends it to all nodes. Since nodes do not lock blocks, all correct nodes will accept the proposal in the* `prepare` *phase, and will reply. L will then generate in the* `pre-commit` *phase a prepare certificate for b, which it will send to all nodes. All correct nodes will accept this prepare certificate and will vote for it. L will then generate a certificate that the prepare certificate for b has been accepted in the* `decide` *phase, which it will send to all nodes. Finally all correct nodes will receive this certificate and will execute b.*

B Chained-Damysus's Correctness

Lemma 4 (Safety). *Correct nodes never execute conflicting blocks.*

Proof Sketch 4. *Assume that a correct node i_1 executes in view v_1 a block b''_1 down a 2-chain of blocks, b_1, b'_1, and b''_1 (voted upon in views v_1, v_1−1, v_1−2); and that a correct node i_2 executes in view v_2 a block b''_2 down a 2-chain of blocks, b_2, b'_2, and b''_2 (voted upon in views v_2, v_2−1, v_2−2).*

Assume w.l.o.g. that $v_2 \geq v_1$. We now prove that $b''_2 \succ^* b''_1$. Because of the 2-chain b_1, b'_1, b''_1, f+1 nodes signed b'_1, and therefore also stored b''_1's hash value as being prepared in their trusted components in view v_1−1. In addition, because of the 2-chain b_2, b'_2, b''_2, f+1 nodes signed b'_2. Let us consider two cases.

Either $v_1 = v_2$, in which case one node must have signed both b''_1 and b''_2 in view v_1−2 = v_2−2, and since nodes (even Byzantine ones) only vote for one block per view, it must be that $b''_1 = b''_2$.

Or $v_2 \geq v_1$+1, in which case we conclude using Lem. 5 (b_2 from Lem. 5 is instantiated with b''_2, b_1 with b'_1 and b'_1 with b''_1).

Lemma 5 (Invariant). *For all views v_2 and v_1, and blocks b_2, b_1, and b'_1, if*

- *a collection A of f+1 nodes send in prepare messages a signed hash value of a block b_2 at view v_2 to the next leader, i.e., of view v_2+1,*
- *a collection B of f+1 nodes send in prepare messages a signed hash value of a block b_1 at view v_1 to the next leader, i.e., of view v_1+1,*
- *b_1 extends b'_1*
- *$v_2 \geq v_1$*
- *a collection C of f+1 nodes stored b''_1's hash value as being prepared in their trusted components at the beginning of view v_1.*

then $b_2 \succ^* b'_1$.

Proof Sketch 5. *We prove this invariant by complete induction on the number of views between v_2 and v_1.*

Base Case. *Votes for b_2 and b_1 are sent at the same view $v_2 = v_1$. Therefore, one node in $A \cap B$ votes for both b_2 and b_1 and because nodes (even Byzantine ones) can only vote for one block per view, then $b_2 = b_1 \succ b'_1$.*

Inductive Case. *The block b_2 was created by a leader, which either (1) used f+1 signatures from the previous view to create b_2's certificate; or (2) used its accumulator to generated b_2's certificate.*

In case (1) then b_2 extends a block b'_2 for which f+1 signatures have been sent at view v_2−1 (let us call D this set). Either v_2−1 = v_1, in which case one node in $D \cap B$ has voted for both

b'_2 and b_1 in view $v_2{-}1 = v_1$. Therefore, because nodes (even Byzantine ones) only vote for one block per view, it follows that $b_2 \succ b'_2 = b_1 \succ b'_1$. Or $v_2{-}1 > v_1$, in which case $f{+}1$ nodes voted for b'_2 in view $v_2{-}1$. By IH (the induction hypothesis) we conclude that $b'_2 \succ^* b'_1$, which by transitivity means that $b_2 \succ^* b'_1$.

In case (2), v_2's leader selected the highest certificate among the $f{+}1$ ones it has received (let us call E this collection), and used its accumulator to prove that it is indeed the highest one. Therefore, there is a node i in $C \cap E$. Being in E, has provided a certificate for a block that was taken as an input to the accumulator of v_2's leader. Being also in C, node i also had b'_1 stored as being prepared in its trusted component at the beginning of view v_1. We consider two cases: either i sent b'_1's hash value in its message to v_2's leader; or it has updated its prepared block in its trusted components in between the two events. In the second case, if i updated its prepared block in its trusted component, then that block must have been certified by $f{+}1$ votes for that block. Therefore, by IH, we can conclude that this block extends b'_1. Therefore, in both cases we obtain that v_2's leader used a certificate from i for a block $b_i \succ^* b$.

Now, v_2's leader will either select the certificate for b_i to produce its accumulator if it has the highest view among the messages used to produce the accumulator; or select another certificate if one has a higher view. In the first case, we conclude that $b_2 \succ b_i \succ^* b'_1$. In the second case, i.e., v_2's leader produced an accumulator for a view higher than the view of the certificate for b_i if one of the other messages it used to produce this accumulator has a higher view, then that other certificate must have been produced through $f{+}1$ votes for that certificate's block, say b_h. Therefore, by IH, we conclude that it must extend b'_1. Here again, we can conclude that $b_2 \succ b_h \succ^* b'_1$.

As in HotStuff, Chained-Damysus makes use of the exponential backoff and leader election mechanisms discussed in §3, which guarantee that after GST there will eventually be a view with a correct leader such that all correct nodes stay in that view long enough to make some progress. We now prove the progress part of this statement.

In addition, we assume here that there will eventually be three consecutive views with correct leaders. This can be achieved through different means. For example, given that there are $2f{+}1$ nodes, each leader could be the leader of two consecutive views. As there ill eventually be two consecutive correct leaders, they will be able to handle three views.

Lemma 6 (Liveness). *After GST, there exists a bounded time period T such that if all correct nodes remain in view v during T and v's leader is correct, then a block is executed.*

Proof Sketch 6. *The leader L_0 of view v checks whether it has a certificate from the previous view $v{-}1$ stored in its qc_{prep}, which would then be the certificate for the highest view. If not, it collects $f{+}1$ new-view messages, and generates an accumulator from those messages, which is then for the highest prepared block among the ones received. L_0 then extends the accumulator with a block b_0, and sends it to all nodes. Since nodes do not lock blocks, all correct nodes accept the proposed block b_0 and send their votes to the leader of the next view, i.e., view $v{+}1$.*

The leader L_1 of view $v{+}1$ receives, at the end of view v, votes from at least the $f{+}1$ correct nodes. The only block that can receive $f{+}1$ votes is b_0. L_1 then moves to view $v{+}1$ and creates an extension b_1 of b_0 (without calling its accumulator as it has a certificate for the previous view), which it sends to all nodes. Since nodes do not lock blocks, all correct nodes accept the proposed block b_1 and send their votes to the leader of the next view, i.e., view $v{+}2$. At this stage b_0 gets stored as being prepared.

The leader L_2 of view $v{+}2$ receives, at the end of view $v{+}1$, votes from at least the $f{+}1$ correct nodes. The only block that can receive $f{+}1$ votes is b_1. L_2 then moves to view $v{+}2$ and creates an extension b_2 of b_1 (without calling its accumulator as it has a certificate for the previous view), which it sends to all nodes. Since nodes do not lock blocks, all correct nodes accept the proposed block b_2. At this stage, the correct nodes store b_1 as being prepared and execute b_0.

C Damysus-C: a $2f{+}1$, 3-phase protocol

This section presents Damysus-C, a hybrid HotStuff-based protocol that uses the same communication phases and message types as in HotStuff (see §3), but requires $2f{+}1$ nodes instead of $3f{+}1$. Damysus-C uses trusted components to force nodes to propose their latest prepared blocks in new-views, and verify that blocks proposed by leaders are "safe".

C.1 Algorithm Overview

Similar to previous works, Damysus-C relies a trusted monotonic counter to ensure that leaders cannot equivocate when proposing blocks to the rest of the system. A leader resorts to validate its proposal by this counter which validates one proposal per view. All other nodes only accept proposals validated by the current leader's trusted counter. A single trusted message counter is however not enough in the context of HotStuff-like protocols because their light new-view messages do not contain enough information for replicas to check that already executed transactions are not being overwritten. Therefore, in addition, Damysus-C requires that replicas store further information about prepared and locked blocks in trusted components.[8] Locked blocks are stored in trusted components for safety, i.e., to prevent nodes from executing blocks that conflict with previously executed ones. Prepared blocks on the other hand are stored in trusted components for liveness, namely to ensure that nodes relay their prepared messages in new-view messages.

[8] Damysus-C stores the views at which blocks are prepared/locked, and for code simplicity the hash values of those blocks. Storing only the views would require trusted components to perform further computations to check the "correctness" of those blocks.

C.2 The Damysus-C Algorithm

Fig. 10a shows the pseudo-code of the non-trusted code that nodes execute, including for view changes. Fig. 10b shows the pseudo-code that trusted components execute.

In the prepare phase (starting Fig. 10a, ln. 4), the leader gathers the prepared blocks (the hash values of prepared blocks to be precise) of f+1 nodes as currently stored by their trusted components. The leader then selects the prepared block with the highest view (Fig. 10a, ln. 7) and proposes to the backups a new block that extends it (a new block is created Fig. 10a, ln. 8, which is sent ln. 10). The goal of this phase is to ensure that the proposal is "safe" in the sense of the SafeNode predicate (see §3). We say that the pair (b, v) is "safe" w.r.t. the pair (h, v') iff $b > h \lor v > v'$. This safety check is partially executed inside trusted components (Fig. 10b, ln. 19 in TEEprepare, which is called Fig. 10a, ln. 9 by the leader, and ln. 17 by the other replicas), and partially outside (Fig. 10a, ln. 16). Nodes receive the following from the leader of a given view *view*: (1) a proposed block b; (2) a hash value/view number pair (h', v') corresponding to the block that b is supposed to extend;[9] and (3) a signature generated by TEEprepare, among other things, authenticating the above information (this signature signs b's hash value, which is not sent as it can be re-computed by the replicas). From this information, nodes can piece together the prepare commitment ϕ_{prep} generated by the leader for b, which among other things guarantees that b was proposed by the leader. Nodes check outside of their trusted components that b extends h' (Fig. 10a, ln. 16). A f+1 prepare commitment (received Fig. 10a, ln. 21) guarantees that at least one correct node i has checked that the first hash value contained in the commitment corresponds to a safe bock proposed by the leader, and that i stores that proposed block. After performing this check outside of their trusted components, nodes call TEEprepare (Fig. 10a, ln. 17) and check within their trusted components that either h' is the locked hash value (therefore guaranteeing that b extends the locked hash value, since $b > h'$), or that v' is greater than the locked view (Fig. 10b, ln. 19). Note that because blocks are not signed – only their hash values are (Fig. 10a, ln. 10, the signature $\phi_{prep}\texttt{.sign}$ is a signature on b's hash value) – Byzantine leaders can still send different blocks to different nodes. However, because f+1 nodes have to perform the above checks, a f+1 prepare commitment (Fig. 10a, ln. 21) guarantees that at least one correct node has done so, and therefore holds a safe block.

In the pre-commit phase (starting Fig. 10a, ln. 19) the leader gathers those f+1 1-commitments for its proposed block, and generates a f+1-commitment that the block is prepared, which the leader disseminates to all the nodes. On receipt of such a message (Fig. 10a, ln. 25), nodes store the block as being prepared in their trusted components by calling TEEstore (Fig. 10a, ln. 26), and vote for the next phase, thereby generating 1-commitments for the next phase. For a f+1-commitment generated in that phase by the leader Fig. 10a, ln. 22, and received ln. 25 by all other nodes, to be verified by TEEstore it must be that all its 1-commitments are of the form $\langle h, view, h', v', \texttt{prep\_p}, \sigma\rangle$ such that f+1 nodes checked $(h' = lockh \lor v' > lockv)$ (where $lockh$ and $lockv$ are the locked hash value/view stored in their trusted components) and 1 correct node has checked that $h = \mathbf{H}(b) > h'$ for some block b and holds that block.

In the commit phase (starting Fig. 10a, ln. 28), the leader gathers f+1 1-commitments for a block being pre-committed, out of which it creates a $(f$+1)-commitment that it sends to all nodes. On receipt of such a $(f$+1)-commitment (Fig. 10a, ln. 34), a node then locks the proposed block in its trusted component by calling TEEstore again (Fig. 10a, ln. 35). This in turn generates a 1-commitment that nodes send back to the leader.

In the decide phase (starting Fig. 10a, ln. 37), the leader gathers f+1 1-commitments for a block being locked, out of which it creates a $(f$+1)-commitment, which it sends to all nodes. On receipt of such a $(f$+1)-commitment (Fig. 10a, ln. 43), a node can finally execute the locked block.

The new-view phase (starting Fig. 10a, ln. 47) is triggered either when a node executes a block, or when a view times out. Timers allow making progress in case the current leader is faulty. They are started when nodes send new-view messages for a given view, and canceled when a block is executed in that view (see [9] for details). In the new-view phase, a node calls TEEsign to sign its latest prepared block. No block is proposed in that phase: the first hash value field is $\perp$ (see Fig. 10b, ln. 14). Note that when a node is lagging behind, it calls TEEsign until it can generate a new-view-phase commitment for the next view.

C.3 Proof of Correctness

Lemma 7 (Safe Prepared/Locked blocks)**.** *Let v_0 be the latest view in which a block b_0 got locked by f+1 trusted components (i.e., the pair $(\boldsymbol{H}(b), v_0)$ is locked in those components). Any block/view pair $(b_1, v' \geq v_0)$ prepared or locked in a trusted component in a view $v_1 \geq v_0$ is such that $b_1 >^* b_0$.*

Proof Sketch 7. *We prove this invariant by induction on the view number v_1.*

Base Case. *For the base case, i.e., the initial view 0, only the genesis block is stored in trusted components, which satisfies this property.*

Inductive Case. *Let us now assume that this property holds about view v_1 (we call this hypothesis IH, this induction hypothesis), and let us prove that it holds for view v_1+1. Let $v_0 \leq v_1$ be the latest view in which a block b_0 was locked by a set T of f+1 trusted components. Let L be the leader of view v_1+1. The property extends directly to blocks stored in trusted components*

[9] Note that in that phase, v' is used to check safety. Once checked, backups would not need to sign it except that if they don't their message would have a different format from the leader's.

Figure 10 Damysus-C ($2f$+1 replicas and 3 phases).

(a) Non-trusted code at replica i

```
$view = 0$ // current view (duplicates the one in the TEE)
$pubs$ // public keys (duplicate the ones in the TEE)

// prepare phase
as a leader
    wait for $\vec{\phi}$ s.t. C-match($\vec{\phi}, f+1, \bot, view,$ nv_p)
    $\phi_{nv}$ := commitment $\phi \in \vec{\phi}$ with highest $\phi$.Vjust
    $b$ := createLeaf($\phi_{nv}$.Hjust, client transactions)
    $\phi_{prep}$ := TEEprepare($\mathbf{H}(b), \phi_{nv}$) // also sent to itself
    send $\langle \phi_{nv}, b, \phi_{prep}$.sign$\rangle$ to backups

as a backup
    wait for $\langle\langle \bot, view, h', v',$ nv_p$, \sigma\rangle, b, \sigma'\rangle$ from the leader
    $\phi_{nv} := \langle \bot, view, h', v',$ nv_p$, \sigma\rangle$
    $\phi_{prep} := \langle \mathbf{H}(b), view, h', v',$ prep_p$, \sigma'\rangle$
    abort if $\neg$(VERIFY($\phi_{prep}$)$_{pubs} \wedge b > h'$)
    send $\phi'$ := TEEprepare($\mathbf{H}(b), \phi_{nv}$) to leader

// pre-commit phase
as a leader
    wait for $\vec{\phi}$ s.t. C-match($\vec{\phi}, f+1, h, view,$ prep_p)
    send $\phi$ := C-combine($\vec{\phi}$) to all replicas

all replicas
    wait for $\langle h, view, h', v',$ prep_p$, \vec{\sigma}^{f+1}\rangle$ from leader
    send $\phi$ := TEEstore($\langle h, view, h', v',$ prep_p$, \vec{\sigma}^{f+1}\rangle$) to leader

// commit phase
as a leader
    wait for $\vec{\phi}$ s.t. C-match($\vec{\phi}, f+1, h, view,$ pcom_p)
    send $\phi$ := C-combine($\vec{\phi}$) to all replicas

all replicas
    wait for $\langle h, view, \bot, \bot,$ pcom_p$, \vec{\sigma}^{f+1}\rangle$ from leader
    send $\phi$ := TEEstore($\langle h, view, \bot, \bot,$ pcom_p$, \vec{\sigma}^{f+1}\rangle$) to leader

// decide phase
as a leader
    wait for $\vec{\phi}$ s.t. C-match($\vec{\phi}, f+1, h, view,$ com_p)
    send $\phi$ := C-combine($\vec{\phi}$) to all replicas

all replicas
    wait for $\langle h, view, \bot, \bot,$ com_p$, \vec{\sigma}^{f+1}\rangle$ from leader
    abort if $\neg$VERIFY($\langle h, view, \bot, \bot,$ com_p$, \vec{\sigma}^{f+1}\rangle$)$_{pubs}$
    execute $b$ corresponding to $h$ & reply to clients

// new-view phase
all replicas
    when executed or timeout
    $(v, ph)$ := ($view$, prep_p); $view$++
    while $(v, ph) \neq$ ($view$, nv_p) do
        $\phi$ := TEEsign(); $(v, ph)$ := ($\phi$.Vprep, $\phi$.phase)
    end while
    send $\phi$ to $view$'s leader
```

(b) TEE code

```
$lockh = \mathbf{H}(G)$ // hash value      $lockv = 0$ // view number
$preph = \mathbf{H}(G)$ // hash value      $prepv = 0$ // view number
$(view, phase) = (0,$ nv_p) // current step
$priv_c$ // confidential private key
$pubs$ // public keys

function createUniqueSign($h, h', v'$)
    $\sigma$ := SIGN($h, view, h', v', phase$)$_{priv_c}$
    $\phi := \langle h, view, h', v', phase, \sigma\rangle$
    $(view, phase)$++ // increased to avoid equivocation
    return $\phi$

function TEEsign()
    return createUniqueSign($\bot, preph, prepv$)

function TEEprepare($h, \phi$)
    $\langle \bot, v, h', v', ph, \sigma\rangle := \phi$
    if VERIFY($\phi$)$_{pubs} \wedge view{=}v \wedge ph =$ nv_p $\wedge\ h \neq \bot$ then
        if $h'{=}lockh \vee v' > lockv$ then // "safety" check
            return createUniqueSign($h, h', v'$)
        end if
    end if

function TEEstore($\phi$)
    $\langle h, v, h', v', ph, \vec{\sigma}^{f+1}\rangle := \phi$
    if VERIFY($\phi$)$_{pubs} \wedge view{=}v \wedge ph \in$ {prep_p, pcom_p} then
        $preph := h$; $prepv := v$
        if $ph$ = pcom_p then $lockh := h$; $lockv := v$
        end if
        return createUniqueSign($h, \bot, \bot$)
    end if
```

that do not change from view v_1 to view v_1+1. Therefore, let us now consider the case where a new block gets prepared or locked in a trusted component in view v_1+1.

*Let us assume that L picks a block b_1 out of a new-view message, and extends it with b_2 (b_2 must extend b_1 otherwise it would not be accepted by f+1 nodes in the prepare phase). For b_2 to be prepared by a node in view v_1+1, it must be considered a "safe" extension of b_1 by f+1 nodes. Therefore, at those f+1 nodes, either b_2 is an extension of the locked block (see **Case 1** below), or b_1's view is higher than the locked view, and there must be at least one node j, possibly a Byzantine node, at the intersection of these f+1 nodes and T (see **Case 2** below).*

***Case 1.** In the first case, where b_2 is an extension of the locked block, then by monotonicity of the views, it must be that the locked block has a view $\geq v_0$ (because $j \in T$). Therefore, by IH (the induction hypothesis), it must be that $b_2 \succ^* b_0$.*

***Case 2.** In the second case, where b_1's view is higher than the locked view (which must be $\geq v_0$ by monotonicity of the views and because $j \in T$), it must be that b_1 was a block held* *as being prepared in a trusted component at the end of view v_1, and by IH (the induction hypothesis) it must be that $b_1 \succ^* b_0$.*

Therefore, in both cases we obtain that $b_2 \succ^* b_0$. As b_2 can be the only block prepared at view v_1+1, we will only prove the property about this block. It holds straightforwardly if the latest view at which a block was locked by f+1 trusted components stays v' because $b_2 \succ^* b_0$. Moreover, if b_2 gets locked in view v_1+1 by f+1 nodes, then the property also holds straightforwardly because b_2 is the only new prepared block in that view, and it is also the latest block locked by f+1 nodes. The same reasoning applies to locked blocks.

Lemma 8 (Safety). *Correct nodes never execute conflicting blocks.*

Proof Sketch 8. *Let us assume that a correct node i_1 executed a block b_1 in view v_1. Because b_1 was executed in view v_1, then b_1 was locked in f+1 trusted components in that view. We prove by (complete) induction on the number of views $v_2 - v_1$ between v_1 and v_2 that for any view $v_2 \geq v_1$, if b_2 gets locked by f+1 trusted components in view v_2, then $b_2 \succ^* b_1$. Once this is proved, we derive that if a correct node i_2 executes a block b_2 in view $v_2 \geq v_1$, then b_2 is locked by f+1 trusted components in view v_2, and therefore $b_2 \succ^* b_1$, which in particular implies that b_1 and b_2 are not conflicting.*

Let us go back to our inductive proof now. We assume that it holds for any view strictly less than v_2, from which we prove that it holds about v_2. Assume b_2 is locked by f+1 trusted components in view v_2. Let b be the block that was last locked by f+1 trusted components in a view v such that $v_1 \leq v < v_2$. By our induction hypothesis, $b \succ^* b_1$, and by Lem. 7, $b_2 \succ^* b$. Therefore, $b_2 \succ^* b_1$.

As in HotStuff, Damysus-C makes use of the exponential backoff and leader election mechanisms discussed in §3, which guarantee that after GST there will eventually be a view with a correct leader such that all correct nodes stay in that view long enough to make some progress. We now prove the progress part of this statement.

Lemma 9 (Liveness). *After GST, there exists a bounded time period T such that if all correct nodes remain in view v during T and v's leader is correct, then a block is executed.*

Proof Sketch 9. *The leader of view v, say L, collects f+1* new-view *messages, and selects the one with the highest view. Let (b_1, v_1) be the block/view pair selected by the leader. It must be that v_1 is as high as the f+1 views received by L in new-view messages (let A be that set of new-views). Let (b_0, v_0) be the highest locked block at the beginning of the view. Since (b_0, v_0) got locked by a node, then f+1 nodes prepared it, and there is a (possibly Byzantine) node j at the intersection of these f+1 nodes and A. Therefore, L must have received a prepared block with a view at least as high as v_0 from j (by monotonicity of the views). Therefore, $v_1 \geq v_0$.*

L then extends b_1 with a block b_2 and proposes it to the other nodes (after having checked it for "safety"—as L is correct, we explain below why this check succeeds). All correct nodes will receive the proposal, and will check it for "safety" against their own locked block.

Let i be such a correct node (the same reasoning applies to L before it sends its proposal). Because i stores a locked block/view pair (b', v'), that block must have been prepared in view v' by f+1 nodes, which means that L must have received a view at least as v', and that therefore $v' \leq v_1$. Let us consider the two other cases where $v' < v_1$ and $v' = v_1$.

If $v' < v_1$ then L's proposal would be deemed "safe" (thanks to the liveness part of the "safety" predicate) and i would prepare it. Otherwise, if $v' = v_1$, then it must be that f+1 nodes have stored b_1 as being prepared in view v_1, and also that f+1 nodes have stored b' as being prepared in view v_1.

Therefore, there is a (possibly Byzantine) node that stored both b_1 and b' as being prepared in view v_1. Since only one block can ever be stored as being prepared in a view, it must be that $b_1 = b'$, and so $b_2 \succ b'$, resulting in i considering L's proposal to be "safe" (thanks to the safety part of the "safety" predicate) and consequently prepared it. Therefore, all correct nodes will prepare b_2, and will vote for that block also on all subsequent phases, leading to b_2 being executed by all correct nodes.

D Damysus-A: a $3f$+1, 2-phase protocol

While §C targeted increasing the resilience of HotStuff by tolerating more Byzantine nodes, this section aims at reducing HotStuff's latency namely by reducing the number of phases. We achieve this by requiring leaders to use a simple trusted message accumulator, which guarantees that a leader cannot lie about the latest prepared block received – hence the latest prepared block in the system.

To recall, at view change a leader in HotStuff gathers quorum certificates of the latest prepared blocks from $2f$+1 nodes (of which at least f+1 are correct). This guarantees that a leader of any view receives at view change the latest prepared block in the system (because there is a correct node at the intersection of any prepare and new-view certificates). However, in HotStuff, a malicious leader can simply lie about the latest prepared block received and claim some other stale prepared block as the latest one. Given some means to guarantee that a leader relays the highest prepared block in the system, we argue that nodes can then execute in two phases instead of three, namely by completely omitting the commit phase (including the need to lock). We show that the latter is possible and that leaders' honesty about the latest prepared block received can be achieved by requiring leaders to use our trusted accumulator. Also as a result of reducing the number of phases per view, the SafeNode predicate of HotStuff changes (see §3).

We next provide a high level overview of our trusted accumulator component and how HotStuff can use our trusted accumulator to operate in two phases.

D.1 Definitions

Message A message m is of the form $\langle ph, v, x, y, \vec{\sigma}\rangle$, where x is either a block b or a hash value h being voted upon in phase ph of view v, and y is either a certificate qc, or an accumulator acc. Both x and y can be empty, in which case we use $\perp$.

We write $\vec{m}$ for a list of messages, and $\langle ph, v, x, y, \sigma\rangle$ for a message of the form $\langle ph, v, x, y, [\sigma]\rangle$. Given m of the form $\langle ph, v, x, y, \vec{\sigma}\rangle$, let $m._{\texttt{cert}}$ be y.

Given $\vec{m} = [\langle ph_1, v_1, x_1, y_1, \sigma_1\rangle, \dots, \langle ph_n, v_n, x_n, y_n, \sigma_n\rangle]$, a list of messages, we define the following operation to check whether we have received enough messages of a certain kind: let $\texttt{M-match}(\vec{m}, k, ph, v)$ be true iff (1) $n = k$; (2) $\forall i \in \{1, \dots, n\}.ph = ph_i \wedge v = v_i$; and (3) all n signatures have been created by different nodes.

Let $\texttt{M-combine}([\langle ph, v, x, y, \sigma_1\rangle, \dots, \langle ph, v, x, y, \sigma_n\rangle])$ be the message $\langle ph, v, x, y, [\sigma_1, \dots, \sigma_n]\rangle$.

D.2 The Damysus-A Algorithm

Fig. 11 specifies a version of basic HotStuff that relies on trusted accumulators. Fig. 11a shows the non-trusted code executed by replicas, while Fig. 11b shows the trusted code that leaders should execute to accumulate new-view messages and create valid new proposals. In particular the non-trusted accumList function (Fig. 11a, ln. 52) shows how Damysus-A combines the trusted TEEstart, TEEaccum, and TEEfinalize functions presented next, to produce an accumulator: it initializes the accumulator with the new-view message received by the leader containing the quorum certificate for the highest prepared block; it then extends the accumulator using the other new-view messages; and finally finalizes it by replacing the ids of the nodes that contributed to the accumulator by the size of that set.

An accumulator is initialized using TEEstart, which takes a message containing a certificate qc, and creates an accumulator that contains this certificate only. TEEaccum takes an accumulator acc and a message m containing a certificate qc, and adds m's signer to acc if qc is for a view at most as high as the view contained in the accumulator. TEEfinalize produces accumulators of a slightly different shape. Instead of containing the identities of the nodes that have contributed to the accumulator, the one returned by TEEfinalize simply contains the number of nodes that have contributed. When backups receive such accumulators, they can then check that this number is greater than or equal to $2f$+1.

D.3 Proof of Correctness

We now prove that the proposed algorithm of Fig. 11 implements all Byzantine agreement properties as in basic HotStuff.

Lemma 10 (Safe Prepared blocks). *Let v be a view and let $v' \leq v$ be the latest view in which a block b was stored as being prepared by f+1 correct nodes (held in the qc_{prep} variable). Then, any block/view pair (b', v'') prepared (signed by $2f$+1 nodes) in view v'' such that $v'' \geq v'$, is such that $b' >^* b$.*

Proof Sketch 10. *We prove this invariant by induction on the view number v.*

Base Case. *For the base case, i.e., the initial view 0, only the genesis block is stored as being prepared by correct nodes, and only an extension of the genesis block can be proposed, which satisfies this property.*

Inductive Case. *Let us now prove the inductive case. Let us assume that this property holds about view v (we call this hypothesis IH), and let us prove that it holds about view v+1. Let $v' \leq v$ be the latest view in which a block b was stored as being prepared by f+1 correct nodes (we call C that set of f+1 correct nodes). Let L be the leader of view v+1. The property holds straightforwardly for blocks b' prepared in views $v'' \leq v$. Therefore, let us now consider the case where b' gets prepared in view v+1.*

For b' to get prepared in view v+1, it must be signed by $2f$+1 nodes (we call B that set of $2f$+1 nodes), and b' must have been proposed by the leader as an extension of some other block. Let us assume that L extends a block b_1 with b'. We know that $b' > b_1$ otherwise it would not be signed by $2f$+1 nodes in the prepare phase, and in particular no correct node would sign it. We will now show that it is also the case that $b_1 >^* b$, which gives us that $b' >^* b$.

Now, because L has to use the trusted accumulator presented in Fig. 2bto select b_1, and because at least one correct node has checked that the accumulator L created is correct, we know that b_1 is the block prepared at the highest view out of $2f$+1 prepare certificates used to build the accumulator. We call A those $2f$+1 certificates accumulated by the leader (see Fig. 11a, ln. 10).

Because there is one correct node i at the intersection between A and C, then either L used b received from i to create the accumulator (this is one among the $2f$+1 messages used to create the accumulator), or i has now prepared another block at a higher view and sent it in its new-view message, which by IH must be an extension of b. In both cases, it must be that L has used an extension of b received from i to create its accumulator. However, that extension might not be the block prepared at the highest view among the blocks accumulated by L. In case another block b_2 was prepared at a higher view than b, then it will be selected by the accumulator. By IH, we know that $b_2 >^* b$. Therefore, the block b_1 selected by the trusted accumulator is such that $b_1 >^* b$. In case b is still stored as being prepared by f+1 correct nodes in view v+1, then we conclude straightforwardly. If some other block b'' (different from b) is stored as being prepared by f+1 correct nodes in view v+1, then we will show that $b'' = b'$, from which we deduce trivially that $b' >^* b''$.

It remains to prove that $b'' = b'$. Because b'' is prepared by at least one correct node, then it was signed by $2f$+1 nodes, which we call D. Because there is one correct node at the intersection

Figure 11 Damysus-A ($3f{+}1$ replicas and 2 phases).

(a) Non-trusted code at replica i

```
 1: $view = 0$ // current view
 2: $priv_{nc}$ // non-TEE stored private key
 3: $qc_{prep} := \langle 0, \mathbf{H}(G), [\,] \rangle$ // last prepared certificate
 4: $pubs$ // Public keys of TEE-generated $\sigma$'s
 5: $pubs'$ // Public keys of non-TEE-generated $\sigma$'s
 6:
 7: // prepare phase
 8: as a leader
 9:     wait for $\vec{m}$ s.t. M-match($\vec{m}$, $2f{+}1$, nv_p, $view$)
10:     $acc$ := accumList($\vec{m}$)
11:     $b$ := createLeaf($acc$.hash, client transactions)
12:     $\sigma$ := SIGN(prep_p, $view$, $b$, $acc$)$_{priv_{nc}}$
13:     send $m := \langle$prep_p, $view, b, acc, \sigma\rangle$ all replicas
14:
15: all replicas
16:     wait for $\langle$prep_p, $view, b, acc, \sigma\rangle$ from the leader
17:     abort if $\neg$VERIFY($\langle$prep_p, $view, b, acc, \sigma\rangle$)$_{pubs'}$
18:     abort if $\neg$VERIFY($acc$)$_{pubs}$
19:     abort if $\neg(|acc| = 2f{+}1 \wedge b \succ acc$.hash$)$
20:     $\sigma$ := SIGN(prep_p, $view$, $b$, $\bot$)$_{priv_{nc}}$
21:     send $m := \langle$prep_p, $view, \mathbf{H}(b), \bot, \sigma\rangle$ to leader
22:
23: // pre-commit phase
24: as a leader
25:     wait for $\vec{m}$ s.t. M-match($\vec{m}$, $2f{+}1$, prep_p, $view$)
26:     send $m$ := M-combine($\vec{m}$) to all replicas
27:
28: all replicas
29:     wait for $\langle$prep_p, $view, h, \bot, \vec{\sigma}^{2f+1}\rangle$ from the leader
30:     abort if $\neg$VERIFY($\langle$prep_p, $view, h, \bot, \vec{\sigma}^{2f+1}\rangle$)$_{pubs'}$
31:     $qc_{prep} := \langle view, h, \vec{\sigma}^{2f+1}\rangle$
32:     $\sigma$ := SIGN(pcom_p, $view$, $h$, $\bot$)$_{priv_{nc}}$
33:     send $m := \langle$pcom_p, $view, h, \bot, \sigma\rangle$ to leader
34:
35: // decide phase
36: as a leader
37:     wait for $\vec{m}$ s.t. M-match($\vec{m}$, $2f{+}1$, $view$, pcom_p)
38:     send $m$ := M-combine($\vec{m}$) to all replicas
39:
40: all replicas
41:     wait for $\langle$pcom_p, $view, h, \bot, \vec{\sigma}^{2f+1}\rangle$ from leader
42:     abort if $\neg$VERIFY($\langle$pcom_p, $view, h, \bot, \vec{\sigma}^{2f+1}\rangle$)$_{pubs'}$
43:     execute $b$ corresponding to $h$ & reply to clients
44:
45: // new-view phase
46: all replicas
47:     when executed or timeout
48:     $view$++
49:     $\sigma$ := SIGN(nv_p, $view$, $qc_{prep}$)$_{priv_{nc}}$
50:     send $m := \langle$nv_p, $view, \bot, qc_{prep}, \sigma\rangle$ to $view$'s leader
51:
52: function accumList($\vec{m}$)
53:     $m_0$ := message $m \in \vec{m}$ with highest $m$.cert.view
54:     $acc$ := TEEstart($m_0$)
55:     for $m \in \vec{m} \setminus \{m_0\}$ do $acc$ := TEEaccum($acc$, $m$)
56:     return TEEfinalize($acc$)
57:
```

(b) TEE-based accumulator

```
 1: $priv_c$ // confidential private key
 2: $pubs$ // Public keys of TEE-generated $\sigma$'s
 3: $pubs'$ // Public keys of non-TEE-generated $\sigma$'s
 4:
 5: function TEEstart($m$)
 6:     $\langle$nv_p, $view, \bot, \langle v, h, \vec{\sigma}\rangle, \sigma\rangle := m$
 7:     if VERIFY($m$)$_{pubs'}$ $\wedge$ VERIFY($\langle v, h, \vec{\sigma}\rangle$)$_{pubs'}$ then
 8:         $\sigma'$ := SIGN($view, v, h, [\sigma$.id$]$)$_{priv_c}$
 9:         return $acc := \langle view, v, h, [\sigma$.id$], \sigma'\rangle$
10:     end if
11:
12: function TEEaccum($acc$, $m$)
13:     $\langle v_1, v_1', h_1, \vec{i}, \sigma\rangle := acc$
14:     $\langle$nv_p, $v_2, \bot, \langle v_2', h_2, \vec{\sigma}_2\rangle, \sigma_2\rangle := m$
15:     if $\left(\begin{array}{l} v_1 = v_2 \wedge v_1' \geq v_2' \wedge \sigma_2.\text{id} \notin \vec{i} \\ \wedge \text{VERIFY}(acc)_{pubs} \wedge \text{VERIFY}(m)_{pubs'} \end{array}\right)$ then
16:         $\sigma'$ := SIGN($v_1, v_1', h_1, \vec{i}$@$[\sigma_2$.id$]$)$_{priv_c}$
17:         return $acc' := \langle v_1, v_1', h_1, \vec{i}$@$[\sigma_2$.id$], \sigma'\rangle$
18:     end if
19:
20: function TEEfinalize($acc$)
21:     $\langle view, v, h, \vec{i}, \sigma\rangle := acc$
22:     if VERIFY($acc$)$_{pubs}$ then
23:         $\sigma$ := SIGN($view, v, h, |\vec{i}|$)$_{priv_c}$
24:         return $acc' := \langle view, v, h, |\vec{i}|, \sigma\rangle$
25:     end if
26:
```

of B and D, and because a correct node only signs one proposal from the leader, then we deduce that $b'' = b'$, which concludes our proof.

Lemma 11 (Safety of the Algorithm presented in Fig. 11). *Correct nodes never execute conflicting blocks.*

Proof Sketch 11. *Let us assume that a correct nodes i_1 executed a block b_1 in view v_1. Because b_1 was executed in view v_1, then b_1 was stored as being prepared by $f{+}1$ correct nodes in that view (see Fig. 11a, ln. 31). We prove by (complete) induction on the number of views $v_2 - v_1$ between v_1 and v_2 that for any view $v_2 \geq v_1$, if b_2 gets stored as being prepared by $f{+}1$ correct nodes in view v_2, then $b_2 \succ^* b_1$. Once this is proved, we derive that if a correct node i_2 executes a block b_2 in view $v_2 \geq v_1$, then b_2 is stored as being prepared by $f{+}1$ correct nodes in view v_2, and therefore $b_2 \succ^* b_1$, which in particular implies that b_1 and b_2 are not conflicting.*

Let us go back to our inductive proof now. We assume that it holds for any view strictly less than v_2, from which we prove that it holds about v_2. Assume b_2 is stored as being prepared by $f{+}1$ correct nodes in view v_2. Let b be a block that was last stored as being prepared by $f{+}1$ correct nodes in a view v such that $v_1 \leq v < v_2$. By our induction hypothesis, $b \succ^* b_1$, and by Lem. 10, $b_2 \succ^* b$. Therefore, $b_2 \succ^* b_1$.

Since blocks do not get locked in the first phase, nodes that have received a prepare certificate can still participate in the next phase.

As in HotStuff, `Damysus-A` makes use of the exponential backoff and leader election mechanisms discussed in §3, which guarantee that after GST there will eventually be a view with a correct leader such that all correct nodes stay in that view long enough to make some progress. We now prove the progress part of this statement.

Lemma 12 (Liveness of the Algorithm presented in Fig. 11)**.** *After GST, there exists a bounded time period T such that if all correct nodes remain in view v during T and the v's leader is correct, then a block is executed.*

Proof Sketch 12. *The leader L of view v collects $2f+1$ new-view messages, and generates an accumulator from those messages. It must be that this accumulator is for the highest prepared block. It then extends the accumulator with a block b, and sends it to all nodes. Since nodes do not lock blocks, all correct nodes will accept the proposal, and will reply. L will then generate a prepare certificate for b, which it will send to all nodes. All correct nodes will accept this prepare certificate and will vote for it. L will then generate a certificate that the prepare certificate for b has been accepted, which it will send to all nodes. Finally all correct nodes will receive this certificate and will execute b.*