No-Commit Proofs: Defeating Livelock in BFT

Neil Giridharan*, Heidi Howard†‡, Ittai Abraham†, Natacha Crooks*, Alin Tomescu†
*UC Berkeley, †VMware Research, ‡University of Cambridge

***Abstract*—This paper presents the design and evaluation of Wendy, the first Byzantine consensus protocol that achieves optimal latency (two phases), linear authenticator complexity, and optimistic responsiveness. Wendy's core technical contribution is a novel aggregate signature scheme that allows leaders to prove, with constant pairing cost, that an operation *did not commit*. This *No-commit proof* addresses prior liveness concerns in protocols with linear authenticator complexity (including view change), allowing Wendy to commit operations in two-phases only.**

I. Introduction

Blockchains and cryptocurrencies have become a popular tool for decentralized trust in a wide-variety of applications, from payment systems to healthcare and supply chains. At the core of these systems are *byzantine-fault tolerant (BFT) consensus protocols* [1]. A BFT consensus protocol ensures that a set of mutually distrustful participants will reach agreement on the same sequence of operations. Existing protocols, however, map poorly to the blockchain ecosystem. Diem [2], for instance, Facebook's new payment network, targets large numbers of geo-distributed participants, often on mobile devices. It thus requires a protocol that scales well with the number of participants and can tolerate the varying latencies of cross-continental links [3]. Current solutions (Table I) suffer from at least one of the following deficiencies: a) incurring cryptographic costs (referred to as authenticator complexity [4]) that grow quadratically with the number of participants [5], [6], b) incurring additional round-trips [4], c) lacking optimistic responsiveness (their latency depends on a fixed worst case network delay bound rather than on the actual network delay) [7], [8], or d) requiring prohibitively-expensive cryptography [9].

At first glance, it would seem that the tradeoffs made by existing BFT protocols are fundamental: that linear authenticator complexity necessarily comes at the cost of additional phases or reduced responsiveness. This paper shows otherwise. We present Wendy, a new BFT consensus protocol that achieves all of optimal latency (two rounds of communication), optimistic responsiveness, and linear authenticator complexity. To the best of our knowledge, Wendy is the first system to achieve all three properties.

The primary bottleneck of existing Byzantine consensus protocols is the *view-change*, the mechanism through which new leaders are replaced. Prior work considered view changes to be rare, and for leaders to be stable. Concerns about fairness and frontrunning [10] point to a need in blockchain systems to *rotate* leaders frequently, thus increasing the frequency of *view changes* and requiring them to be fast. Specifically, the view change logic is tasked with preserving all committed or ongoing operations in the consensus log from the old to the new leader. Therein lies the key challenge: a new leader must prove to all other replicas that the set of committed operations it proposes to include in the new view is up-to-date. Otherwise replicas will remain *locked* on the proposal they believe is the most recent. PBFT [5] unlocks replicas in a straightforward manner: it provably includes all received proposals. This comes at a cost: $O(n^2)$ messages must be verified (*no linearity*) [11], [12]. Convincing skeptical replicas without this quadratic cost is more challenging as a leader could intentionally omit the most recent proposal in the view change. Tendermint [7] and Casper FFG [8] sidestep this issue by making an additional network synchrony assumption (*no optimistic responsiveness*) and asserting that, after a fixed upper bound Δ, all honest nodes will have necessarily received the most recent committed proposal. HotStuff [4] makes no such assumption, but instead must add a third communication round (*suboptimal latency*).

Wendy takes a different approach. It relies on explicit *no-commit proofs* that allow the leader to prove to other replicas, in a single round-trip *and only when necessary*, that their locked proposal was not committed. Crucially, we present a novel aggregate signature scheme that ensures that such a no-commit proof only requires two *cryptographic pairings* [14] to verify, in keeping with the ethos of linearity. When compared to HotStuff, in a wide-area network, Wendy achieves 33% lower latency and comparable throughput. No-commit proofs have constant verification time, which in the presence of byzantine attackers ensures that performance remains high, even at scale.

In summary, we make the following contributions:

1) A new aggregate signature scheme tailored specifically for no-commit proofs, with constant pairing cost. This new building block and the new no-commit proof construct it enables, we believe, is of independent interest.
2) We use this no-commit proof to design the first BFT protocol, Wendy, that can commit operations in two round-trips, while providing linear authenticator complexity and optimistic responsiveness.
3) We implement and evaluate Wendy, and demonstrate latency improvements over HotStuff while maintaining comparable throughput.

II. Background

This work focuses on leader-based, partially synchronous BFT protocols, which assume an upper-bound on maximum

TABLE I: Comparison of BFT protocols

Protocol	Authenticator Complexity (View Change)	Optimistic Responsiveness	# of rounds	Leader paradigm
PBFT [5]	$O(n^2)$ or $O(n^3)$	Yes	2	Stable
SBFT [13]	$O(n^2)$	Yes	1-2	Stable
Tendermint [7]	$O(n)$	No	2	Rotating
Casper FFG [8]	$O(n)$	No	2	Rotating
HotStuff [4]	$O(n)$	Yes	3	Rotating
Wendy	$\mathbf{O(n)}$	**Yes**	**2**	**Rotating**

message delay for guaranteed liveness after Global Stabilization Time (GST). These protocols underpin popular permissioned blockchains today such as Concord [15] and Diem. We summarize related research efforts along three axes: latency-optimality, optimistic responsiveness, and linearity. Finally, we describe the tension between latency-optimality and liveness, and give an example of the *hidden-lock problem*, an execution in which linear protocols can livelock if implemented without care.

A. Prior work

Overview Leader-based BFT protocols proceed in a sequence of *views*. For each view, a leader is responsible for finalizing consensus decisions for operations previously submitted by clients. Each view traditionally consists of two primary phases: a non-equivocation phase, and a persistence phase. The non-equivocation phase ensures that at most one proposal can reach agreement in the current view, even in the presence of a Byzantine leader, while the persistence phase ensures that an agreed-upon decision (if any) is preserved across views. The view-change protocol replaces an older leader with a new leader, and guarantees that if a consensus decision is made, then the new leader must abide by it.

No linear authenticator complexity PBFT [5] was the first practical leader-based protocol, and remains the most influential BFT protocol. It consists of a pre-prepare and prepare phase, which together form the non-equivocation phase. In the pre-prepare phase, the leader broadcasts a command to all replicas. The replicas subsequently broadcast *prepare* messages to all. Having received a *prepare quorum certificate* ($2f+1$ prepare messages), replicas move on to the commit phase, which is the persistence phase, and broadcast *commit* messages. Replicas execute the command after forming a *commit quorum certificate* ($2f+1$ commit messages). This all-to-all pattern of communication allows PBFT to commit an operation in two round-trips but incurs $O(n^2)$ authenticator complexity. View-changes are especially costly in PBFT-like systems: the new leader must collect prepare certificates from all replicas and include all these votes in a *new-view* message. Authenticator complexity is thus cubic: n replicas must verify $2f+1$ prepare certificates, each of which consists of $2f+1$ messages. This cost can be reduced to $O(n^2)$ by using signatures rather than simple message authentication codes (MACs). Subsequent work augments PBFT with a one round-trip fast-path [6], [16] by requiring $3f+1$ matching replies in the first phase but does not improve authenticator complexity. SBFT [13] reduces the common-case operation cost to $O(n)$ while preserving the optimistic fast-path by relying on a combination of collector nodes and threshold signatures. The view change procedure, however, is still $O(n^2)$. Non-linear protocols scale especially poorly when the number of participants grows large.

Concurrent research efforts, Fast-Hotstuff [17] and Jolteon/Ditto [18], also achieve Byzantine consensus in two phases with optimistic responsiveness. They both achieve linear authenticator complexity in the common case, but fall back to quadratic authenticator complexity in the view change when the leader is faulty. MSCFCL [9] does indeed achieve linear authenticator complexity, but requires the use of expensive succinct arguments of knowledge.

No optimistic responsiveness A protocol is *optimistically responsive* if, after Global Stabilization Time (GST) [19] in the steady state, a non-faulty leader can reach consensus in time depending only on the actual network delays, independent of the GST worst-case message delay parameter, Δ. The idea of optimistic responsiveness was introduced by Thunderella [20], which combined an optimistic fast-path with a slow, synchronous path in the presence of failures. Tendermint [7] and Casper FFG [8] made similar networking assumptions to reduce the cost of PBFT's view chain down to $O(n)$ at the cost of progress being defined by a worst-case delay Δ. Lack of optimistic responsiveness is especially problematic in the wide-area setting where networks exhibit long tail latencies, as the worst-case delay is significantly worse than the actual average network delay. This can cause significantly longer slow downs in the presence of Byzantine actors or network hiccups.

Suboptimal latency HotStuff adopts a similar communication structure to SBFT: it eschews all-to-all communication between replicas and has replicas exchange messages with the leader only. HotStuff operates in the lock-commit paradigm: replicas become locked on a value when it is possible that the value could have been committed by some other replica. The replica unlocks only when it is shown by the leader that the locked value definitely did not commit. HotStuff hence consists of three phases: a prepare phase (non-equivocation phase), a precommit phase (persistence phase), and a commit phase. In the prepare phase, the leader broadcasts a proposal to the replicas and waits for a quorum ($2f+1$ replicas) of responses in return to form a prepare quorum certificate

(QC). No two prepare quorum certificates can exist in the same view. In the precommit phase, the leader broadcasts the prepare quorum certificate, and generates a precommit quorum certificate from the $2f+1$ replies. While this two-step process ensures safety, it does not guarantee liveness. As we describe in §II-B, a Byzantine leader could abuse this process to prevent termination by creating a *hidden lock problem*. Instead, HotStuff adds a third commit phase: the leader broadcasts a precommit certificate. At this point, replicas become *locked* on this quorum certificate: they will unlock only if presented with a QC of a higher view. Once the leader receives commit responses back, it is finally safe for all replicas to execute the operation. In this way, HotStuff sends and verifies only $O(n)$ messages per phase.

B. Livelessness in existing protocols

HotStuff, a linear protocol with optimistic responsiveness, guarantees liveness by achieving consensus in three phases rather than two. Not doing so can trigger an infinite sequence of view changes after GST due to honest leaders continually failing to make progress. Specifically, it is possible to create executions in which the view change never includes the most up-to-date locked proposal, causing honest replicas to reject valid commands, and thus trigger constant view changes and prevent progress. We illustrate this scenario using a two-phase variant of HotStuff as our example. HotStuff operates in the lock-commit paradigm where if replicas become *locked* on a proposal in view v, they will reject all proposals in a smaller view. To illustrate the potential issue, consider the following example in Figure 1.

A Byzantine primary, $R1$, equivocates in view 1, sending proposal u to itself, $R2$, and $R3$, but proposal v to $R4$. $R1$ generates a lock-certificate for u in view 1, and sends it to $R2$. $R1$ then fails to make further progress, triggering a view change. $R3$ becomes the new primary and receives responses from $R1, R3, R4$, all of whom contain no locks. It thus proposes a new command v. $R4$ votes in favor of the proposal. $R2$ rejects this command as it is locked on u, and $R1$ fails to reply. A view-change is once again triggered as the leader received fewer than $2f+1$ responses. At the last minute, $R1$ votes to accept, allowing $R3$ to create a lock-certificate. $R2$ is the next primary for view 3. It collects new view messages from $R1, R2, R4$, and proposes the lock certificate for u in view 1. The same scenario repeats itself. $R1$ does not reply, and $R3$ rejects the proposal as it is locked on a higher view (v for view 2). This triggers yet a new view change, after which $R1$ finally sends a reply, allowing $R2$ to create a lock-certificate for u in view 3. This sequence of events can repeat infinitely, even after GST, thus violating liveness.

The root cause of the problem comes from replicas rejecting proposals (by failing to respond) if they are locked on a higher view. They must do so as they may have helped form a quorum allowing that command to commit. Unlocking such replicas requires convincing them that this command did not in fact commit. As discussed in the background section, existing BFT protocols achieve that in several different ways, from adding an extra phase in HotStuff (which guarantees that a majority of honest replicas will learn about the lock before the command can commit), to sending all the locks as in PBFT and Fast-HotStuff. In this paper, we propose an alternative approach: we introduce a new cryptographic mechanism called a *no-commit proof* that allows the leader to convince a replica, with linear authenticator complexity, that its locked command was not committed. By sending this command as part of a view-change message, Wendy fully sidesteps the hidden lock problem without the need of an additional phase or quadratic cryptographic costs.

III. System Model

Wendy adopts the standard system model of existing BFT protocols.

Failure Model We assume a static set of $n = 3f+1$ replicas, of which at most f can be Byzantine faulty. Faulty replicas can deviate arbitrarily from their specification. Replicas send messages to each other in point-to-point, authenticated and reliable FIFO channels. A strong but static adversary can coordinate faulty replicas's actions but cannot break standard cryptographic primitives. Wendy adopts the partial synchrony [19] model: there is a known bound Δ and an unknown Global Stabilization Time (GST), such that after GST, all transmission between two replicas arrive within Δ.

Cryptographic building blocks In addition to the novel *aggregate signature scheme* that we describe in §IV, Wendy uses BLS-based multi-signatures [14], [21] for signing groups of identical messages sent by distinct replicas. In a BLS multi-signature scheme, there is a single public key ($pk = (pk_1, \ldots, pk_n)$) held by all replicas and separate private keys held by individual replicas. Each replica i uses its private key to sign a message m. Each partial signature for i is then aggregated and used to produce a digital signature for m that can only be verified if it was signed by all desired replicas. We make use of the following procedures for key generation, signing, aggregation and verification: BLS.KGen, BLS.SignShare, BLS.VerifyShare, BLS.Agg, BLS.VerifyAgg (see Algorithm 2 in the Appendix). As is standard, Wendy assumes authenticated channels and a cryptographically secure, collision-resistant hash function h which maps an arbitrary length input to a fixed length output (also referred to as message digest).

Rogue-key attacks Aggregate and multi-signature schemes are vulnerable to *rogue-key attacks* [22] where an adversary can maliciously choose their PK as a function of the honest players' public keys. All of the schemes we present prevent such attacks using *proofs-of-knowledge (PoKs)* or *proofs-of-possession (PoPs)* [22]. Importantly, although our algorithms verify these PoKs/PoPs during signature verification, in practice, this cost is easily avoided. Specifically, in our BFT setting, where the number of signers n is fixed, the PoKs/PoPs are only verified *once* when the system is bootstrapped.

Correctness Wendy is *safe* as long as no more than f replicas exhibit byzantine faults. By safe, we mean that no two honest replicas decide on a different sequence of commands. Wendy is live after GST. By live, we mean that any command proposed by a client will eventually be executed.

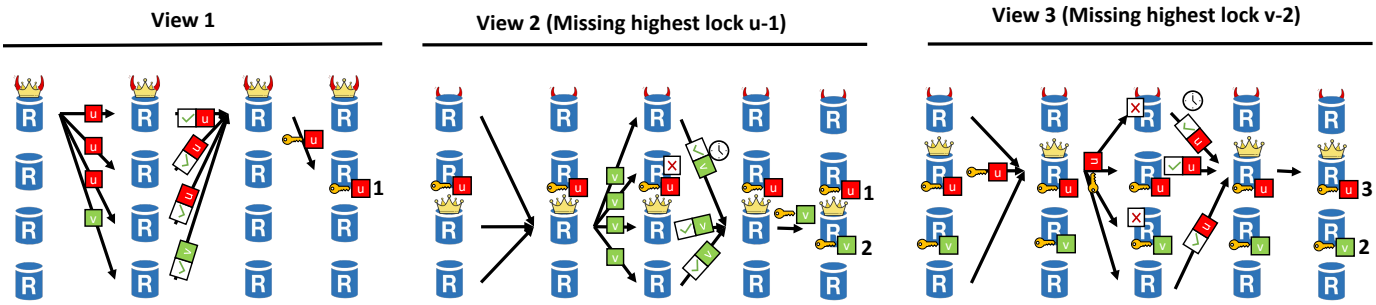


Fig. 1: Illustrating the hidden lock problem

IV. Designing proofs of no-commit

As described in §II, a new Byzantine leader must convince other replicas that the set of commands it includes in the view change is correct: a view-change must preserve all commands that could have been committed in prior views. This can be achieved by sending all view-change messages [5], making additional synchrony assumptions such that pending commands will eventually be received [7], or adding additional phases to address liveness issues when sending a subset of received view change messages [4]. Wendy sidesteps these trade-offs. It adopts the best of both worlds: it adopts HotStuff's view change optimistically but also uses a PBFT-like view change in the pessimistic case that conveys key information from all received view change messages (but in an efficient way that preserves linearity). Wendy's novel *No-commit proof* is the core technical innovation that enables this hybrid approach. It allows a leader to prove to replicas that a specific block did not commit without sending all view change messages naively.

In this section, we begin by formulating a simpler question: assume that a leader in view v receives signed view numbers from $2f+1$ replicas. How can it efficiently and provably convey these distinct values to an arbitrary replica r? By efficient, we mean that the replica must verify the authenticator in constant time and using a constant number of pairing operations. We first describe our construction informally (§IV-A) before sketching out its formal underpinnings (§IV-B). The new custom *aggregate signature scheme* that we develop represents a significant performance improvement over prior aggregate schemes such as BGLS aggregate signatures [23]. Its efficiency relies on three core assumptions: 1) replicas know the leader's view v (**A1**) 2) honest replicas will never send two different numbers for the same leader's view (**A2**) 3) the difference in view numbers is generally small (**A3**). These assumptions derive from properties of the Wendy protocol, which we describe next.

A. Intuition

Leveraging multi-signatures The most natural, inefficient option is to send all of the signed view numbers to a replica r. A similar approach is taken by PBFT, SBFT, and Jolteon/Ditto.

This naturally incurs linear verification cost for each replica. Multi-signatures or threshold signatures are the next obvious choice: they allow for replicas to partially sign a message, and for a leader to aggregate these partial signatures into a single signature.

Unfortunately, these signatures require that the *same* message be signed. We take a different approach: rather than encoding the view number in the *message* that we sign, we encode it in the *signature* of the message. For each replica, we generate ℓ BLS signature keys, one for each possible number that could be sent (we show how to reduce this next). We then have each replica sign the leader's view v with the corresponding signing key. If a replica wants to send view 5 for instance to the leader for view v_l, it uses its fifth signing key to sign v_l. The leader can then aggregate the partial signatures into a single signature, which replica r can efficiently verify using the matching public keys. This verification process requires only two pairings, which is significantly faster than prior approaches, which leverage BGLS aggregate signatures [23] (as we confirm in Figure 6 in our evaluation).

Reducing number of key pairs The above protocol allows replicas to efficiently verify a leader's claim. It is, however, clearly not practical as it would require generating an infinite number of key pairs to account for all possible view numbers. As the leader's view is known to all, we instead choose to encode the *difference* between the leader and each replica's view, c_i (this difference is bounded after GST). To further reduce overhead, we generate keys according to the *binary representation* of the resulting view difference. Specifically, we generate $2\log(v_d)$ keys, where v_d is the maximum assumed view difference: for each bit in v_d, we generate two keys, one for set bits, and one for zero-bits. Note that the replica's view difference could still be outside the maximum assumed target view difference. We consequently include a special signing key pair (sk_{v_d}, pk_{v_d}), which is used to sign the replica's view difference when it is outside this bound. Note that the resulting signature only conveys that the replica's view is outside the bound and not the specific view difference.

B. Formalisms

The above section informally introduced our approach. We now describe in more detail the formalism underpinning this novel aggregate signature scheme. We carefully model its security in Theorem 7 and prove it holds under *computational co-Diffie-Hellman (co-CDH)* [22] in Theorem 8. Algorithm 1 summarizes the pseudocode.

Algorithm 1 Our Wendy aggregate signature scheme

procedure $\mathsf{AS.KGen}(1^\lambda, \ell) \to (\mathsf{sk}, \mathsf{pk})$
  For $j \in [0,\ell), b \in \{0,1\}$, pick $\mathsf{sk}_{j,b} \in_R \mathbb{Z}_p$ uniformly at random
  For $j \in [0,\ell), b \in \{0,1\}$, set $\mathsf{pk}_{j,b} = g_2^{\mathsf{sk}_{j,b}}$
  Set $\pi_{j,b} = H_1(\mathsf{pk}_{j,b})^{\mathsf{sk}_{j,b}}$
  Set $\mathsf{sk} = (\mathsf{sk}_{j,b})_{j\in[0,\ell),b\in\{0,1\}}$
  Set $\mathsf{pk} = ((\mathsf{pk}_{j,b}, \pi_{j,b})_{j\in[0,\ell),b\in\{0,1\}})$
procedure $\mathsf{AS.SignShare}(\mathsf{sk}_i, m = (c_i|v)) \to \sigma_i$
  Let $c_{i,j}$ denote the jth bit of c_i
  Parse $\mathsf{sk}_i = (\mathsf{sk}_{i,j,b})_{j\in[0,\ell),b\in\{0,1\}}$
  Set $\sigma_i = \prod_{j\in[0,\ell)} H_0(v)^{\mathsf{sk}_{i,j,c_{i,j}}}$
procedure $\mathsf{AS.VerifyShare}(\mathsf{pk}_i, m = (c_i|v), \sigma_i) \to \{0,1\}$
  Let $c_{i,j}$ denote the jth bit of c_i
  Parse $\mathsf{pk}_i = (\mathsf{pk}_{i,j,b}, \pi_{i,j,b})_{j\in[0,\ell),b\in\{0,1\}}$
  ▷ PoP verification (only done during bootstrapping in Wendy)
  $\forall j \in [0,\ell), b \in \{0,1\}$, check if:
    $e(\pi_{i,j,b}, g_2) = e(H_1(\mathsf{pk}_{i,j,b}), \mathsf{pk}_{i,j,b})$
  Check if $e(\sigma_i, g_2) = e(H_0(v), \prod_{j\in[0,\ell)} \mathsf{pk}_{i,j,c_{i,j}})$
procedure $\mathsf{AS.Agg}((\sigma_i)_{i\in I}) \to \sigma$
  Let $\sigma = \prod_{i\in I} \sigma_i$
▷ Note: Assumes the m_i's have the same v, or returns 0.
procedure $\mathsf{AS.VerifyAgg}((\mathsf{pk}_i, m_i = (c_i|v))_{i\in I}, \sigma) \to \{0,1\}$
  Let $c_{i,j}$ denote the jth bit of c_i, for all $i \in I$
  Parse $\mathsf{pk}_i = (\mathsf{pk}_{i,j,b}, \pi_{i,j,b})_{j\in[0,\ell),b\in\{0,1\}}$, for all $i \in I$
  ▷ PoP verification (only done during bootstrapping in Wendy)
  $\forall i \in I, \forall j \in [0,\ell), b \in \{0,1\}$, check if:
    $e(\pi_{i,j,b}, g_2) = e(H_1(\mathsf{pk}_{i,j,b}), \mathsf{pk}_{i,j,b})$
  Check if $e(\sigma, g_2) = e(H_0(v), \prod_{i\in I}\prod_{j\in[0,\ell)} \mathsf{pk}_{i,j,c_{i,j}})$

Key generation. The ith replica's secret key will consist of 2ℓ different *secret subkeys* $(\mathsf{sk}_{i,0,b}, \mathsf{sk}_{i,1,b}, \ldots, \mathsf{sk}_{i,\ell-1,b})$ for each of the ℓ bits of a *difference* c_i, covering the possibility that the bit is $b = 0$ or $b = 1$. More formally, the ith replicas's key-pair $(\mathsf{sk}_i, \mathsf{pk}_i)$ consists of:

$$\mathsf{sk}_i = \left(\mathsf{sk}_{i,j,b}\right)_{j\in[0,\ell),b\in\{0,1\}} \in_R \mathbb{Z}_p^\ell \tag{1}$$

$$\mathsf{pk}_i = \left(\mathsf{pk}_{i,j,b}, \pi_{i,j,b}\right)_{j\in[0,\ell),b\in\{0,1\}} \in \mathbb{G}_2^\ell \tag{2}$$

where $\mathsf{pk}_{i,j,b} = g_2^{\mathsf{sk}_{i,j,b}}$ and $\pi_{i,j,b} = H_1(\mathsf{pk}_{i,j,b})^{\mathsf{sk}_{i,j,b}}$ is a proof-of-possesion (PoP) [22], to prevent rogue-key attacks.

Signing. Let $c_{i,j}$ be the jth bit of c_i. Note that $c_{i,j} \in \{0,1\}$, and $j \in [0,\ell]$. To sign $m_i = (c_i|v)$, replica i computes its *signature share*:

$$\sigma_i = H_0(v)^{\sum_{j\in[0,\ell)} \mathsf{sk}_{i,j,c_{i,j}}} \tag{3}$$

In effect, this is a multisignature on v, but with secret subkeys selected based on the bits of c_i. Note that two signature shares on v with different c_i' leads to forgeries, but our security definition from Theorem 7 discounts such attacks which are not problematic in our BFT setting.

Aggregation. To aggregate several *signature shares* $(\sigma_i)_{i\in I}$, each on a $(c_i|v)$, from a set of replicas I, we multiply them together. Below, $c_{i,j}$ denotes the jth bit of c_i.

$$\sigma = \prod_{i\in I} \sigma_i = \prod_{i\in I} H_0(v)^{\sum_{j\in[0,\ell)} \mathsf{sk}_{i,j,c_{i,j}}} \tag{4}$$

$$= H_0(v)^{\sum_{i\in I}\sum_{j\in[0,\ell)} \mathsf{sk}_{i,j,c_{i,j}}} \tag{5}$$

Verification. Our key observation is that verifying such an aggregated signature, only requires two pairings, rather than $|I|$. Recall that the verifier has all the signed messages $m_i = (c_i|v), \forall i \in I$. Again, let $c_{i,j}$ denote the jth bit of c_i. The verifier checks if:

$$e(\sigma, g_2) \stackrel{?}{=} e(H_0(v), \prod_{i\in I}\prod_{j\in[0,\ell)} \mathsf{pk}_{i,j,c_{i,j}}) \tag{6}$$

V. Wendy: a two-phase, linear protocol

We next use this novel aggregate signature scheme to design Wendy, a two-phase BFT protocol that achieves linear authenticator complexity and optimal latency while preserving optimistic responsiveness. Wendy is similar in spirit to Hot-Stuff but reduces the number of phases necessary to ensure liveness. Indeed, HotStuff requires three phases (optimal is two) to address the hidden lock problem illustrated in §II). Wendy instead uses no-commit proofs to convince skeptical replicas that it is safe to change their mind during a view change: a leader can now prove to a replica that its chosen command was not committed. A leader could pro-actively send this no-commit proof to replicas to always achieve consensus in two phases. However, Wendy further observes that the hidden lock problem is in fact exceedingly rare and thus does not send no-commit proofs by default. Instead, skeptical replicas must explicitly request such a proof when necessary, reducing cryptographic load on the system.

A. Terminology

Wendy adopts the *chaining model*, popular in new BFT protocols targeted towards blockchain systems [4], [7], [8]. Replicas reach consensus on a *chain*, which consists of a sequence of *blocks* containing one or more commands.

Blocks More formally, a block is defined as a tuple $b = (v, c, h)$ where v is the view in which the block is proposed, c is a batch of commands, and h is a hash pointer to its parent block. Each block has a greater view than its parent and the first block in a chain is represented as an empty *genesis block*.

Chains A chain is then represented as a tuple of three blocks: $c = (bCommit, bLock, bProposal)$. Each block is at a different stage of the consensus protocols: *bCommit* represents the last block in the chain to have been *committed*, *bLock* the last block in the chain which is locked (more on this later), while *bProposal* represents a block that is currently being proposed and the tail of the chain. These blocks extend each other: *bProposal* has a hash pointer to *bLock*, and *bLock* has a hash pointer to *bCommit*). It follows that $bCommit.v < bLock.v < bProposal.v$. We say that a chain C' *extends* C if there exists a sequence of hash parent pointers starting from $C'.bProposal$ that reach $C.bLock$.

Quorum Certificates (QC) A quorum certificate (QC) authenticates individual chains. It is also represented as a tuple $QC = (C, v, \sigma)$ where C is a chain, v is the view in which the QC was formed, and σ is an authenticator that consists of $2f+1$ signatures on chain C in view v.

We assume that replicas always verify any signed message that they receive and discard the message if the signature is invalid or buffer it if it is for a view greater than the replica's

current view. We omit explicitly mentioning these steps for clarity in later sections.

B. Protocol - Common case

Wendy consists of two phases: a *prepare phase* in which replicas agree on a block to process next, and a *commit phase* in which the choice of that command is persisted across failures. An optional *unlock phase* allows the leader to "unstuck" replicas who believe that an old block might have already committed. Each phase executes in a separate round or *view*. If progress stalls in a phase and a timeout expires, replicas invoke a *view-change* procedure to elect a new leader. The new primary then restarts the protocol beginning with the Prepare phase.

Each replica r maintains, as *local state*, its current view v_r, a quorum certificate $QC_r = (C, v, \sigma)$, where C is a chain, v is the view in which QC_r formed, and σ is an authenticator indicating that $2f+1$ replicas signed (C, v), and C_r, the latest proposed chain.

Phase 1: Prepare Phase Intuitively, the prepare phase ensures that at most one block of commands can be agreed on per view (non-equivocation).

1) Leader actions:

1: Leader receives $2f+1$ VOTE-RESP$\langle v-1, QC, C, \sigma_r\rangle$

Wendy operates in a chaining model. The start of the prepare phase for view v is thus overlapped with the completion of the preceding view $v-1$. The leader for view v thus waits to receive $2f+1$ matching VOTE-RESP$\langle v-1, QC_{parent}, C_{propose}, -\rangle$ ($QC_{parent} = QC, C_{propose} = C$) messages from the previous view. $C_{propose}$ refers to the chain being voted on in view $v-1$. The leader combines these responses using BLS multi-signature aggregation to form a QC authenticating the new latest chain $C_{propose}$. Specifically, the primary computes $QC_{propose} = (C_{propose}, v, \mathsf{BLS.Agg}\ (\sigma_r)_{r\in R})$, aggregating all signature shares σ_r and stores it locally, $QC_r = QC_{propose}$.

The leader then proposes a new block by appending that block to $C_{propose}$. If $QC_{propose}.v = QC_{parent}.v + 1$ (consecutive views) then it is safe to additionally commit $C_{propose}.bLock$ by creating a new chain $C_{new} = (C.bLock, C.bPropose, bPropose)$. Otherwise, the leader moves $C.bPropose$ into the $C.bLock$ position, sets $C.bPropose.h$ pointer to the hash of $C.bCommit$ (since $C.bLock$ did not satisfy the commit rule), and creates $C_{new} = (C.bCommit, C.bPropose, bPropose)$.

The leader then updates its state to store C_{new} locally ($C_r = C_{new}$) and broadcasts this new proposal to all replicas, containing its current view v, and its latest proposal C_{new} alongside $QC_{propose}$ indicating that C_{new} is a valid proposal. Intuitively, $C_{propose}$, which is now authenticated by $QC_{propose}$ becomes the parent chain, while C_{new} is the new chain. Sending $QC_{propose}$ with C_{new} proves that C_{new} does in fact extend the former $C_{propose}$.

2: Leader broadcasts VOTE-REQ$\langle v, QC_{propose}, C_{new}\rangle$ to all replicas

2) Replica actions:

1: Replica receives a VOTE-REQ$\langle v, QC_{propose}, C_{new}\rangle$

Replica r first validates $QC_{propose}$ by verifying the authenticator $QC_{propose}.\sigma$, and checking that C_{new} extends the $QC_{propose}.C$ chain. If these checks fail then the replica does not vote. It then checks whether C_{new} extends its current latest chain C_r. If true, r knows that it is safe to continue extending this chain and votes to support this proposal. Otherwise, r checks whether the chain's locked view $C_{new}.bLock.v$ is greater than its own $C_r.bLock.v$. This scenario indicates that C_r failed to commit due to a subsequent view change, r must consequently change its vote, moving to support C_{new}. In either case, r sends a VOTE-RESP$\langle -\rangle$ in support of C_{new} and updates its local state to $(v_r = v, QC_r = QC_{propose}, C_r = C_{new})$. r also computes $\sigma_r = \mathsf{BLS.SignShare}\ (C_{new}, v)$.

2: Replica sends VOTE-RESP$\langle v, QC_r, C_r, \sigma_r\rangle$ to the leader

Phase 2: Commit phase Recall that in chained BFT systems, phases are overlapped, the commit phase of C in view v and for block b is thus overlapped with the prepare phase of the next proposed block in view $v+1$. In other words, the commit phase of C aligns with Step 1 of the prepare phase described above. The leader in view $v+1$ combines the $2f+1$ received vote responses to form a quorum certificate $QC = (C, v, \sigma)$. The leader then proposes a new block $bPropose$ (as part of that block's prepare phase in view $v+1$) and creates a new chain C' that extends C. Once $2f+1$ vote responses have been received, b is deemed committed.

Execution: the asynchronous decide stage A block b is executed once 1) it has been committed and a commit certificate has been generated 2) all blocks that precede it in the chain have also been committed.

We illustrate the common case with a worked example in Figure 2. In view 1, replica 1 is the leader and proposes b_1. It thus creates a chain $C_{propose} = (\perp, \perp, b_1)$, as no prior blocks have been locked or committed. It sends this chain as part of a VOTE-REQ$\langle\rangle$ message, which all replicas accept. No replica is locked on a QC with a higher view. All replicas send VOTE-RESP$\langle\rangle$ messages to the leader of view 2, R_2, signing $C_{propose}$. R_2 aggregates these signature shares, stores C_r and forms a $QC_r = (C_{propose}, 1, \sigma) = ((\perp, \perp, b_1), 1, \sigma)$. R_2 then extends this chain. Block b_1 has been successfully locked, and is thus moved to the $bLock$ position in the chain, while $bPropose$ is set to b_2. The new chain C_r is thus equal to $C_r = (\perp, b_1, b_2)$. R_2 sends a VOTE-REQ$\langle\rangle$ to all, including C_r and QC_r. All replicas check that C_r extends $QC_r.C$ (which is the case here, as C_r contains b_1, b_2 and QC_r contains b_1) and that they are not currently locked on a QC with a higher view. Once again, they all vote to accept and send VOTE-RESP$\langle\rangle$ messages to R_3, the leader of view 3. This leader also creates a $QC_r = ((\perp, b_1, b_2), 2, \sigma)$, aggregating all signature shares. R_3 marks b_1 as committed and executes it, promoting

b_2 to locked, and proposing a new block b_3. R_3 thus proposes a new chain $C_r = (b_1, b_2, b_3)$. This process continues until a view change is triggered.

C. Protocol - View Change

We next describe the *view-change* logic. A view change is the process through which a leader is replaced. Its objective is to 1) elect a new leader 2) preserve all committed blocks.

1) Replica actions: A view timeout is associated with each view v and started when a replica enters view v as a non-primary. Upon detecting a timeout in view $v-1$, the replica r sends a NEWVIEW$\langle\rangle$ message to the leader of view v. The NEWVIEW$\langle\rangle$ message for the new view v contains the new target view v, the replica's local state and a signature share on the difference between the target view v and the latest QC's current view $QC_r.v$ ($\sigma_r = \mathsf{AS.SignShare}\ (v - QC_r.v, v)$). σ_r will be used by the leader to create a no-commit proof if a hidden lock is detected. Signing the difference between views rather than the current view itself is an efficiency optimization, key to reducing the verification cost of the view change (§IV-A).

1: Upon timeout for $v-1$, send NEWVIEW$\langle v, QC_r, \sigma_r\rangle$ to leader in v

2) Leader actions:

1: Leader receives $2f+1$ NEWVIEW$\langle v, QC_r, \sigma_r\rangle$ messages

The primary of view v waits for $2f+1$ NEWVIEW$\langle\rangle$ messages, validates that they are for target view v, verifies that $\mathsf{AS.VerifyShare}\ (pk_r, v - QC_r.v, \sigma_r) == 1$, and sets the view timeout for v.

The leader picks the NEWVIEW$\langle\rangle$ message with the highest locked view, $QC_r.v$, validates the $QC_r.\sigma$ authenticator and sets $QC_{propose} = QC_r$, and $C_{propose} = QC_r.C$. The leader then proposes $bPropose$, by appending that block to $C_{propose}$. Let QC_{parent} be the QC for chain $(-, C_{propose}.bCommit, C_{propose}.bLock)$. If $QC_{propose}.v = QC_{parent}.v + 1$ (consecutive views) then it can additionally commit $C_{propose}.bLock$ by creating a new chain $C_{new} = (C.bLock, C.bPropose, bPropose)$. Otherwise, the leader does not commit $C.bLock$, and the resulting new chain is $C_{new} = (C.bCommit, C.bPropose, bPropose)$. The primary updates its state to store C_{new} locally, $C_r = C_{new}$, and $v_r = v$. The primary broadcasts a VOTE-REQ$\langle v, QC_{propose}, C_{new}\rangle$ message to all replicas.

2: Leader broadcasts VOTE-REQ$\langle v, QC_{propose}, C_{new}\rangle$ to all replicas

3) Replica actions: Upon receiving a VOTE-REQ$\langle\rangle$ following a view change, the replica takes one of two options, depending on the view number that it is currently locked on (v_r). If C_{new} extends the replica's current chain C_r or $QC_{propose}.v > QC_r.v$ (higher locked view), the replica proceeds as in §V-B, step 2 and votes to support the leader's new proposal. Note that σ_r refers to the replica's own signature share on (C_r, v_r).

2A: Replica sends VOTE-RESP$\langle v, QC_r, C_r, \sigma_r\rangle$ to the leader

If instead the replica's stored chain C_r has a higher locked view ($C_r.bLock.v$) and C_{new} does not extend C_r, r cannot safely adopt the new chain. The existence of a lock certificate for C_r in a higher view means that C_r could have committed, with r's help. r thus rejects the proposal with a NACK$\langle-\rangle$ message containing its current state.

2B: Replica sends NACK$\langle v, QC_r\rangle$ to the leader

Sending a NACK$\langle\rangle$ message reveals the existence of a hidden lock (a lock for a higher view) that the leader was unaware of or maliciously omitted. This NACK$\langle\rangle$ message is in direct contrast with prior protocol's approach to simply ignore the message, as illustrated in our hidden lock example. We describe how the leader handles this negative response next.

D. Protocol - Optional unlocking

The Unlock phase is optional and only occurs when a replica informs the leader of a hidden lock. Even if a replica informs the leader of a hidden lock, the leader can proceed to the commit phase if it receives $2f+1$ matching VOTE-RESP$\langle v, -, -\rangle$ messages. Additionally, in the common case, the primary skips directly to the commit phase. In practice, hidden locks are rare. Wendy thus chooses to pay additional cryptographic cost only in the rare cases where a malicious leader has sufficiently created such a lock. This is in direct contrast to HotStuff, which always incurs the cost of an extra phase. Wendy further ensures, through its novel aggregate signature scheme, that recovering from this hidden lock is not prohibitively expensive. By comparison, similar schemes incur quadratic authenticator complexity [17], [18] or rely on still prohibitively expensive cryptography [9].

1) Leader actions:

3: Leader receives a NACK$\langle v, QC_i\rangle$ from replica i

The leader first validates that the NACK$\langle\rangle$ message contains a hidden lock by checking that its own $QC_r.v < QC_i.v$. As stated previously, the existence of a hidden lock QC_i is an indication that the chain $QC_i.C$ might have been committed. The leader must therefore convince the replica otherwise. We note that, if a chain C_i had been committed in view v, its associated quorum-certificate must *necessarily* be in the set of $2f+1$ NEWVIEW$\langle v, -, -, -\rangle$ messages. The reasoning is simple: committing an operation requires $2f+1$ quorum certificates, of which $f+1$ belong to honest nodes. The set of $2f+1$ NEWVIEW$\langle v, -, -, -\rangle$ necessarily intersects with at least one honest replica that contributed to committing the operation. It follows that, if no such lock-certificate is present, the operation did not commit. We leverage the novel aggregate signature

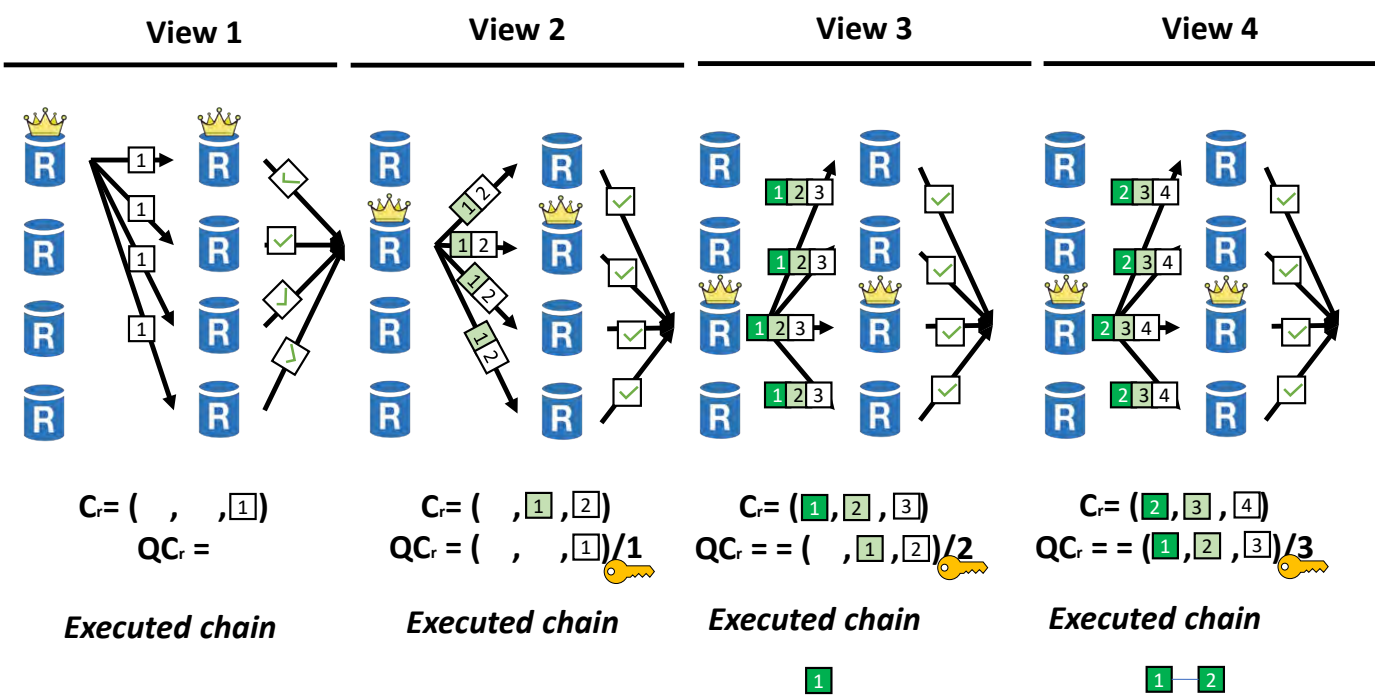


Fig. 2: Common-case execution

scheme developed in §IV-B to convey this information. As described above, each replica sends to the leader a signed signature share encoding the difference between the current view and its current locked view, $QC_r.v$. The leader then aggregates those signatures into a single no-commit proof $\sigma = \mathsf{AS.Agg}(\sigma_r)$. Note that each QC is unique per view, thus a signed view number uniquely identifies the signed QC itself.

4: Leader sends NO-COMMIT$\langle v, \sigma, M, QC_r, C_r\rangle$ to locked replica

The leader sends a NO-COMMIT$\langle - \rangle$ message containing its state (v, QC_r, C_r), the signed no-commit proof σ, and the list of $QC_r.v$ views for each replica, M.

2) Replica actions:

3: Replica receives a NO-COMMIT$\langle v, \sigma, M, QC, C\rangle$

The replica r extracts the set of views v_i from the list of views in M, verifies the signature σ and checks that no v_i is for a view greater than or equal to $QC_r.v$. Intuitively, this ensures that 1) the leader was honest and did in fact send the highest QC_r (if $QC_r.v > QC_i$ for some i, there would not have been any need to send a NACK$\langle\rangle$ 2) $QC_r.v$ was not included in any of the quorum certificates, thus confirming its associated chain did not commit. The replica then downgrades its lock and updates its state to (v, QC_r, C_r), and votes to support this new proposal.

4: Replica sends VOTE$\langle v, QC_r, C_r\rangle$ to the leader

The leader progresses as in step 2A and moves to the Commit phase once $2f+1$ VOTE-RSP$\langle\rangle$ messages have been received.

E. Safety

Chained Wendy makes the following safety guarantee:

Theorem 1. *No two non-faulty replicas can execute conflicting chains.*

An honest replica will only execute a chain C if it receives two QCs: QC on chain C, and $QC1$ on a chain $C1 = (C.bLock, C.bProposal, -)$, such that $QC.v + 1 == QC1.v$ (consecutive views). We define the *Lock-Certificate* of chain C to be QC. If C is executed, then define the *Commit-Certificate* of chain C to be $QC1$.

We first prove the following lemma:

Lemma 1: *If a QC forms in view v, then any other QC' with view $QC'.v = v$, must have the property that $QC'.C = QC.C$.*

Proof: Suppose for contradiction that $QC'.C \neq QC.C$, then $2f+1$ replicas voted for $QC'.C$, and $2f+1$ replicas voted for $QC.C$. These two quorums must intersect in at least 1 honest replica. This honest replica must have voted for both $QC'.C$ and $QC.C$, a contradiction.

We now prove safety by proving the following stronger theorem:

Theorem 2. *If there exists a Commit-Certificate in view v for chain C and a Lock-Certificate in view v' (where $v' \geq v$) for chain C' then C' must extend chain C.*

Proof: We prove by induction over the view v', starting at view v.

Base case: If $v' = v$ then $C = C'$ since there can only be at most one Lock-Certificate in view v. This is because each non-faulty replica can only vote for one chain in v, and each Lock-Certificate requires $2f+1$ votes (quorum intersection).

Inductive case: Assume this property holds for view v' from v to $v+k-1$ (inclusive) for some $k \geq 1$. We will consider the case that $v' = v+k$.

Suppose for the sake of contradiction that there exists a QC for chain C' in view $v+k$ that does not extend chain C. Since there exists a Commit-Certificate for chain C in view v, this implies that there are $2f+1$ replicas who had a QC_r such that $QC_r.v = v-1$ and $QC.C = C$. Thus if there is a QC' for chain C' in view $v+k$ then $2f+1$ must have voted for C' in view $v+k$, and this set must intersect the quorum of replicas that had a QC_r for view $v-1$ and chain C in at least one honest replica. This honest replica must have had that QC_r before it sent VOTE-RESP$\langle\rangle$ message for C' in view $v+k$. In the meantime, this honest replica may have updated its QC_r to be for a higher view. However all of these $QC.C$ must also extend C as (under the inductive hypothesis) any QCs for views from v to $v+k-1$ must be for a chain that extends chain C. This honest replica must therefore have a QC_r on a proposal that extended C from some view $v+j$ (where $0 \leq j < k$) before it sent a VOTE-RESP$\langle\rangle$ message for C' in view $v+k$. This honest replica could have voted after receiving a VOTE-REQ$\langle\rangle$ for C' in view $v+k$ or if not, then after it sent a NACK$\langle-\rangle$ message. We will consider each case in turn.

If the non-faulty replica voted for C' in view $v+k$ and did not send a NACK$\langle\rangle$ message, then C' must extend C as any QC with a view between $v+j$ and $v+k$ must extend C (under the induction hypothesis), and thus a contradiction.

Otherwise, if the honest replica did not vote in the Prepare phase (thus C' does not extend C) then the honest replica must have voted for C' during the unlock phase. This means that this honest replica sent a NACK$\langle\rangle$ message because its $QC_r.v > QC_{propose}.v$ and C' does not extend $QC_r.C$ in the VOTE-REQ$\langle\rangle$ message. We know that $QC_r.v \geq v$ from the commit on C, and know that all QCs in views $v+j$ (where $0 \leq j < k$ are for C, a contradiction, since C' cannot extend $QC_r.C$.

F. Liveness

For liveness we assume we have a pacemaker which satisfies the following synchronization property.

Property 1: There exists a bounded value t such that after GST, if view v has an honest leader then all honest replicas will enter view v.

Let Δ be the message delay, Λ be the view timeout duration, and $v_d = O(n)$ be the maximum view difference used for the No-Commit proofs. We now first prove the following theorem.

Theorem 3. *Let v_1, $v_2 = v_1+1$, and $v_3 = v_1+2$ be the first views that have honest leaders after GST, then a decision will be reached in a bounded duration, T_f after v_1 starts.*

Proof: By Property 1 from the pacemaker, we know that all honest replicas will enter view v within t time. We consider the two cases of when the new view v starts. A new view starts when either 1) the leader collects matching $2f+1$ VOTE-RESP$\langle\rangle$ messages from the previous view, $v-1$, or 2) the leader collects $2f+1$ NEWVIEW$\langle\rangle$ messages for view v with at least one QC_r such that $v - QC_r.v < v_d$ (within the max view difference range) or waits Δ for all NEWVIEW$\langle\rangle$ messages from honest replicas to be received. In case 1), these $2f+1$ VOTE-RESP$\langle\rangle$ messages can be used to form a QC for view $v-1$. This QC is guaranteed to be the highest since there can be at most one QC per view from *Lemma 1*. Since all honest replicas are synchronized in their view and the leader is honest, when the leader sends a VOTE-REQ$\langle\rangle$ message, it is guaranteed that all honest replicas will send a VOTE-RESP$\langle\rangle$ message since $v-1$ is the highest view in which a QC can form. For case 2), the leader collects $2f+1$ NEWVIEW$\langle\rangle$ messages and picks the QC with the highest $QC.v$. This QC is not guaranteed to be the highest an honest replica has since out of the $2f+1$ NEWVIEW$\langle\rangle$ messages only $f+1$ are guaranteed to be from honest replicas, unless the primary waited for Δ if there was no QC_r within the v_d max view difference range. Thus, when the leader broadcasts a VOTE-REQ$\langle\rangle$ message, not all honest replicas will send a VOTE-RESP$\langle\rangle$ message immediately. If an honest replica has a QC_r such that $QC_r.v > QC_{propose}.v$, then this honest replica will send a NACK$\langle\rangle$ message to the leader. The leader will then generate a No-Commit proof, since $QC_{propose}$ was the QC with the highest view from the NEWVIEW$\langle\rangle$ messages it received in the view change. Since we assume there is at least one NEWVIEW$\langle\rangle$ message with a QC_r such that $v - QC_r.v < v_d$, we are guaranteed that the replica that sent the NACK$\langle\rangle$ message has a QC with $QC.v > v - v_d$. There are two cases for each QC received in the view change: either 1) its view is $> v - v_d$ or 2) its view is $\leq v - v_d$. For case 1), the σ_r in the NEWVIEW$\langle\rangle$ message will be signed using the secret key encoding $QC_r.v$ while for case 2) the σ_r will be signed using the secret key encoding that the $QC_r.v$ is $\leq v - v_d$ without encoding $QC_r.v$ directly. Since the replica which sent the NACK$\langle\rangle$ message must have a QC with $QC.v > v - v_d$, this proof will convince the honest replica that its $QC_r.v$ is greater than all QCs sent in the view change. Thus all honest replicas that sent a NACK$\langle\rangle$ message will send a VOTE-RESP$\langle\rangle$ message for C_{new}. This means all honest replicas will send matching VOTE-RESP$\langle\rangle$ messages. With $2f+1$ VOTE-RESP$\langle\rangle$ messages, a QC in view v will form. This will also hold for view $v+1$ since the leader of $v+1$ is honest and the network is synchronous. A decision is reached after two consecutive QCs. Since the leader of view $v+2$ is honest, then similarly, during view $v+2$, all honest replicas will commit to the proposal from view v. The total time for this decision is bounded by the time to synchronize to v added to the time for 2.5 rounds to complete, which is $T_f = t + 2(2\Delta) + \Delta = t + 5\Delta$.

Theorem 4. *Let v be any view with an honest leader that is larger than v_2. Then after view synchronization to v, the leader will form a QC in v with a latency of $O(\delta)$, where δ is the actual network speed.*

Proof: The only step where the leader could explicitly wait Δ is when collecting $2f+1$ NEWVIEW$\langle\rangle$ messages for view

v. If the primary receives at least one NEWVIEW$\langle\rangle$ message containing a QC_r such that $QC_r.v > v - v_d$, then it does not wait Δ to receive NEWVIEW$\langle\rangle$ messages from all honest replicas. We show that this case always occurs by showing at least $f+1$ honest replicas satisfy this property.

Theorem 5. *Let v'' be a view with an honest leader $> v_2$, then there exists $f+1$ honest replicas with a $QC_r.v > v'' - v_d$*

We prove this theorem by induction.

Base Case. Let $v'' = v_2 + 1$, and $v''-2$, $v''-1$, and v'' have honest leaders. From *Theorem 3* we know that at least $f+1$ replicas have a $QC_r.v = v_2 > v'' - v_d$.

Let v_y be the yth view with an honest leader. We now assume that this theorem holds for all $v'' = v_{y-1}$.

Induction Step. Let $v'' = v_y$. If there was a commit in view $> v'' - v_d$, then we are guaranteed that $f+1$ honest replicas have a $QC_r.v > v'' - v_d$. Otherwise, we do not have a commit. Since $v_d = O(n)$, we are guaranteed that between $v - v_d$ and v there exists three consecutive views with honest leaders a the balls and bins argument. Namely given $n = 3f+1$ bins, no matter how f balls are thrown by the adversary, there must be three consecutive empty bins. Let these views be v',$v'+1$,$v'+2$. From the induction hypothesis, v' must have $f+1$ honest replicas that have a $QC_r.v > v' - v_d$ and therefore, from the same argument as *Theorem 3*, it will commit at network speed. Since there was a commit in view $v'+2$, there must be $f+1$ honest replicas with a $QC_r.v$ of at least $v'+1 > v_y - v_d$.

VI. IMPLEMENTATION

We implement Wendy in Go on top of an open-source implementation of HotStuff[1] using the Gorums [24] networking library. We use the Herumi Go bindings[2] for BLS signatures [14]. We use the Go *ecdsa*[3] implementation for digital signatures used for all messages besides view change messages, which require BLS multi-signatures. Note that like HotStuff, Wendy uses a list of *ecdsa* signatures for QCs. Wendy is multi-threaded with separate threads for networking, agreement, and cryptography (including signature verification).

VII. EVALUATION

In our evaluation, we answer three questions:

- How does Wendy's performance compare to HotStuff as a function of batch size, network latency, execution time and scale?
- How does Wendy's aggregate signature scheme compare to BGLS aggregate signatures?
- What is the performance of Wendy's view-change in the presence of Byzantine attacks?

A. Experimental Setup

We run our experiments on Cloudlab *m510* machines (8-core 2.0 GHz CPU, 64 GB RAM, 10 GB NIC). Each replica server is run on a separate machine, and up to 5 clients are run on the same machine. We increase the number of clients until the system saturates. Clients execute in a closed-loop, and each experiment runs for 60 seconds. We compare against the aforementioned HotStuff implementation. Unless otherwise stated, Wendy and HotStuff both run batches of no-op commands containing a zero-byte payload, and with $\Delta = 10s$. For local area (LAN) experiments, ping latency between machines is 0.2ms. WAN latencies are simulated for wide-area experiments using the tc [25] Linux command. Our WAN deployment models a setup with four regions: Europe, Virginia, Oregon, and China, using latency data from CloudPing [26].

B. Basic Performance

We evaluate the basic performance of Wendy with $f = 1$ failures in both a LAN and a WAN.

Local Area (LAN) We consider three different batch sizes, 100, 400, and 800, and report the throughput/latency results in Figure 3. We observe that Wendy and HotStuff exhibit similar throughput, with Wendy achieving at most a 3% increase in peak throughput independent of batch size. Pipelining amortizes the cost of consensus to 1 RTT per batch independently of the number of phases. The throughput for all protocols is thus simply a function of batch size divided by round latency. Round latency is identical in HotStuff and Wendy in the failure-free case; they consequently exhibit the same throughput. In contrast, Wendy's latency is on average 27.6% lower than HotStuff due to the fewer number of phases (Figure 3).

Wide-Area (WAN) Results in the wide-area follow an identical pattern to those in the LAN setup (Figure 4). Wendy and HotStuff exhibit identical throughput, but Wendy has 33% lower latency thanks to its two-phased protocol.

Execution Prior experiments considered no-ops only. We now measure the relative performance of Wendy and HotStuff with a simulated 0.25ms of execution time per command (Figure 5). This corresponds to an approximate average execution time of a simple smart contract for payment transfers [27]. Once again, we draw identical conclusions: Wendy and HotStuff exhibit identical throughput, with Wendy having 23.4% lower latency than HotStuff.

Scale Finally, we confirm that the aforementioned conclusions hold independently of scale. As f increases, throughput remains similar for both Wendy and HotStuff, while Wendy continues to have lower latency (Figure 5).

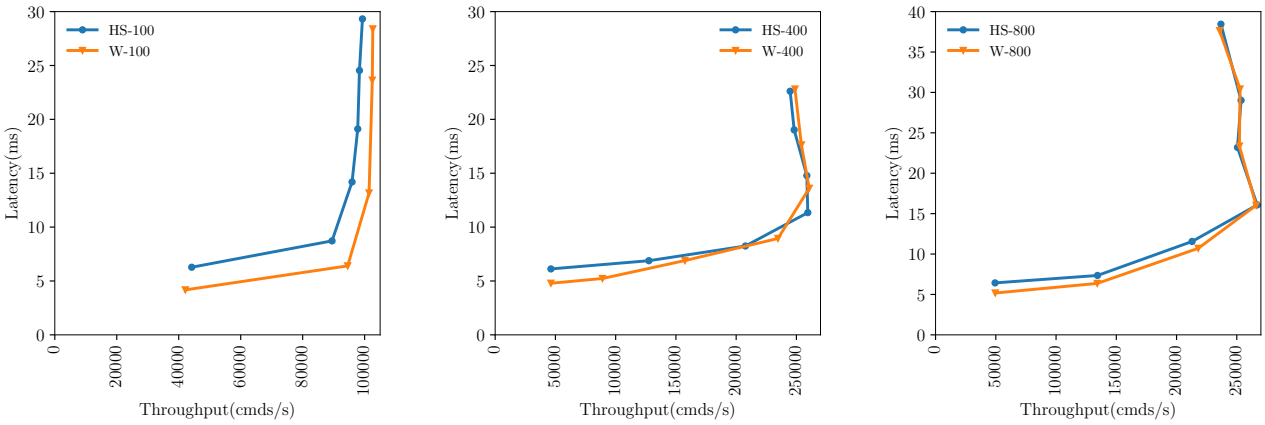
C. Aggregate Signature Cost

Prior experiments quantified the relative performance of Wendy and Hotstuff in the common case. We now look at Wendy's performance in the presence of view changes. Specifically, we look at the cost of Wendy's core technical contribution, its novel aggregate signature scheme, which lies at the core of its No-commit proof. We compare Wendy's aggregate signature scheme against a baseline of BGLS's aggregate signatures [23] (Figure 6).

[1]https://github.com/relab/hotstuff

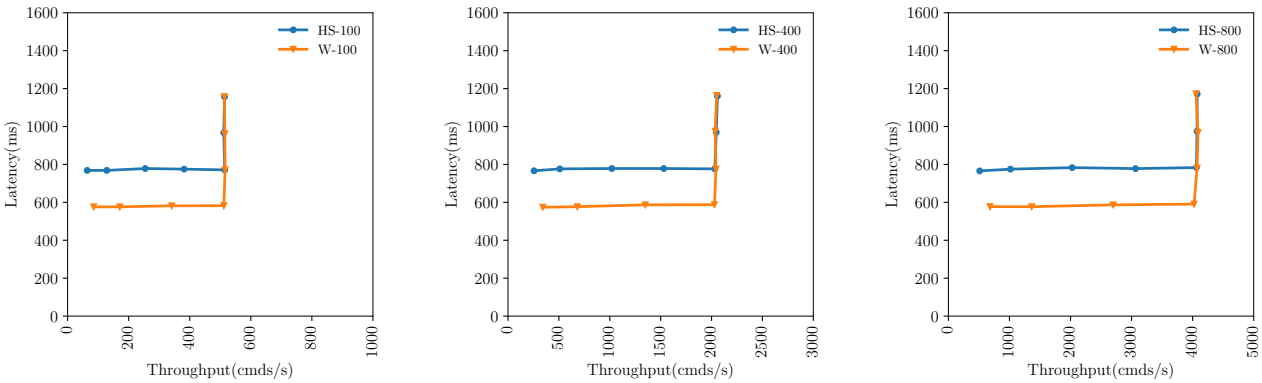
[2]https://github.com/herumi/bls-eth-go-binary

[3]https://github.com/golang/go/blob/master/src/crypto/ecdsa/ecdsa.go



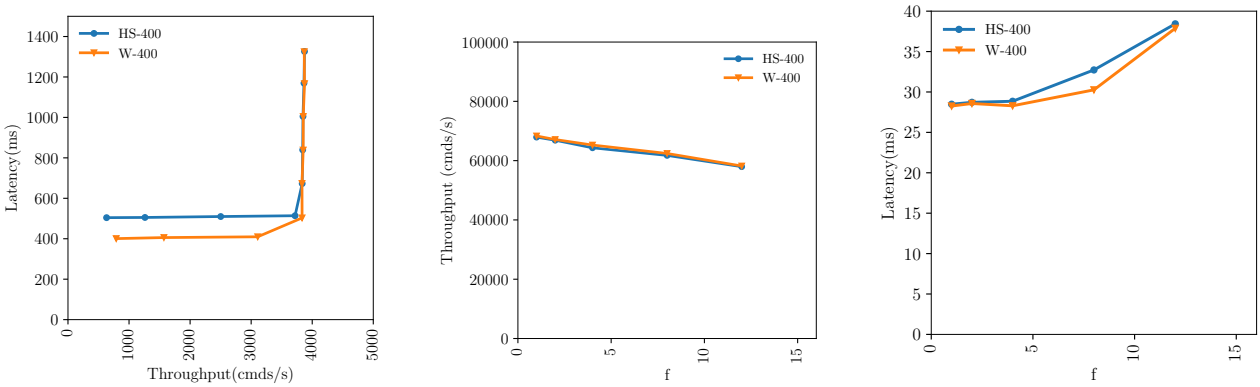
(a) LAN with batch size 100 (b) LAN with batch size 400 (c) LAN with batch size 800

Fig. 3: Impact of batch size (LAN)



(a) WAN with batch size 100 (b) WAN with batch size 400 (c) WAN with batch size 800

Fig. 4: Impact of batch size (WAN)

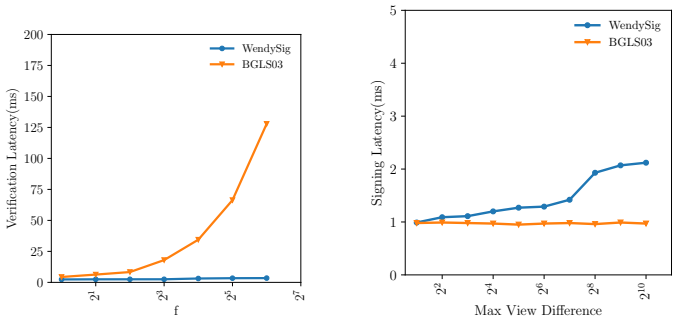


(a) Execution Time (b) Throughput vs f (c) Latency vs f

Fig. 5: Impact of execution time and of scale

Verification We first measure verification time as a function of f (a). With 193 replicas ($f = 64$), the latency of verification for our scheme is 3.44ms compared to 127ms in BGLS. This significant speedup stems from our use of multi-signatures internally, which incur $O(1)$ pairing cost, compared to $O(n)$ pairing cost in BGLS.

Signing We next quantify the relative latency of *signing* as a function of the maximum view difference v_d compared to BGLS (b). We fix the number of replicas to 193 ($f = 64$). We find that as the maximum view difference, v_d, increases, the number of signatures that need to be aggregated also increases by $\log v_d$. In contrast, BGLS is unaffected by the



(a) Verification Latency (b) Signing Latency

Fig. 6: Aggregate signature scheme performance

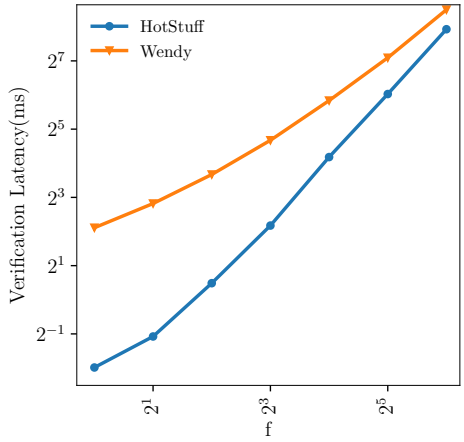


Fig. 7: View change performance

view difference: its signing cost is constant. Nonetheless, even for a large view difference (1024), Wendy has less than a millisecond higher signing latency than HotStuff. We find this to be a worthy trade-off given the 124ms speedup for signature verification.

D. Relative view change performance

The other experiments describe the performance benefits of our aggregate signature scheme over traditional BGLS signatures. For this experiment we quantify the performance of Wendy's view change relative to HotStuff (Figure 7), which does not send all locks, and instead simply sends the highest lock. We note that the default Go implementation of HotStuff makes use of ECDSA signatures rather than BLS multi-signatures. The cost of verifying a QC thus grows linearly as a function of f in HotStuff. We quantify the relative overhead of Wendy over HotStuff as a function of f. With $f = 1$, Wendy's overheads are high: 4.32ms compared to 0.25ms, a 6x increase in latency over HotStuff. This is because ECDSA signatures are significantly cheaper to verify than BLS multi-signatures. For smaller values of f, it is thus preferable to eschew linearity and send all locks. For larger values of f, Wendy's view change overheads go down significantly. For instance, for $f = 64$, Wendy's view change in the presence of a hidden lock has a latency of 362.09ms (compared to 243.46ms for HotStuff). Hidden locks are generally rare since it requires a very specific order of events to occur. Thus, our experiments show that at high values of f (the traditional setup for modern permissioned systems), Wendy can always achieve two-round trips in the common case and, in the rare cases where a hidden lock does exist, suffers only moderate overheads in its view change compared to HotStuff. In other words, Wendy's optimism pays off.

VIII. Conclusion

This paper introduced Wendy, a new pipelined BFT consensus protocol that achieves low-latency, linear authenticator complexity, and optimistic responsiveness. Wendy's main contribution is its innovative use of a linear *no-commit proof* that allows it to efficiently prove to replicas that it is safe to change their vote. Wendy's ability to scale and resilience to varying network conditions makes it well-suited for usage in modern permissioned blockchain systems.

References

[1] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, p. 382–401, Jul. 1982. [Online]. Available: https://doi.org/10.1145/357172.357176

[2] [Online]. Available: https://www.diem.com/en-us/white-paper/#cover-letter

[3] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Highly available transactions: Virtues and limitations," *Proc. VLDB Endow.*, vol. 7, no. 3, p. 181–192, Nov. 2013. [Online]. Available: https://doi.org/10.14778/2732232.2732237

[4] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "Hotstuff: Bft consensus with linearity and responsiveness," in *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, ser. PODC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 347–356. [Online]. Available: https://doi.org/10.1145/3293611.3331591

[5] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, ser. OSDI '99. USA: USENIX Association, 1999, p. 173–186.

[6] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative Byzantine Fault Tolerance," *ACM Transactions on Computer Systems (TOCS)*, vol. 27, no. 4, pp. 7:1–7:39, Jan. 2010. [Online]. Available: http://doi.acm.org/10.1145/1658357.1658358

[7] E. Buchman, "Tendermint: Byzantine fault tolerance in the age of blockchains," Ph.D. dissertation, 2016.

[8] V. Buterin and V. Griffith, "Casper the friendly finality gadget," *arXiv preprint arXiv:1710.09437*, 2017.

[9] M. Abspoel, T. Attema, and M. Rambaud, "Malicious security comes for free in consensus with leaders," Cryptology ePrint Archive, Report 2020/1480, 2020, https://ia.cr/2020/1480.

[10] Y. Zhang, S. Setty, Q. Chen, L. Zhou, and L. Alvisi, "Byzantine ordered consensus without byzantine oligarchy," pp. 633–649, Nov. 2020. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/zhang-yunhao

[11] I. Abraham and G. Stern, "Information Theoretic HotStuff," in *24th International Conference on Principles of Distributed Systems (OPODIS 2020)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), Q. Bramas, R. Oshman, and P. Romano, Eds., vol. 184. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2021, pp. 11:1–11:16. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2021/13496

[12] C. Cachin, "Yet another visit to paxos," IBM Research, Tech. Rep., 2011. [Online]. Available: https://cachin.com/cc/papers/pax.pdf

[13] G. Golan Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. Reiter, D.-A. Seredinschi, O. Tamir, and A. Tomescu, "Sbft: A scalable and decentralized trust infrastructure," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2019, pp. 568–580.

[14] D. Boneh, B. Lynn, and H. Shacham, "Short Signatures from the Weil Pairing," *Journal of Cryptology*, vol. 17, no. 4, pp. 297–319, Sep 2004. [Online]. Available: https://doi.org/10.1007/s00145-004-0314-9

[15] [Online]. Available: https://projectconcord.io/
[16] I. Abraham, G. Gueta, D. Malkhi, L. Alvisi, R. Kotla, and J. Martin, "Revisiting fast practical byzantine fault tolerance," *CoRR*, vol. abs/1712.01367, 2017. [Online]. Available: http://arxiv.org/abs/1712.01367
[17] M. M. Jalalzai, J. Niu, and C. Feng, "Fast-hotstuff: A fast and resilient hotstuff protocol," *CoRR*, vol. abs/2010.11454, 2020. [Online]. Available: https://arxiv.org/abs/2010.11454
[18] R. Gelashvili, L. Kokoris-Kogias, A. Sonnino, A. Spiegelman, and Z. Xiang, "Jolteon and ditto: Network-adaptive efficient consensus with asynchronous fallback," *CoRR*, vol. abs/2010.11454, 2021. [Online]. Available: https://arxiv.org/pdf/2106.10362.pdf
[19] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *Journal of the ACM (JACM)*, vol. 35, no. 2, pp. 288–323, 1988.
[20] R. Pass and E. Shi, "Thunderella: Blockchains with optimistic instant confirmation," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2018, pp. 3–33.
[21] A. Boldyreva, "Threshold Signatures, Multisignatures and Blind Signatures Based on the Gap-Diffie-Hellman-Group Signature Scheme," in *PKC 2003*, Y. G. Desmedt, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 31–46.
[22] D. Boneh, M. Drijvers, and G. Neven, "Compact Multi-Signatures for Smaller Blockchains," Cryptology ePrint Archive, Report 2018/483, 2018, https://eprint.iacr.org/2018/483.
[23] D. Boneh, C. Gentry, B. Lynn, and H. Shacham, "Aggregate and Verifiably Encrypted Signatures from Bilinear Maps," in *Advances in Cryptology — EUROCRYPT 2003*, E. Biham, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 416–432.
[24] T. Lea, L. Jehl, and H. Meling, "Towards new abstractions for implementing quorum-based systems," pp. 2380–2385, 06 2017.
[25] [Online]. Available: https://man7.org/linux/man-pages/man8/tc.8.html
[26] [Online]. Available: https://www.cloudping.co/grid
[27] D. Perez and B. Livshits, "Broken metre: Attacking resource metering in evm," *ArXiv*, vol. abs/1909.07220, 2020.
[28] D. Boneh, C. Gentry, B. Lynn, and H. Shacham, "Aggregate and Verifiably Encrypted Signatures from Bilinear Maps," Cryptology ePrint Archive, Report 2002/175, 2002, https://eprint.iacr.org/2002/175.

Algorithm 2 BLS multi-signature scheme

procedure $\mathsf{BLS.KGen}(1^\lambda) \to (\mathsf{sk}, (\mathsf{pk}, \pi))$
  Set $\mathsf{sk} \xleftarrow{\$} \mathbb{Z}_p$ uniformly at random
  Set $\mathsf{pk} = g_2^{\mathsf{sk}}$
  Set $\pi = H_1(\mathsf{pk})^{\mathsf{sk}}$
procedure $\mathsf{BLS.SignShare}(\mathsf{sk}_i, m) \to \sigma_i$
  Set $\sigma_i = H_0(m)^{\mathsf{sk}_i}$
procedure $\mathsf{BLS.VerifyShare}((\mathsf{pk}_i, \pi_i), m, \sigma_i) \to \{0,1\}$
  ▷ PoP verification (only done during bootstrapping in Wendy)
  Check if $e(\pi_i, g_2) = e(H_1(\mathsf{pk}_i), \mathsf{pk}_i)$
  Check if $e(\sigma_i, g_2) = e(H_0(m), \mathsf{pk}_i)$
procedure $\mathsf{BLS.Agg}((\sigma_i)_{i \in I}) \to \sigma$
  Set $\sigma = \prod_{i \in I} \sigma_i$
procedure $\mathsf{BLS.VerifyAgg}(((\mathsf{pk}_i, \pi_i), m_i)_{i \in I}, \sigma) \to \{0,1\}$
  ▷ PoP verification (only done during bootstrapping in Wendy)
  Check if $e(\pi_i, g_2) = e(H_1(\mathsf{pk}_i), \mathsf{pk}_i), \forall i \in I$
  Check if $e(\sigma, g_2) = e(H_0(v), \prod_{i \in I} \mathsf{pk}_i)$

APPENDIX

Our protocols rely on *Boneh-Lynn-Shacham (BLS) signatures* [14]. BLS signatures work in *bilinear groups* $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ of prime order p endowed with a pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$. Let g_2 denote the generator of $\mathbb{G}_2$.

The secret key in BLS is a random field element $s \in_R \mathbb{Z}_p$ and the public key is g_2^s. A random oracle $H : \{0,1\}^* \to \mathbb{G}_1$ is used for signing. To sign a message m, the signer computes $\sigma = H(m)^s$. To verify a signature on m given the public key g_2^s, the verifier checks if:

$$e(\sigma, g_2) = e(H(m), g_2^s) \tag{7}$$

Correctness follows from the bilinerity of the map e:

$$e(\sigma, g_2) = e(H(m), g_2^s) \Leftrightarrow \tag{8}$$

$$e(H(m)^s, g_2) = e(H(m), g_2^s) \Leftrightarrow \tag{9}$$

$$e(H(m), g_2)^s = e(H(m), g_2)^s \tag{10}$$

Unforgeability of signatures under chosen-message attack holds under the computational Diffie-Hellman (CDH) assumption in bilinear groups [14].

We also compare our aggregate signature scheme against normal, BGLS aggregate signatures [28]. We summarize this scheme in Algorithm 3, where H_0 denotes a random oracle that maps signed messages to $\mathbb{G}_1$ and H_1 denotes the oracle used in the proof of possession (PoP) [22], mapping public keys in $\mathbb{G}_2$ to a $\mathbb{G}_1$ element.

A. Security definition and proof

We first describe the generic notion of unforgeability for aggregate signatures from [23, Section 3.2] and then slightly modify it for our Wendy aggregate signature scheme.

Unforgeability for traditional aggregate signatures. To model security in a normal aggregate signature scheme such as BGLS [23], we require that an adversary $\mathcal{A}$, who controls the last $k-1$ of k different signers (i.e., picks their PKs, potentially knowing their corresponding SKs), cannot forge an aggregate signature on $(m_1, \ldots, m_k)$ under these k signers, even if $\mathcal{A}$ gets oracle access for signatures by the first signer. Of course, the

Algorithm 3 BGLS aggregate signature scheme

procedure $\mathsf{BGLS.KGen}(1^\lambda) \to (\mathsf{sk}, \mathsf{pk})$
  Set $\mathsf{sk} \xleftarrow{\$} \mathbb{Z}_p$ uniformly at random
  Set $\mathsf{pk} = g_2^{\mathsf{sk}}$
procedure $\mathsf{BGLS.SignShare}(\mathsf{sk}_i, m) \to \sigma_i$
  Set $\sigma_i = H_0(m)^{\mathsf{sk}_i}$
procedure $\mathsf{BGLS.VerifyShare}(\mathsf{pk}_i, m, \sigma_i) \to \{0,1\}$
  Check if $e(\sigma_i, g_2) = e(H_0(m), \mathsf{pk}_i)$
procedure $\mathsf{BGLS.Agg}((\sigma_i)_{i \in I}) \to \sigma$
  Set $\sigma = \prod_{i \in I} \sigma_i$
procedure $\mathsf{BGLS.VerifyAgg}((\mathsf{pk}_i, m_i)_{i \in I}, \sigma) \to \{0,1\}$
  Check if $e(\sigma, g_2) = \prod_{i \in I} e(H_0(m_i), \mathsf{pk}_i)$

forgery's first message m_1 cannot be queried to this signing oracle.

Definition 6. *We say an aggregate signature scheme is $(t, q_H, q_s, N, \varepsilon)$-secure in the aggregate chosen-key model if, for all adversaries $\mathcal{A}$ that run in time at most t, make at most q_H random oracle queries and at most q_s signing oracle queries, $\mathcal{A}$ wins $\mathsf{AggSigForge}_{\mathcal{A},N}(1^\lambda)$, defined below, with probability at most ε:*

$\underline{\mathsf{AggSigForge}_{\mathcal{A},N}(1^\lambda) \to \{0,1\}}$
$(\mathsf{sk}_1, \mathsf{pk}_1) \leftarrow \mathsf{AS.KGen}(1^\lambda)$
$(\sigma, \mathsf{pk}_2, \ldots, \mathsf{pk}_k, m_1, \ldots, m_k, k \le N) \leftarrow \mathcal{A}^{\mathsf{AS.SignShare}(\mathsf{sk}_1, \cdot)}(1^\lambda, \mathsf{pk}_1)$
Return 1 if:
  1. $\mathsf{AS.VerifyAgg}((pk_i, m_i)_{i \in [k]}, \sigma) = 1$
  2. $m_1 \notin Q$
Return 0 otherwise.

$\underline{\mathcal{O}^{\mathsf{AS.SignShare}(\mathsf{sk}_1, m)} \textit{ oracle}}$
Add m to set Q
Return $\mathsf{AS.SignShare}(\mathsf{sk}_1, m)$

Unforgeability for Wendy aggregate signatures. Our *Wendy aggregate signature* has the same notion of security as defined above, except the unforgeability game uses a slightly different signing oracle that does not allow double-signing on messages $(c|v)$ and $(c'|v)$ that share the same suffix v but have $c \neq c'$. This captures honest replicas not double signing in our application scenario.

Definition 7. *We say a Wendy aggregate signature scheme is $(t, q_H, q_s, N, \varepsilon)$-secure in the aggregate chosen-key model, if it satisfies Theorem 6 but with the modified signing oracle below:*

$\underline{\mathcal{O}^{\mathsf{AS.SignShare}(\mathsf{sk}_1, m=(c|v))} \textit{ oracle}}$
If $\nexists(\cdot|v)$ *in Q, then:*
  1. Add $(c|v)$ *to set Q*
  2. Return $\mathsf{AS.SignShare}(\mathsf{sk}_1, m)$

Observation: The constraint that the forged $m_1 = (c|v) \notin Q$ from Definition 6, together with the modified oracle above, guarantees that a successful forgery on $(c|v)$ is meaningful in our application. Specifically, that either:

1) v was **not** previously queried to the signing oracle for sk_1, or
2) v was queried before to the signing oracle for sk_1, but **not** with c.

Theorem 8. *The Wendy aggregate signature scheme from Algorithm 1 is secure as per Theorem 7 under the computational co-Diffie-Hellman (co-CDH) [22] assumption.*

Proof: Suppose $\mathcal{A}$ $(t, q_H, q_s, N, \varepsilon)$-breaks our Wendy aggregate signatures scheme with non-negligible probability. We show how to construct another adversary $\mathcal{B}$ that solves a random co-CDH instance with probability:

$$\varepsilon' \geq \frac{\varepsilon/\sqrt{e}}{2(q_s+N)} \tag{11}$$

This will contradict the hardness of co-CDH.

Specifically, $\mathcal{B}$ is given a co-CDH instance $(g_1, g_1^a, g_2, g_2^a, h) \in \mathbb{G}_1^2 \times \mathbb{G}_2^2 \times \mathbb{G}_1$ and must output $h^a \in \mathbb{G}_1$ to break co-CDH. $\mathcal{B}$ will interact with $\mathcal{A}$, acting as the challenger in the unforgeability game from Theorem 6 and simulating the (modified) signing oracle from Theorem 7. $\mathcal{B}$ also simulates the two random oracles H_0 and H_1 to $\mathcal{A}$, which are used for signing and for proofs of possesion (PoPs), respectively. $\mathcal{B}$'s goal is to turn a forgery from $\mathcal{A}$ into an h^a solution to its co-CDH instance.

Setup: $\mathcal{B}$ guesses that $\mathcal{A}$ will forge on a message $(c|v)$ whose first bit of c is $\hat{b} \in_R \{0,1\}$, picked at random. Next, $\mathcal{B}$ embeds the co-CDH challenge in the public key pk_1 of the first player. For this, $\mathcal{B}$ randomly picks $\mathsf{sk}_{1,j,b} \in_R \mathbb{Z}_p$ for all $j \in [1,\ell)$ and $b \in \{0,1\}$ but lets $\mathsf{sk}_{1,0,\hat{b}} = a$. Even though $\mathcal{B}$ does not know a, $\mathcal{B}$ can still compute all $\mathsf{pk}_{1,j,b} = g_2^{\mathsf{sk}_{1,j,b}}$ since he knows g_2^a. Therefore, pk_1 is distributed as if it was picked by the challenger, but actually embeds the co-CDH challenge.

$\mathcal{B}$ also computes PoPs $\pi_{i,j,b}$ for all the secret subkeys $\mathsf{sk}_{i,j,b}$. However, note that for $\mathsf{sk}_{1,0,\hat{b}} = a$, $\mathcal{B}$ does not actually know a, so $\mathcal{B}$ must simulate a PoP by programming the random oracle H_1. Specifically, on input $\mathsf{pk}_{1,0,\hat{b}}$, $\mathcal{B}$ programs H_1 to return g_1^z for random $z \in_R \mathbb{Z}_p$. This way, $\pi_{1,0,\hat{b}} = H_1(\mathsf{pk}_{1,0,\hat{b}})^{\mathsf{sk}_{1,0,\hat{b}}} = (g_1^z)^a = (g_1^a)^z$, which $\mathcal{B}$ can compute. $\mathcal{B}$ remembers this by adding $(\mathsf{pk}_{1,0,\hat{b}}, z)$ to a so-called H_1*-list*. (In fact, $\mathcal{B}$ will program H_1 even when computing PoPs for $\mathsf{sk}_{i,j,b}$'s it knows, since the oracle must always return a random value.)

Finally, $\mathcal{B}$ gives $\mathcal{A}$ the public parameter g_2 and the public key $\mathsf{pk}_1 = (\mathsf{pk}_{1,j,b}, \pi_{1,j,b})_{j \in [0,\ell), b \in \{0,1\}}$, on which $\mathcal{A}$ has oracle access.

H_1 hash queries: Recall that $\mathcal{A}$ can query the random oracle $H_1 : \mathbb{G}_2 \to \mathbb{G}_1$ on any inputs it pleases. $\mathcal{B}$ has already programmed this oracle in order to simulate PoPs for pk_1. In addition, $\mathcal{B}$ must also simulate random responses when queried by $\mathcal{A}$ and record them on its H_1-list defined above. When $\mathcal{A}$ queries H_1 on y, $\mathcal{B}$ responds as follows:

1) If y already appears on the H_1 list in some tuple (y,z), then $\mathcal{B}$ responds with $H_1(y) = h^z \in \mathbb{G}_1$, where h is part of the co-CDH challenge.
2) Otherwise, $\mathcal{B}$ picks a random $z \in \mathbb{Z}_p$, records (y,z) on the H_1-list and returns $H_1(y) = h^z$.

H_0 hash queries: Recall that $\mathcal{A}$ can query the random oracle $H_0 : \{0,1\}^* \to \mathbb{G}_1$ on the suffix v of the $(c|v)$ messages being signed. $\mathcal{B}$ will be responding to these queries by keeping an H_0*-list* of (v_i, w_i, f_i, r_i) tuples as follows. Initially, the H_0-list is empty. When $\mathcal{A}$ queries H_0 on v, $\mathcal{B}$ responds as follows:

1) If v already appears on the H_0-list in some tuple (v,w,f,r), then $\mathcal{B}$ responds with $H_0(v) = w \in \mathbb{G}_1$.
2) Otherwise, $\mathcal{B}$ flips a coin $f \in_R \{0,1\}$, so that $\Pr[f=0] = 1/(q_s+N)$.
   a) *Note:* When $f=1$, $\mathcal{B}$ is effectively guessing that $\mathcal{A}$ will *not* forge on this queried v.
3) $\mathcal{B}$ picks a random $r \in_R \mathbb{Z}_p$
4) $\mathcal{B}$ sets $w = h^{1-f} g_1^r$
5) $\mathcal{B}$ adds (v,w,f,r) to the H_0-list and returns $H_0(v) = w$ to $\mathcal{A}$

Importantly, whether f is 0 or 1, w is uniform in $\mathbb{G}_1$, and independent of $\mathcal{A}$'s *current view* (i.e., its input and received oracle outputs), due to the randomization of w by g_1^r.

Signature queries: Recall that $\mathcal{A}$ has oracle access to signatures under pk_1 (but through the modified oracle from Theorem 7). Let c_j denote the jth bit of c. On a request from $\mathcal{A}$ to sign $(c|v)$, $\mathcal{B}$ implements the signing oracle as follows:

1) $\mathcal{B}$ runs its hash-query algorithm on v and obtains the (v,w,f,r) tuple from the H_0-list.
2) If $[f=0$ and $c_0 = \hat{b}]$, then $\mathcal{B}$ fails (i.e., $\mathcal{B}$ incorrectly guessed that $\mathcal{A}$ will forge on this v; therefore, $\mathcal{B}$ would need to know $\mathsf{sk}_{1,0,\hat{b}} = a$ to simulate a signature on $(c|v)$, but $\mathcal{B}$ cannot since $\mathcal{B}$ does not know a).
3) Otherwise (i.e., $[f=1]$ or $[f=0$ and $c_0 = 1-\hat{b}]$), $\mathcal{B}$ can simulate a signature σ_1 on $(c|v)$, which is defined as:

$$\sigma_1 = H_0(v)^{\sum_{j \in [0,\ell)} \mathsf{sk}_{1,j,c_j}} \tag{12}$$

$$= w^{\sum_{j \in [0,\ell)} \mathsf{sk}_{1,j,c_j}} \tag{13}$$

   a) If $[f=0$ and $c_0 = 1-\hat{b}]$, then $w = h g_1^r$ and $\mathcal{B}$ knows all the $\mathsf{sk}_{1,j,b}$'s in order to compute σ_1

$$\sigma_1 = (h g_1^r)^{\sum_{j \in [0,\ell)} \mathsf{sk}_{1,j,c_j}} \tag{14}$$

$$= (h g_1^r)^{\mathsf{sk}_{1,0,1-\hat{b}}} (h g_1^r)^{\sum_{j \in [1,\ell)} \mathsf{sk}_{1,j,c_j}} \tag{15}$$

   b) Otherwise, if $[f=1]$, then $w = g_1^r$ and $\mathcal{B}$ knows r. Thus, independent of c's binary representation, $\mathcal{B}$ can always simulate as:

$$\sigma_1 = (g_1^r)^{\sum_{j \in [0,\ell)} \mathsf{sk}_{1,j,c_j}} \tag{16}$$

$$= \left(g_1^{\sum_{j \in [0,\ell)} \mathsf{sk}_{1,j,c_j}} \right)^r \tag{17}$$

$$= \left(\prod_{j \in [0,\ell)} \mathsf{pk}_{1,j,c_j} \right)^r \tag{18}$$

$\mathcal{A}$'s output: Having interacted with $\mathcal{B}$, requesting hashes and signatures via the oracles, $\mathcal{A}$ eventually halts. If $\mathcal{A}$ fails to produce a forgery, $\mathcal{B}$ fails to break co-CDH. Otherwise, if $\mathcal{A}$ succeeds, $\mathcal{A}$ returns an aggregated signature forgery on v of size $k \leq N$:

$$\sigma, \mathsf{pk}, \ldots, \mathsf{pk}_k, m_1 = (c_1|v), \ldots, m_k = (c_k|v) \tag{19}$$

Recall that:

$$\mathsf{pk}_i = ((\mathsf{pk}_{i,j,b}, \pi_{i,j,b})_{j\in[0,\ell),b\in\{0,1\}} \tag{20}$$

Also, recall that each $\mathsf{pk}_{i,j,b} = g_2^{\mathsf{sk}_{i,j,b}}$. Importantly, $\mathcal{A}$ either never requested a signature on the v above or, if it did, the signature was requested with a c different from the c_1 above. Since $\mathcal{A}$ succeeded, $\mathcal{B}$ makes sure there is an entry (v,w,f,r) for v on the H_0-list by running an oracle query for v, if need be.

If $f=1$, then $\mathcal{B}$ guessed incorrectly and fails to break co-CDH. Otherwise (i.e., $f=0$), we have two cases. Let $c_{i,j}$ denote the jth bit of c_i. Recall that $\mathcal{B}$'s guess for the first bit $c_{1,0}$ of c_1 was $\hat{b}$. In the first case, $c_{1,0}\neq\hat{b}$ and $\mathcal{B}$ fails to break co-CDH. In the second case, $c_{1,0}=\hat{b}$, and $\mathcal{B}$ can break co-CDH as follows.

Recall that, since $f=0$, we have $H_0(v)=w=hg_1^r$. Since σ is a valid forgery, we have:

$$e(\sigma,g_2) = e(H_0(v), \prod_{i\in[1,k]}\prod_{j\in[0,\ell)}\mathsf{pk}_{i,j,c_{i,j}}) \tag{21}$$

$$= e(w, \prod_{i\in[1,k]}\prod_{j\in[0,\ell)}\mathsf{pk}_{i,j,c_{i,j}}) \tag{22}$$

$$= e(hg_1^r, \prod_{i\in[1,k]}\prod_{j\in[0,\ell)}\mathsf{pk}_{i,j,c_{i,j}}) \tag{23}$$

Let $\sigma_{i,j} = (hg_1^r)^{\mathsf{sk}_{i,j,c_{i,j}}}$ and $\sigma_i = \prod_{j\in[0,\ell)}\sigma_{i,j}$. Note that if $\mathcal{B}$ can obtain $\sigma_{1,0} = (hg_1^r)^{\mathsf{sk}_{i,0,c_{i,0}}} = (hg_1^r)^{\mathsf{sk}_{i,0,\hat{b}}} = (hg_1^r)^a$ he can output the co-CDH solution $h^a = (\sigma_{1,0})/(g_1^a)^r$, since $\mathcal{B}$ knows g_1^a and r.

Rewrite Eq. (23) as:

$$e(\sigma,g_2) = \prod_{i\in[1,k]}\prod_{j\in[0,\ell)} e((hg_1^r)^{\mathsf{sk}_{i,j,c_{i,j}}}, g_2) \tag{24}$$

$$= \prod_{i\in[1,k]}\prod_{j\in[0,\ell)} e(\sigma_{i,j},g_2) = \prod_{i\in[1,k]} e\left(\prod_{j\in[0,\ell)}\sigma_{i,j}, g_2\right) \tag{25}$$

$$= \prod_{i\in[1,k]} e(\sigma_i,g_2) \tag{26}$$

As a result, observe that:

$$\sigma_1 = \sigma/(\prod_{i\in[2,k]}\sigma_i) \tag{27}$$

Later on, we show how $\mathcal{B}$ can recover all the other signature shares σ_i from players $i\in[2,k]$ using their proofs of possession (PoP) from pk_i. This way, $\mathcal{B}$ can recover σ_1 as per Eq. (27). Since $\mathcal{B}$ randomly picked all the $\mathsf{sk}_{1,j,b}$ secret subkeys when $j\geq 1$, $\mathcal{B}$ can also recover $\sigma_{1,0} = \sigma_1/\prod_{j\in[1,\ell)}\sigma_{1,j}$. Thus, $\mathcal{B}$ can break co-CDH by outputting $h^a = (\sigma_{1,0})/(g_1^a)^r$.

Analyzing $\mathcal{B}$'s success probability: For $\mathcal{B}$ to successfully break co-CDH, the following three events must occur:

- E_1: $\mathcal{B}$ does not fail as a result of any of $\mathcal{A}$'s signature queries.
- E_2: $\mathcal{A}$ generates a valid signature forgery $(k,\mathsf{pk}_2,\ldots,\mathsf{pk}_k, m_1,\ldots,m_k)$, where $m_i=(c_i|v)$.
- E_3: Event E_2 occurs and $f=0$ in the H_0-list entry (v,w,f,r) for v, with $c_{1,0}=\hat{b}$ (In other words, $\mathcal{B}$ can use the forgery to break co-CDH.)

$\mathcal{B}$ succeeds if all events above happen:

$$\Pr[E_1\wedge E_2\wedge E_3] = \Pr[E_1]\cdot\Pr[E_2 \mid E_1]\cdot\Pr[E_3 \mid E_1\wedge E_2] \tag{28}$$

We analyze the probabilities above one by one. Let e denote the base of the natural logarithm. (This overloads e, which also denotes a pairing, but meaning should be clear from context.)

Claim 1: $\Pr[E_1]\geq 1/\sqrt{e}$. Recall that $\mathcal{A}$ makes q_s signature queries and that, by Theorem 7, $\mathcal{A}$ never queries the signing oracle on the same v twice. We prove by induction that the probability that $\mathcal{B}$ does not fail after the first i signature queries is at least $\left(1-\frac{1}{2(q_s+N)}\right)^i \geq 1/\sqrt{e}$.

First, note that the probability $\mathcal{B}$ does not fail on the ith query is independent from the probability on the previous queries and from $\mathcal{A}$'s current view. In other words, the probability only depends on $\mathcal{B}$'s coin flips. After $i=0$ queries, clearly $\mathcal{B}$ does not fail with probability 1, which is $\geq \left(1-\frac{1}{2(q_s+N)}\right)^0$. Assume inductively that after $i-1$ queries the probability is $\left(1-\frac{1}{2(q_s+N)}\right)^{i-1}$. Then, at the ith query on (c,v), the probability that $\mathcal{B}$ fails is at most the probability that $f=0$ and $c_{1,0}=\hat{b}$:

$$\Pr[\mathcal{B}\text{ fails on query }i] \leq \Pr[f=0\wedge c_{1,0}=\hat{b}] = \tag{29}$$

$$= \frac{1}{q_s+N}\cdot\frac{1}{2} = \frac{1}{2(q_s+N)} \tag{30}$$

Thus, the probability that $\mathcal{B}$ does not fail on the ith signature query is:

$$\Pr[\mathcal{B}\text{ succeeds on query }i] \geq 1-\frac{1}{2(q_s+N)} \tag{31}$$

Therefore, by the inductive hypothesis, the probability that $\mathcal{B}$ does not fail after the first i signature queries is:

$$\Pr[\mathcal{B}\text{ succeeds on queries }1,2,\ldots,i-1]\cdot \tag{32}$$

$$\cdot\ \Pr[\mathcal{B}\text{ succeeds on query }i] \geq \tag{33}$$

$$\geq \left(1-\frac{1}{2(q_s+N)}\right)^{i-1}\cdot\left(1-\frac{1}{2(q_s+N)}\right) = \tag{34}$$

$$= \left(1-\frac{1}{2(q_s+N)}\right)^{i} \tag{35}$$

Since $\lim_{q_s\to\infty}\left(1-\frac{1}{2(q_s+N)}\right)^{q_s} = 1/\sqrt{e}$, this probability is $\geq 1/\sqrt{e}$.

Claim 2: $\Pr[E_2 \mid E_1]\geq\varepsilon$. The adversary $\mathcal{A}$ gets a public key pk_1 from the same distribution as public keys produced by $\mathsf{AS.KGen}$, since the co-CDH challenge g_2^a is uniform in $\mathbb{G}_2$, since a is uniform in $\mathbb{Z}_p$. The responses w_i on the ith random oracle query are also uniform in $\mathbb{G}_1$ since they are computed as $w_i = h^{1-f_i}g_1^{r_i}$ where r_i is uniform in $\mathbb{Z}_p$. Since E_1 happened (i.e., $\mathcal{B}$ succeeded in simulating all q_s signatures to $\mathcal{A}$), the probability $\Pr[E_2 \mid E_1]$ that $\mathcal{A}$ outputs a forgery is at least ε. This is because, in the beginning of this proof, we assumed $\mathcal{A}$ $(t,q_H,q_s,N,\varepsilon)$-breaks the Wendy aggregate signature scheme.

Claim 3: $\Pr[E_3 \mid E_2\wedge E_1]\geq\frac{1}{2(q_s+N)}$*:* Recall this is the probability that $\mathcal{B}$ can use the forgery on $m_1=(c_1|v)$ to break co-CDH. In other words, the probability that (1) $f=0$ in

the H_0-list entry (v,w,f,r) for the forged v and (2) $c_{1,0} = \hat{b}$. From Eq. (29), we already know that this probability is $\Pr[f = 0 \wedge c_{1,0} = \hat{b}] = \frac{1}{2(q_s+N)}$.

Recall that in order to break co-CDH, $\mathcal{B}$ needs to compute all $\sigma_i, i \in [2,k]$ from Eq. (27). Recall that each adversarially-generated $\mathsf{pk}_i = (\mathsf{pk}_{i,j,b})_{j\in[0,\ell),b\in\{0,1\}}$ comes with a proof-of-possesion (PoP) $\pi_i = (\pi_{i,j,b})_{j\in[0,\ell),b\in\{0,1\}}$, where $\pi_{i,j,b} = H_1(\mathsf{pk}_{i,j,b})^{\mathsf{sk}_{i,j,b}}$. Also, recall that $\mathcal{B}$ programmed $H_1(y)$ to return h^z, for random $z \in \mathbb{Z}_p$, and recorded (y,z) in its H_1-list. Thus, since $\mathcal{B}$ has all the $\pi_{i,j,b}$ PoPs, $\mathcal{B}$ actually has all the $(h^{z_{i,j,b}})^{\mathsf{sk}_{i,j,b}}$. $\mathcal{B}$ can thus exponentiate by $(z_{i,j,b})^{-1}$ and obtain all $h^{\mathsf{sk}_{i,j,b}}$'s. Once it has these, recall that:

$$\sigma_{i,j} = (hg_1^r)^{\mathsf{sk}_{i,j,c_{i,j}}} \tag{36}$$

$$\sigma_i = \prod_{j\in[0,\ell)} \tag{37}$$

Thus, $\mathcal{B}$ can compute the $\sigma_{i,j}$'s from the $h^{\mathsf{sk}_{i,j,b}}$'s. Next, $\mathcal{B}$ computes all $\sigma_i, i \in [2,k]$ and, lastly, σ_1 as per Eq. (27).

Therefore, $\mathcal{B}$'s success probability here is $\Pr[E_3 \mid E_2 \wedge E_1] \geq \frac{1}{2(q_s+N)}$.

$\mathcal{B}$'s final success probability: To summarize, $\mathcal{B}$'s probability of breaking co-CDH is:

$$\Pr[E_1 \wedge E_2 \wedge E_3] = \Pr[E_1] \cdot \Pr[E_2 \mid E_1] \cdot \Pr[E_3 \mid E_1 \wedge E_2] \tag{38}$$

$$\geq \frac{1}{\sqrt{e}} \cdot \varepsilon \cdot \frac{1}{2(q_s+N)} \tag{39}$$

$$= \frac{\varepsilon/\sqrt{e}}{2(q_s+N)} \tag{40}$$

∎