Remote Staking with Economic Safety

ABSTRACT

Proof-of-stake (PoS) blockchains require validators to lock their tokens as collateral, slashing these tokens if they are identified as protocol violators. PoS chains have mostly been secured by their *native* tokens. However, using only the native token upper-bounds the value eligible for staking by the market capitalization of the native token. In contrast, the *remote staking* of another crypto asset from a *provider chain* provides an avenue to improve the *consumer chain*'s economic security. In this paper, we present the first known remote staking protocols with guaranteed *optimal economic safety*: whenever there is a safety violation on the consumer chain, at least one third of the provider's stake securing the consumer chain is slashed. To achieve this goal for a broad range of provider and consumer chains, two independent contributions are made: 1) a remote unbonding protocol that ensures slashing before the stake is unbonded on the provider chain if there is safety violation on the consumer chain; 2) a protocol to slash stake even without smart contracts on the provider chain. The remote staking protocol is analyzed and implemented in the case where the provider chain is Bitcoin and the consumer chain is a Cosmos SDK chain running the Tendermint consensus protocol.

1 INTRODUCTION

1.1 Proof-of-stake Security

A major trend of the blockchain ecosystem in the past few years is the shift from proof-of-work (PoW) to proof-of-stake (PoS) based sybil resistance mechanisms, as in Ethereum's 'Merge' in 2022. Besides lower energy usage, PoS blockchains provide the potential to provably hold validators accountable and slash their stake when they violate the protocol. Indeed, *accountable safety* [16], *i.e.* the ability to identify the adversarial validators whenever there is a safety violation, was the main motivation behind Ethereum's migration to PoS [8]. It is also a central tenet of Tendermint [13, 15], a widely used consensus protocol for building PoS blockchains (*e.g.*, Polygon, BNB Chain and over 60 application-specific chains in the Cosmos ecosystem). With accountable safety, the staked assets can be viewed as *collateral* to provide economic security to a PoS blockchain. The larger the amount of staked assets on a chain, the higher the economic security.

1.2 Native vs Remote Staking

PoS blockchains are typically secured by the native assets maintained on the blockchain. For example, Ethereum is secured by ETH, Cosmos Hub is secured by ATOM, and BNB Chain is secured by BNB. However, using only the native token upper-bounds the economic security of the PoS chain by the market capitalization of the token. Staking of *remote* crypto assets *instead of* or in addition to the native assets provides an avenue to improve the chain's security by increasing the total staked value.

One approach is to first *bridge* the remote asset from its chain, the provider chain, and then to use them to secure the PoS chain, the consumer chain. However, this is subject to the security risks and capacity limitations of the existing bridging solutions such as the use of trusted third parties [3] and sidechains [6, 38, 43] with security vulnerabilities and the requirement of over-collateralized vaults [7, 28].

Another emerging approach in the blockchain industry is *remote staking*: the staked foreign assets stay on the provider chain, but is locked in a smart bond contract designated for a preferred validator of the consumer chain. This asset is slashed only if the validator commits slashable offenses in its execution of the secured protocol on the consumer chain. Remote staking was proposed to realize the concept of *mesh security* for the Cosmos ecosystem [4, 9], where the assets of one Cosmos chain are *remote-staked* to help secure another Cosmos chain. This protocol was in turn inspired by Eigenlayer's Ethereum *restaking* concept [49], which uses the staked ETH on Ethereum as collateral to secure middleware such as bridges, data availability and oracle services. With Cosmos mesh security and Ethereum restaking, a generalized form of remote staking is emerging, where a crypto asset can be used to secure chains and services other than its own chain.

1.3 Contributions

In this paper, we present the first known remote staking protocols with *optimal economic safety*: whenever there is a safety violation on the consumer chain, at least one third of the provider's stake securing the consumer chain is slashed [1]. Towards this goal, two independent contributions are made:

- **Secure unbonding:** In standalone PoS chains with native staking, enforcing the slashing of adversarial validators is known to be impossible without an external source of trust [48]. As in long range posterior corruption attacks, the adversarial validators can unbond before they can be slashed. [48] proposed using a separate provider chain as a secure timestamping server for checkpointing the consumer chain to achieve secure unbonding. As remote staking already incorporates a provider chain, we use an analogous timestamping protocol to help identify the adversarial validators before they unbond. However, in remote staking, the separation of the chain tracking the stake (provider) and the chain validated by the staked entities (consumer) introduces challenges unique to the remote staking design, such as signalling the validator set changes happening on the provider chain to the consumer chain and ensuring that the adversarial validators cannot fork the consumer chain after removing their stake from the provider chain.
- **Dumb contracts for slashing:** Provider chains like Ethereum support Turing-complete smart contracts, which makes the implementation of slashing on the provider chain technically straightforward once the adversarial validators are identified. However, there are other chains that do not support smart contracts, the most important example being Bitcoin, an asset worth over 1.3 trillion USD as of 15 April, 2024. Motivated by

Authors are listed alphabetically.

[1] Note that economic safety is stronger than accountable safety: not only adversarial validators are *identified*, but they are actually *slashed* by the protocol.

the Bitcoin use case, the second main contribution of this work is a secure remote staking protocol that does *not* require Turing-complete smart contract capability on the provider chain and can provide security to a consumer chain running any consensus protocol.

The first contribution is relevant to all remote staking use cases, regardless of whether the provider chain has a smart contract layer or not. This contribution is highlighted in Section 4, where we present a remote staking protocol with smart contract slashing on the provider chain. The second contribution is specific to the case when the provider chain has no smart contract layer. This contribution is highlighted in Section 5, where we present a remote staking protocol for the case Bitcoin is the provider chain. In this setting, we will show how slashing can be implemented using today's Bitcoin scripting language (in particular, only multi-signatures and timelocks) *without* any upgrade of the existing Bitcoin protocol. Our construction is *modular*, applicable to a broad range of consensus protocols on the consumer chain, although for concreteness, we will focus on the Tendermint protocol for our analysis and implementation.

In this paper, for simplicity we focus on the case when the consumer chain is *entirely* secured by assets on the remote provider chain. Variation of the design incorporating both the remote and the native asset in securing the consumer chain (dual staking) is possible but is beyond the scope of this paper.

1.4 Security Properties

For economic safety, a PoS blockchain must not only identify the adversarial validators responsible for a safety violation, but also slash their stake afterwards. To capture this requirement, we strengthen *accountable safety* [16, 47], *i.e.*, the ability to identify adversarial validators after a safety violation, to *economic safety*, the ability to *slash* the provider chain tokens staked by the adversarial validators after a safety violation (*cf.* Def. 4). We say that the protocol satisfies $1/3$-economic safety if, when there are n validators in total, $\lfloor n/3 \rfloor$ adversarial validators can be slashed after a safety violation, regardless of the total number of adversarial validators.

Theorem 1 (Informal, *cf.* Theorems 3, 8, 9). *Assuming the security of the provider chain (*e.g.*, Bitcoin), both remote staking protocols satisfy $1/3$-economic safety.*

Theorem 2 (Informal, *cf.* Theorems 4, 7, 10). *Assuming the security of the provider chain (*e.g.*, Bitcoin), both remote staking protocols satisfy liveness with finite latency, if the fraction of adversarial validators is less than $1/3$ at all times.*

A remote staking protocol satisfying $1/3$-economic safety means that no matter how many adversarial validators there are, at least $1/3$ of them is guaranteed to be slashed after a safety violation. On the other hand, no PoS blockchain secured only by its native stake can slash the validators if the fraction of adversarial validators exceeds $2/3$ [23]. Indeed, by borrowing a sufficient amount of stake, the adversary can temporarily control over $2/3$ of the validator set, gaining complete power over the PoS chain for some time. It can then cause a safety violation, and subsequently withdraw its stake to pay back its loan, thus violating safety without any financial cost. This attack highlights the circularity in the security argument for native staking: the chain where the native stake is locked is the same as the chain secured by this stake. The remote staking protocol overcomes the limitations of native staking by breaking the circularity above, *i.e.*, by separating the consumer chain secured by the remote stake, and the provider chain maintaining this stake.

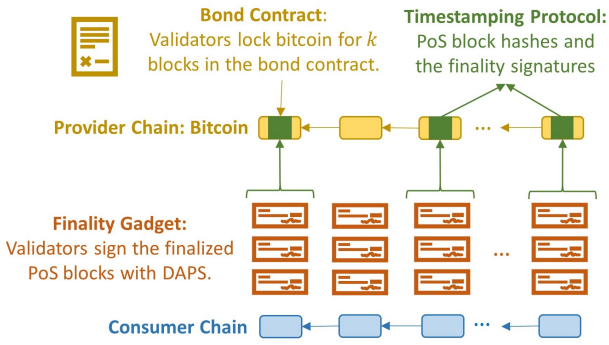

Figure 1: Remote staking protocol with dumb contracts . Validators lock their stake in a *bond contract*. They then become eligible to run the consensus protocol of the consumer chain. During this time, they sign the consumer blocks confirmed by the underlying consensus protocol with double-authentication-preventing signatures (DAPS) as part of the *finality gadget*. Hashes of the consumer blocks are periodically timestamped on Bitcoin along with the finality signatures on them as part of the *timestamping protocol*.

1.5 Protocol Overview

We highlight the main components of the remote staking protocol with dumb contracts: (i) the timestamping protocol, (ii) the finality gadget, and (iii) a bond contract on Bitcoin (Figure 1):

1.5.1 Timestamping protocol (Section 4 and 5.3). The timestamping protocol enables supporting an evolving validator set for both remote staking solutions. Similar to the design of [48], it writes the hashes of the PoS blocks onto the provider chain (*e.g.*, Bitcoin) along with the confirming signatures, to timestamp these blocks. Then, the adversarial validators cannot cause a safety violation by creating a conflicting chain after they unbond their stake (*cf.* long range, posterior corruption attacks [11, 20, 21]); since the timestamps would signal which of the conflicting chains was built earlier. The protocol also requires the timestamping of the provider chain (*e.g.*, Bitcoin) blocks within the consumer chain blocks; so that the validators and clients can track the changes in the stake distribution on the provider chain and verify the eligibility of the validator set for each consumer block height.

1.5.2 Finality gadget (Section 5.1). The finality gadget adds an extra layer of confirmation called *finalization* to the consumer chain's consensus protocol. It requires each validator to sign a *single* block, confirmed by the underlying consensus protocol, at each height with a double-authentication-preventing signature (DAPS) [41, 45]. These signatures, called *finality signatures*, enable the extraction

of the private key of the validator if it *equivocates*, *i.e.*, signs two distinct consumer blocks at the same height. A block is considered finalized if it gathers finality signatures from $2f + 1$ or more validators. Therefore, if two consumer blocks become finalized at the same height (safety violation), *i.e.*, gather $2f + 1$ finality signatures, the secret keys of at least $f + 1$ adversarial validators who have equivocated are exposed and thus, their stake can be burned.

When the consumer chain does not satisfy accountable safety, the finality gadget can be used to enforce accountability of the remote-staked validators regardless of whether the provider chain has a smart contract layer or not. With smart contracts on the provider chain, the finality signatures do not have to be DAPS, since slashing would be done by the smart contract and does not require the extraction of the private keys.

1.5.3 Bond contract (Section 5.2). Validators deposit their remote tokens in a bond contract to participate in the consensus protocol. In the case of a smart bond contract, the contract locks the stake until sufficient time passes after an unbonding request. In the case of Bitcoin as the provider chain, the contract ensures that before a timeout, a validator can send its bitcoin only to an unspendable address (*i.e.*, can only burn/slash its token). Once the timeout expires, the validator can retrieve (*i.e.*, unbond) its token by sending it to an address it controls. In the remote staking protocol, validators must use the same signing keys for the finality signatures as for the spending transactions sent to the bond contract. Therefore, after a safety violation and before the timeout, anyone can burn the stake of the adversarial validators whose secret keys have been exposed, without the risk of frontrunning. The bond contract can be instantiated with timelocks and *covenants*, a new primitive that enables restricting the spending address of Bitcoin contracts, or in lieu of covenants, with a *covenant committee* that emulates the functionality of covenants with an external committee of signers (*i.e.*, a multi-signature).

1.6 Implementation

We report on a production-ready implementation of a remote (Bitcoin) staking validator for a consumer chain running the Tendermint consensus protocol in Section 6. Our protocol does not require changing the original validator code of the consensus protocol besides sending finality signatures and monitoring Bitcoin. The additional overhead of our finality gadget and the bond contract for the consumer chain validators is a mere 179 MB of memory and the usage of 10% of the core on a Xeon E5 2698 v4 CPU. With minimal memory and CPU usage, our measurements demonstrate the practicality of our construction.

While our implementation uses a consumer chain running the Tendermint consensus protocol, the remote staking protocol can in fact be combined with any consumer chain. Moreover, the overhead due to our protocol on top of participating in the consumer chain's consensus remains the same, and our implementation remains unchanged when ported to other chains.

2 RELATED WORK

2.1 Accountability and Slashing

Accountable safety (also known as the forensic property [47]), *i.e.*, the ability to identify a fraction of the adversarial validators in the event of safety violations, is central to the design of PoS Ethereum [16, 17] and Tendermint [13, 14]. These protocols require their validators to be backed by their native tokens as collateral; so that these tokens can be *slashed*, *i.e.*, taken away, if the validator is found responsible for a safety violation. Token holders typically lock (*i.e.*, bond) their tokens and designate a particular validator to be supported by their bonded tokens in a process called *stake delegation* [24, 42].

Note that identifying the adversarial validators might not necessarily lead to their slashing. For instance, the adversarial validators can create a consensus fork by first *unbonding* their stake, and then, later in time, building a conflicting and finalized chain as if they were part of the validator set. This is called a posterior corruption attack, also known as the long range attack [11, 20, 21]. Although these validators will be identified as adversarial, they cannot be slashed after unbonding their stake. In this context, [48] proved that in the absence of external trust assumptions, there are attacks, where the adversarial validators cannot be identified before unbonding. Although thee clients can agree on the temporal order of the confirmed PoS blocks via a notion of *social consensus* to mitigate these attacks, as social consensus is a slow process, this would imply a long unbonding delay on the order of weeks.

To reduce the unbonding delay, [48] proposed using a separate provider chain as a secure timestamping server for checkpointing the confirmed PoS blocks. Through these timestamps, it provided *slashable safety*, *i.e.*, the ability to identify the adversarial validators in the event of safety violations, *before they unbond their stake*. However, slashable safety does not imply the act of slashing either. Indeed, in the case of a safety violation, more than 1/3 of the validators are already adversarial and can censor the evidence of protocol violation from being included on the chain and enforcing the slashing. In such cases, again, a complex social consensus process has to happen off-chain so that the violators can be slashed and kicked out of the validator set, and the remaining honest validators can restart the chain. In contrast, our remote staking protocol does not suffer from this issue as the remote stake resides on the provider chain, not on the PoS consumer chain, and it is automatically slashed if the safety of the consumer chain breaks down.

2.2 Finality Gadgets

Our finality gadget is an example of a broad class of protocols called *finality or accountability gadgets*, which are instantiated on top of existing consensus protocols to provide extra guarantees such as safety under network partitions and accountable safety. An early example of finality gadgets is Casper FFG used in Ethereum on top of a dynamically available consensus protocol (LMD GHOST) to checkpoint blocks, which constitute an accountably-safe prefix of the Ethereum ledger [16, 17]. Other examples include [36, 46]. Our finality gadget also provides accountable safety to the underlying protocol, albeit it is much simpler as it is instantiated with a fixed-sized validator set (without dynamic availability). A similar finality gadget was used in [33] with the purpose of enabling clients to opt for higher safety resilience at the expense of reduced

liveness resilience. In contrast to these works, [30] explored remote staking for consensus protocols without using a finality gadget. For accountable safety, it directly relies on a quorum intersection argument over the validators' signatures on the consumer blocks. However, without a view change mechanism, the construction gets stuck when there is an adversarial block proposer, thus suffering from liveness problems.

2.3 Accountable Assertions and DAPS

Accountable assertions were introduced to impose financial punishment by means of burning cryptocurrency in the event of equivocation such as double-spending [45]. They enable users to assert a single statement in a given context using their Bitcoin secret keys, which can then be verified with the corresponding public keys. If a user asserts two different statements in the same context, then its secret key can be obtained via a public and efficient algorithm using the two assertions, which leads to the loss of the user's funds on Bitcoin. Accountable assertions were used to design payment channels, where the payee is a distributed entity with asynchronous communication among its distributed components. In this case, if the payer commits a double-spend in its interaction with different components, it can eventually be punished, as accountable assertions do not require synchrony for leaking the secret keys of the equivocating parties.

DAPS, proposed earlier, is a special type of accountable assertion [41]. Potential use-cases include providing certificate authorities with cryptographic arguments to resist legal coercion and discouraging equivocations by such binding authorities. Both accountable assertions and DAPS are characterized by four algorithms: a key generation algorithm, an assertion or signing (for DAPS) algorithm, a verification algorithm and an extraction algorithm. Accountable assertions are required to satisfy completeness, secrecy for the secret key and extractability in the event that two distinct statements are asserted for the same context (called subject in [41]). DAPS are in addition required to have existential unforgeability, which implies secrecy. Unlike accountable assertions, DAPS do not require any non-extractable auxiliary secret information, *i.e.*, the whole secret signing key become extractable (*cf.* [45, Appendix A] for comparison). DAPS were later generalized to lattice-based predicate authentication preventing signatures (PAPS) that provide extractability with general predicates [12].

Our work uses DAPS (rather than accountable assertions) for finality signatures to ensure their existential unforgeability. Without existential unforgeability, there would be no guarantee that the finality signatures cannot be forged by the adversary on random blocks other than those confirmed by the consumer chain, an event that can lead to a liveness violation. As [45] rely on third parties to slash equivocating users' tokens, it cannot guarantee slashing if the adversarial users frontrun these third parties. In contrast, our remote protocol enforces the slashing of the equivocating users' tokens with the use of covenants (Section 2.4) or a covenant committee (Section 5.2).

2.4 Covenants

Covenants are powerful primitives to express Bitcoin contracts. In contrast to platforms like Ethereum, which are based on an accounts model, Bitcoin is based on the 'UTXO model', which is inherently stateless. When a transaction is executed, its input coins are destroyed, and new output coins are created. In a regular transaction, the owner of the input coin chooses which output coins are created. Covenants limit this freedom and restrict a coin such that the owner can send it only to a certain recipient or contract. This primitive can be combined with other contracting primitives, such as timelocks, to design stateful Bitcoin contracts. Covenants have been discussed in the Bitcoin community since at least 2013 [31] and in academic literature since 2016 [32]. In 2022, an attempt to activate the OP_CHECKTEMPLATEVERIFY proposal (CTV) [27] that would have enabled covenants failed to gain consensus [44] (*cf.* [10, 40] for details). Despite this, there are proposals [19, 26] currently in discussion that can be used to emulate covenants [39]. Covenants enable designing a bond contract on Bitcoin that achieves economic security without any trust assumption on third parties.

3 PRELIMINARIES

3.1 Model

3.1.1 Notation. Let $\kappa \in \mathbb{N}$ denote the security parameter. We say that an event happens with negligible probability if its probability is $o(1/\text{poly}(\kappa))$. It happens with overwhelming probability (w.o.p. for short) if it happens except with negligible probability.

3.1.2 Proof-of-stake consensus protocol. A proof-of-stake (PoS) consensus (state machine replication or total order broadcast) protocol involves two types of nodes: validators and clients. Validators receive transactions from the environment $\mathcal{Z}$ and communicate with each other via 'consensus messages' (*e.g.*, blocks, votes) to impose a total order on these transactions. Clients collect consensus messages from the validators, and upon gathering messages from sufficiently many validators, invoke a *confirmation rule* to output a sequence of *confirmed* transactions called the *ledger*. By outputting the same ledgers the clients can obtain the same end state after executing these transactions. The set of clients includes honest validators[2] and wallets that can come online and query for messages at arbitrary times.

The validator set of a PoS protocol can be *static* or *dynamic*. In the case of a *static* validator set, there is a public-key infrastructure (PKI) that assigns unique and publicly known identities to the validators, and the validator set does not change over time. In the case of a *dynamic* validator set, a node becomes eligible to participate in the SMR protocol upon bonding some minimum amount of stake in the protocol. A validator can also leave the validator set by unbonding its stake, in which case, it is no longer treated as a validator. Our goal is to analyze PoS consumer chains with a dynamic validator set. Although validators can bond different amounts in return for more power to influence the SMR protocol, in the subsequent sections, we will represent each validator as a unit-stake validator; since those with large stake can be represented as multiple unit-stake validators controlled by the same entity.

3.1.3 Blocks and chains. Transactions are often batched into *blocks* to ensure higher throughput. Then, the validators must ensure that the clients agree on a sequence of blocks, since agreement on a block

[2] An honest validator consists of (i) a validator algorithm exchanging the consensus messages of the SMR protocol, and (ii) a client algorithm outputing a ledger.

sequence together with the ordering of the transactions within the blocks determine a total order across the transactions. There is a *genesis* block B_0 that is common knowledge. Each block points to a *parent* block via a collision-resistant hash function. A block B is an ancestor of B', denoted by $B' \preceq B$, if $B' = B$, or B can be reached from B' via a path of parent pointers. Thus, each block B identifies a unique *chain*, denoted by C, that starts at the genesis block and ends at B. Similarly, each chain is identified by a unique block at the tip of the chain (when it is clear from the context, we will use the notation B to also denote the chain identified by the block B). Two blocks B and B' (and their chains) are said to *conflict* if neither $B \preceq B'$ nor $B' \preceq B$.

3.1.4 Adversary. The adversary $\mathcal{A}$ is a PPT algorithm that corrupts a subset of the validators, hereafter called *adversarial*. It gains access to the internal states of the corrupted validators and can cause them to violate the SMR protocol in an arbitrary and coordinated fashion (Byzantine faults). The remaining validators are called *honest* and execute the prescribed protocol. We denote the maximum number of adversarial validators by f and assume that the total number of validators is $n \geq 3f + 1$ at all times.

3.1.5 Networking. Time proceeds in discrete slots. Validators exchange messages with each other and the clients can receive messages from the validators through authenticated and reliable point-to-point channels [29]. The adversary controls the timing of message delivery and can peek into all messages before they are delivered. Upon coming online, clients receive all messages delivered to them while asleep. We say that a validator *broadcasts* a message if its intended recipients include all other validators and clients.

The network is *partially synchronous* [22]: the adversary has total control over message delays until an adversarially determined, finite global stabilization time (GST). After GST, the adversary has to deliver the messages sent by any honest validators to *all* intended recipients within a known Δ delay bound. Messages sent before GST are delivered by time GST + Δ. GST can be a causal function of the protocol randomness and is unknown to the clients and honest validators.

3.1.6 Security. Let C^{c}_t denote the confirmed chain output by a client c at time t.

Definition 1. *We say that an SMR protocol is secure with latency $T_{\mathrm{cf}} = \mathrm{poly}(\lambda)$ if:*
Safety: *For any time slots t, t' and clients c, c', either $C^{\mathsf{c}}_t \preceq C^{\mathsf{c}'}_{t'}$ or vice versa. For any client c, $C^{\mathsf{c}}_t \preceq C^{\mathsf{c}}_{t'}$ for all time slots t and $t' \geq t$.*
Liveness: *If $\mathcal{Z}$ inputs a transaction tx to an honest validator[3] at some slot t, then, $tx \in C^{\mathsf{c}}_{t'}$ for all $t' \geq \max(\mathrm{GST}, t) + T_{\mathrm{cf}}$ and clients c.*

A protocol is said to provide f_{s}-safety if it satisfies safety w.o.p. for all PPT $\mathcal{A}$ and when $f \leq f_{\mathrm{s}}$.

3.1.7 Accountable safety. In an accountably-safe protocol, when a safety violation happens, the clients can call a *forensic protocol* with the consensus messages they have observed so far, and obtain a *transferable* proof identifying f_a validators as protocol violators.

[3]In the case of dynamic stake, the transaction is input to an honest validator that is eligible to participate in the SMR protocol in its local view.

Definition 2 ([16, 34]). *A protocol provides accountable safety with resilience f_a, if (i) when there is a safety violation, at least f_a adversarial validators are identified by the forensic protocol as protocol violators, and (ii) no honest validator is identified w.o.p. Such a protocol is said to provide f_a-*accountable-safety.

By definition, when safety of a protocol providing f_a-accountable-safety is violated, the forensic protocol identifies at least f_a adversarial validators, which cannot happen if fewer than f_a validators are adversarial. As such, f_a-accountable-safety implies $f_a - 1$-safety.

3.2 Double-authentication-preventing Signatures

We next define the algorithms and properties that characterize the double-authentication-preventing signatures (DAPS) [12, 41].

Definition 3 (Double-authentication-preventing signatures (DAPS)). *Algorithms for DAPS:*

- $sk \xleftarrow{\$} \mathsf{DAPS\text{-}KeyGen}(1^\kappa)$: *The key generation algorithm outputs a secret signing key.*
- $pk \leftarrow \mathsf{DAPS\text{-}PK}(sk)$: *The public key generation algorithm takes a secret key and outputs a public verification key.*
- $\sigma \xleftarrow{\$} \mathsf{DAPS\text{-}Sign}(sk, m, \mathsf{ct})$: *The signing algorithm is a probabilistic algorithm that outputs a signature $\sigma \in \Sigma$ given a secret signing key sk, a message $m \in \mathcal{M}$ and a context $\mathsf{ct} \in C$.*
- $\{0, 1\} \leftarrow \mathsf{DAPS\text{-}Ver}(pk, m, \mathsf{ct}, \sigma)$: *The verification algorithm is a deterministic algorithm that outputs 1 if a given signature σ is verified against a public verification key pk, a message m and a context ct (0 otherwise).*
- $sk \leftarrow \mathsf{DAPS\text{-}Ext}(pk, m_1, \sigma_1, m_2, \sigma_2, \mathsf{ct})$: *The extraction algorithm is a probabilistic algorithm that outputs the secret signing key sk of a validator (w.o.p.) given two distinct message-signature pairs (m_1, σ_1) and (m_2, σ_2), where the signatures are valid under the same context ct.*

We separate the secret and public key generation to facilitate the EXT-SCMA *security property that is analogous to extractability [41, 45]. This is necessary for a DAPS scheme without a trusted setup, when there may be many different secret signing keys (that are hard to find) corresponding to a given public verification key. For the same reason, [41] assumes the existence of an efficient algorithm akin to our* DAPS-PK(.) *(without explicitly defining the algorithm), which verifies that a given secret key sk is the key corresponding to a public key pk. In turn, [45] assumes that for each pk, there is a unique sk.*

Security of a DAPS scheme is characterized by three properties: correctness, EXT-SCMA security (extractability, or formally, extractability under single chosen message attacks) and sEUF-CMA security (existential unforgeability, or formally, strong existential unforgeability under adaptive chosen message attacks). Intuitively, correctness guarantees that a correctly generated signature always passes verification. Existential unforgeability ensures that signatures are, w.o.p., unforgeable when the secret key is unknown, even after querying for multiple signatures. Finally, extractability guarantees that, w.o.p., two valid signatures on distinct messages under the same key and context can be used to extract the secret key. Formal definitions for these properties are stated in Appendix C.

3.3 The Provider Chain and Bitcoin

Let $\mathcal{B}_t^{\mathsf{c}}$ denote the confirmed Bitcoin chain in a client c's view at time t, *i.e.*, the k-deep block and its prefix within the longest Bitcoin chain held by c at time t. When working with other provider chains than Bitcoin, we will use $\mathcal{B}_t^{\mathsf{c}}$ to denote the confirmed provider chain in a client c's view at time t. We will denote the confirmed consumer chain by C_t^{c}). We hereafter denote the consumer blocks by capital B and the provider (*e.g.*, Bitcoin) blocks by small b.

In all future sections, we assume that the provider chain's consensus protocol, in particular, Bitcoin with confirmation depth k is safe and live with some finite latency (w.o.p.). The following proposition will be used in the description and analysis of the timestamping protocol (Section 5.3).

Proposition 1. *Suppose the provider chain's consensus protocol protocol is safe and live with latency T_{cf} (w.o.p.). Then for any two clients c_1 and c_2, and times t_1 and t_2, it holds that $\mathcal{B}_{t_1}^{\mathsf{c}_1}$ and $\mathcal{B}_{t_2}^{\mathsf{c}_2}$ are consistent, and for any client c and times t_1 and $t_2 \geq t_1$, $\mathcal{B}_{t_1}^{\mathsf{c}} \preceq \mathcal{B}_{t_2}^{\mathsf{c}}$. Moreover, there exists a parameter k_f, as a function of k, such that if a transaction is input to the provider chain when a client c_1's confirmed provider chain has height h, then for any client c_2, the transaction appears in the confirmed provider chain of c_2, before it reaches height $h + k_c$. If T_{cf} time passes as measured in wall clock time, there exists a constant k_f such that the confirmed provider chain grows by at most k_c blocks in the view of any client.*

When the provider chain is Bitcoin, Proposition 1 follows from the security analysis in [25], where k is the security parameter.

3.4 Tendermint

Tendermint is a PBFT-style [18] SMR protocol designed for the partially synchronous network (*cf.* Appendix A for details). It proceeds in *rounds*, each with a unique, known leader that proposes a block. Suppose there are $n = 3f+1$ active validators. Each honest validator tracks a step variable denoting the stage of the protocol execution within the current round. It can be one of Proposal, Prevote and Precommit. All messages are signed by the broadcasting validator.

At the beginning of the Proposal step, the leader sends a Proposal message, $\langle \text{Proposal}, h, r, v, vr \rangle$, (proposal for short) containing a block v of transactions. Here, h and r denote the leader's current height and round number respectively. Upon observing a proposal, each validator enters the Prevote step and sends a Prevote message $\langle \text{Prevote}, h, r, s \rangle$ (prevote) for either the proposed block ($s = id(v)$), or a special *nil* value ($s = \perp$), depending on the proposal and its internal state. Here, $id(v)$ represents a succinct, cryptographically secure hash of the block. If the validator observes $2f+1$ prevotes for a block v (or the *nil* value), it subsequently enters the Precommit step and sends a Precommit message $\langle \text{Precommit}, h, r, id(v) \rangle$ (precommit) for that block or the *nil* value. Finally, a validator or client *confirms* a block for height h upon observing $2f + 1$ precommits with height h for the block.

4 REMOTE STAKING PROTOCOL WITH SMART CONTRACTS

There are two main challenges for any remote staking protocol that aims to provide economic security for PoS consumer chains: (i) ensuring agreement on the validator set for each height of the consumer chain as stake shifts hand, and (ii) slashing the adversarial validators on the provider chain after a safety violation on the consumer chain. Our protocol relies on the timestamps of the provider blocks *within consumer blocks* to ensure agreement on the validator set; while using the timestamps of the consumer blocks *within provider blocks* to help identify the adversarial validators before they unbond. However, slashing the identified adversarial stake becomes a challenge if the provider chain is limited in its computational capabilities. For modularity, in this section, we consider a provider chain with Turing-complete smart contracts, which can slash any adversarial validator's stake once it is identified. In Section 5, we overcome the need to support smart contracts and extend our design to Bitcoin as the provider chain.

Our remote staking protocol assumes a consumer chain that runs a PBFT-style consensus protocol with accountable safety, where consumer blocks are confirmed by a quorum of validator signatures (*e.g.*, PBFT [18], HotStuff [50]). Our design can also be extended to consumer chains that lack accountable safety via the use of the *finality gadget* in Section 5. We next provide an intuitive description of the protocol. The complete description along with the algorithms can be found in Appendix B.

4.1 Bonding, Unbonding and the Validator Set

To bond its stake, a validator locks it in a bond contract on the provider chain via a *bonding transaction*. The stake remains locked in the contract until an unbonding request by the validator is received by the contract at some provider block b, and b becomes k_u blocks deep within the provider chain. Here, $k_u = O(k_f + k_c)$ is called the unbonding delay (*cf.* Section 3.3 for k_f and k_c). Some minimal unbonding delay is necessary to accommodate for delays in sending messages to the provider chain, such as evidence of protocol violation to slash the adversarial stake.

Proposer of a consumer block must include the hash of the highest confirmed provider block in its view within the consumer block. Then, the validator set for a consumer chain height h is determined by the highest provider block b whose hash is referred by the previously confirmed consumer blocks. This set consists of the validators who have bonded their stake at the provider chain blocks preceding b and have not sent an unbonding request received by the bond contract by block b.

4.2 The Timestamping Protocol

Validators send periodic timestamps of the consumer chain to the provider chain. These timestamps consist of the hash of the timestamped consumer block, a quorum of $2f + 1$ signatures on this hash that confirms the block (*e.g.*, precommits), and the block's height. When there is a posterior corruption attack, PoS blocks with earlier timestamps take precedence over those with latter timestamps. Note that the timestamps can be frequent (*e.g.*, every block), or at an interval of m blocks for some $m > 1$. To mitigate the attacks described below, the timestamping protocol imposes *stopping rules*:

4.2.1 Data availability attacks and stopping rule 1. If the clients observe a timestamp on the provider chain such that the consumer block B of the timestamp or a block in B's prefix is (partially or fully) unavailable or not confirmed, they stop outputting new consumer

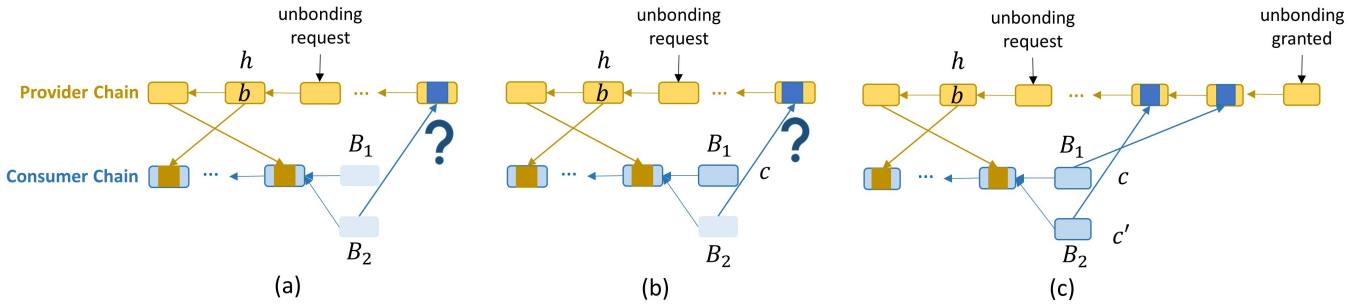

Figure 2: Illustration of the data availability attack and the safe-stop rule 1 (*cf.* Section 4.2.1). Yellow squares within the consumer blocks represent the hashes of the provider blocks included within the consumer blocks. Similarly, blue squares within the provider blocks represent the timestamps of the consumer blocks included within the provider blocks. Light blue blocks denote unavailable consumer blocks.

blocks to prevent non-slashable safety violations. This is called safe-stop rule 1. This so-called data availability attack was first discussed in [48] and is illustrated in Fig. 2. In the figure, the provider block at height h denotes the highest provider block referred by the earlier confirmed consumer blocks. Over 2/3 of the validators specified by h are adversarial and create two conflicting consumer blocks, B_1 and B_2. They subsequently send a timestamp to the provider chain for B_2, but initially keep both blocks private (Fig. 2-a). Then, the adversary reveals B_1 to a client c, but keeps B_2 hidden (Fig. 2-b). At this point, if c outputs B_1 as part of its ledger, it would cause a safety violation. This is because the adversary can reveal both blocks, after unbonding its stake, to a late-coming client c′, which would output B_2 instead of B_1, as consumer blocks with earlier timestamps take precedence over those with latter ones (Fig. 2-c). Moreover, the adversary will not be slashed as it has already unbonded its stake. Hence, to prevent non-slashable safety violations, upon observing a data unavailable timestamp, clients stop outputting new consumer blocks in their ledgers and send timestamps to the provider chain for the latest confirmed provider blocks in their views.

4.2.2 Escaping stake attacks and block output rules. We next describe a series of *escaping stake attacks* that exploit the fact that the stake is maintained on a different (provider) chain than the validated (consumer) chain. Again, suppose over 2/3 of the validators specified by b are adversarial. In the first attack, the adversarial validators send an unbonding request to the provider chain (Fig. 3-a), and once their request is granted, they create two conflicting confirmed blocks B_1 and B_2 (Fig. 3-b). They show the blocks B_1 and B_2 to the clients c_1 and c_2 respectively, yet, keep block B_2 hidden from c_1 and vice versa. At this point, if the clients choose to output their respective blocks, then they risk a non-slashable safety violation, as the adversarial validators have unbonded (*i.e.*, the stake has *escaped*). In the second attack on Fig. 3-c, a late-coming client c′ outputs B_2 upon observing its timestamp, thus conflicting with c_1 that has output B_1 before, after the adversarial stake has escaped.

To avoid these attacks, clients refuse to output blocks confirmed by old validator sets determined by old provider chain blocks. In these examples, they would reject blocks B_1 and B_2 as their validator sets were determined by a provider block, namely b, that has become too deep in the provider chain by the time the clients observe B_1 and B_2 (Fig. 3-b). Similarly, the clients would refuse to accept consumer blocks with timestamps far removed on the provider chain from the provider blocks that determined the validator set, *i.e.*, c' would reject block B_2 as its timestamp appears long after block b (Fig. 3-c). Indeed, if the majority of the validators were honest, a block and its timestamp would appear in the clients' views and on the provider chain respectively, long before the provider chain grows by more than k_u blocks, the unbonding delay.

4.2.3 Mismatched timestamp attacks and stopping rule 2. In a mismatched timestamp attack, the adversary exploits the fact that the bond contract cannot always detect the adversarial validators by purely inspecting the timestamps on the provider chain. Suppose over 2/3 of the validators specified by b are adversarial and create three conflicting consumer blocks, B_1, B_2 and B_3. Here, the adversary reveals B_1 to a client c, but keeps B_2 and B_3 private (Fig. 4-a). However, B_3 is timestamped before B_1 on the provider chain (Fig. 4-b). Upon seeing this timestamps with a block (namely B_3) that is either unavailable in its view, or conflicting with B_1, c urgently sends a timestamp of B_1 to the provider chain to notify the bond contract and the future clients about a potential safety violation. This timestamp for B_1 appears on the provider chain within a few blocks of the timestamp for B_3. At this point, depending on the design of the remote staking protocol, there are two possibilities:
1) Frequent timestamps: If we require every consumer chain block to be timestamped in order, the adversary must first send a timestamp of B_2 to the bond contract before B_3's timestamp can be accepted. Then, by sending a timestamp of B_1, c would have notified the bond contract about the adversarial validators that have confirmed conflicting blocks with their signatures. Therefore, these validators can be slashed by the contract before they unbond. However, this solution requires frequent timestamping, which might not be suitable for data-limited provider chains such as Bitcoin.
2) Rare timestamps: If we do not require every consumer chain block to be timestamped, then the adversary can directly timestamp B_3 without sending a timestamp of B_2 or any of the blocks in B_3's prefix, which are kept hidden from c. In this case, the bond contract cannot detect if B_3 is conflicting with B_1, and c cannot necessarily give any evidence that B_3's timestamp is for a block conflicting with B_1. Thus, the adversarial validators will be allowed to unbond.

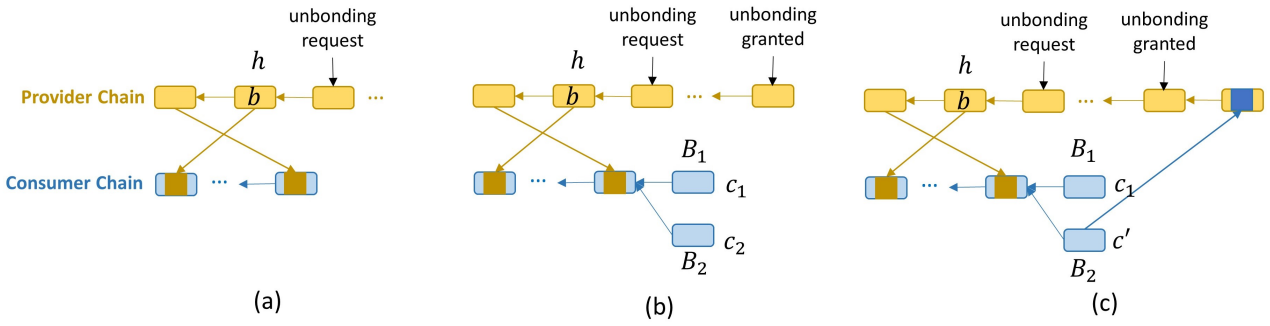

Figure 3: Illustration of the escaping stake attacks and the block output rules (*cf.* Section 4.2.2).

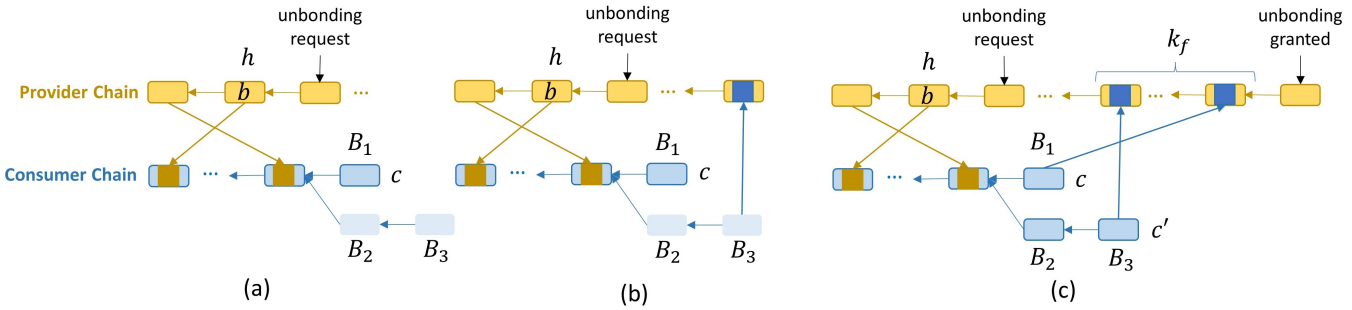

Figure 4: Illustration of the mismatched timestamp attack and the safe-stop rule 2 (*cf.* Section 4.2.3). Here k_f denotes the number of blocks that would be added to the provider chain during the time it takes for the timestamp of a consumer block to be included in the provider chain.

Finally, suppose a late-coming client c′ observes the system after the adversary unbonded its stake (Fig. 4-c). At this point, the adversary shows the previously unavailable blocks B_2 and B_3 to c′. If c′ decides to output B_2 and B_3 instead of B_1 as chains with earlier timestamps take precedence, it would cause a non-slashable safety violation by conflicting with c that has output B_1. Therefore, to prevent such safety violations, c′ does not output blocks whose timestamp conflicts with another timestamp appearing within vicinity (*e.g.*, within k_f blocks) of the original timestamp on the provider chain. This is called safe-stop rule 2.

4.2.4 Slashing. In certain cases, clients send timestamps to the provider chain in addition to the periodic timestamps to warn other clients about a potential safety violation as discussed above. If there were indeed a safety violation, these extra timestamps ensure the identification and slashing of the adversarial validators by the bond contract. Details about these conditions can be found in Appendix B.

4.3 Economic Security

Suppose there are $n \in \{3f+1, 3f+2, 3f+3\}$ validators. Then, equipped with the stopping rules, the remote staking protocol satisfies economic safety with resilience $f+1$ (against *all* attacks):

Definition 4 (Economic Safety). *A protocol provides economic safety with resilience f_a, if (i) when there is a safety violation, provider chain stake of at least f_a adversarial validators are slashed, and (ii) no honest validator is ever slashed (w.o.p.). Such a protocol is said to provide f_a-*economic-safety.

Theorem 3 (Economic Safety). *The remote staking protocol satisfies $f+1$-economic safety.*

Proof of Theorem 9 is given in Appendix G. It follows from a case-by-case analysis, where after a safety violation, $f+1$ adversarial validators' stake is slashed in all possible cases.

Theorem 4 (Liveness). *If the number of adversarial validators in any window of provider blocks is less than or equal to f, the remote staking protocol satisfies liveness with finite latency.*

Proof of Theorem 10 is given in Appendix H. It shows that when there are sufficiently few adversarial validators, the timestamps on the provider chain do not affect the consumer chain output by the clients. Liveness then follows from the consumer chain's liveness.

5 REMOTE STAKING PROTOCOL WITH DUMB CONTRACTS

We now build a remote staking protocol with Bitcoin as the provider chain and for concreteness, Tendermint as the consumer chain. In lieu of a smart bond contract that can slash adversarial stake, we design a novel slashing mechanism using DAPS, finality gadgets and covenant emulation. We first show how DAPS can be used to expose the secret signing keys of the adversarial validators that

violate the protocol rules to cause a safety violation (Section 5.1). We then describe how to use the extracted keys of these validators to financially punish them, thus achieve economic safety. Finally, we combine the slashing mechanism with the protocol of Section 4 to support changes in the stake distribution. We note that it is possible to extend the protocol to consumer chains other than Tendermint by using the finality gadget.

5.1 Tendermint with Finality Gadgets

We start with the observation that accountable safety of the consumer chain is a prerequisite for our remote staking protocol (Section 5.1.1), and Tendermint lacks accountable safety [14, Section III-C] (*cf.* Appendix D for more detail). We then demonstrate how Tendermint or any other consensus protocol can be made accountable with our *finality gadget*, and describe a forensic protocol that exposes the secret signing keys of the adversarial validators in the event of a safety violation (Section 5.1.2).

5.1.1 Accountable safety is necessary for key extraction. As a building block towards our remote staking protocol, we require the consumer chain to satisfy *DAPS safety*, which captures the ability of the protocol to expose the secret signing keys of the adversarial validators after a safety violation.

Definition 5 (DAPS safety). *A protocol provides DAPS safety with resilience f_a, if (i) when there is a safety violation, the secret signing keys of at least f_a adversarial validators are extracted by an efficient forensic protocol, and (ii) for any honest validator, given the set Q of (message, context, signature) tuples created by the validator, $\forall(\mathsf{ct}, m, m')$ such that $(m, \mathsf{ct}, .) \in Q \wedge (m', \mathsf{ct}, .) \in Q$, it holds that $m = m'$,* i.e., *the validator does not sign distinct messages with the same context. Such a protocol is said to provide f_a-*DAPS safety.

We observe that $(f+1)$-DAPS safety (Definition 5) implies $f+1$-accountable safety (Definition 2).

Theorem 5. *An SMR protocol that provides $(f+1)$-DAPS safety also provides $(f+1)$-accountable safety.*

Proof. Suppose the protocol provides $(f+1)$-DAPS safety. Then, in the event of a safety violation, the forensic protocol can extract the secret signing keys of at least $f+1$ adversarial validators. Moreover, by Defs. 5-(ii) and 7, no PPT adversary can extract the secret signing keys of any honest validator (w.o.p.). Hence, the set of signing keys exposed by the forensic protocol acts as a proof of protocol violation identifying at least $f+1$ adversarial validators, and no PPT adversary can identify an honest validator as a protocol violator by exposing its signing key (w.o.p.). Thus, the protocol provides $(f+1)$-accountable safety. □

5.1.2 Providing Tendermint with accountable safety and DAPS safety. To make Tendermint accountably-safe, we replace the original confirmation rule of Tendermint with a novel finalization rule based on finality signatures. Upon confirming a block B for height h within the Tendermint protocol, *i.e.*, outputting decision$[h] = B$ [15, Algorithm 1, line 49], each honest validator sends a height h finality signature $\sigma_{h,B}$ for block B, if it had not already sent a height h finality signature (Alg. 1). Each finality signature $\sigma_{h,B}$ by a validator val is a DAPS created with the secret signing key sk_{val} of the validator on the message $id(B)$ with context h, where $id(.)$ is a unique identifier for the block B (*e.g.*, a collision-resistant hash): $\sigma_{h,B} = \text{DAPS-Sign}(sk_{\mathsf{val}}, id(B), h)$ (it can be verified with the corresponding public verification key $pk_{\mathsf{val}} = \text{DAPS-PK}(sk_{\mathsf{val}})$). In other words, finality signatures are DAPS with a message space of hash *id* values and a context space of heights $h \in \{0, 1, \ldots\}$. Other signatures used by Tendermint need not be DAPS and can be of any type. For consistency with the Tendermint notation, we denote a height h finality signature for a block B by $\langle \mathsf{Final}, h, id(B)\rangle$. An honest validator sends a height h finality signatures only after sending finality signatures for the previous heights $1, \ldots, h-1$. A client finalizes a block B at height h upon observing a quorum of $2f+1$ *unique* height h finality signatures for block B, and after it has finalized blocks for all previous heights (unless the client has previously finalized a block conflicting with B, *cf.* Alg. 2).

Algorithm 1 A validator val's execution of the finality gadget. The function Broadcast broadcasts the provided signature and the messages. Here, message m and a signature on it by the validator is denoted by $\langle m\rangle_{\mathsf{val}}$. Each validator keeps track of the latest height for which a finality signature was broadcast.

```
1: height ← 0
2: upon decision[h]                          ▷ A block is confirmed at height h.
3:     if h = height + 1 then
4:         Broadcast ⟨Final, h, id(decision[h])⟩_val
5:         height ← height + 1
6:     end if
7: end upon
```

Algorithm 2 The finalization algorithm run by client c of Tendermint augmented with the finality gadget. The inputs $\mathcal{T}$ and sigs denote the new blocks and finality signatures downloaded by the client c from the network. The input C denotes the chain of blocks previously finalized by c ($C = B_0$, the genesis block, if no block has been finalized yet). The function GetBlocks($\mathcal{T}$) returns the sequence of blocks within $\mathcal{T}$ in increasing order of heights, where ties can be broken arbitrarily. Height of a block B is denoted by $|B|$. The algorithm returns a chain of finalized blocks. By construction, all finalized blocks are valid.

```
1: function OutputChain(𝒯, sigs, C)
2:     for B = B_1, ..., B_h ← GetBlocks(𝒯) do
3:         if |B| = |C|+1 ∧ C ⪯ B ∧ ∃(2f+1) ⟨Final, |B|, id(B)⟩ ∈ sigs
   then        ▷ If ∃ 2f + 1 signatures by the validator set for height |B|
4:             C ← C ‖ B
5:         end if
6:     end for
7:     return C
8: end function
```

The forensic protocol uses a single condition to identify the adversarial validators, and it is satisfied by at least $f+1$ validators in the event of a safety violation (and no honest validator under any circumstances). It identifies a validator as a protocol violator and returns its secret signing key upon receiving two finality signatures created by the validator for the same height, *i.e.*, context, but different blocks, *i.e.*, messages (Alg. 3).

Algorithm 3 The condition checked by the forensic protocol for key extraction.

```
1:  function KEY-EXTRACT(signatures)
2:      height ← 0
3:      upon ⟨Final, h, id(B′)⟩_val ∧ ⟨Final, h, id(B)⟩_val ∧ B ≠ B′
4:          Identify val as a protocol violator
5:          σ ← ⟨Final, h, id(B)⟩
6:          σ′ ← ⟨Final, h, id(B′)⟩
7:          sk_val ← DAPS-Ext(pk_val, id(B), σ, id(B′), σ′, h)
8:      end upon
9:      return sk_val
10: end function
```

We can finally prove the DAPS safety and liveness of the Tendermint protocol enhanced with the finality signatures.

THEOREM 6 (DAPS SAFETY). *Tendermint with the finality gadget satisfies $(f+1)$-DAPS safety.*

This theorem holds as long as the clients agree on the validator set for each height, which is ensured by the protocol in Section 4.

PROOF. Suppose the clients c_1 and c_2 finalized two conflicting chains. Then, there must be an earliest height h, at which they finalized two conflicting blocks, B_1 and B_2 respectively. Then, c_1 and c_2 must have respectively observed two quorums of $2f+1$ height h finality signatures $\langle \mathsf{Final}, h, id(B_1)\rangle$ and $\langle \mathsf{Final}, h, id(B_2)\rangle$ for B_1 and B_2. Upon obtaining the two quorums from the clients c_1 and c_2, the forensic protocol identifies the $f+1$ validators at the intersection of the two quorums as protocol violators since they have satisfied the condition in Alg. 3. By the extractability property of DAPS (Def. 8), the forensic protocol can extract their secret signing keys (w.o.p.). Morever, since honest validators send at most one finality signature per height, for any honest validator, given the set Q of message, height, signature tuples returned by the validator, $\forall (h, B, B')$ such that $(id(B), h, .) \in Q \wedge (id(B'), h, .) \in Q$, it holds that $id(B) = id(B')$. Thus, Tendermint with the finality gadget satisfies $(f+1)$-DAPS safety. □

COROLLARY 1. *Tendermint with the finality gadget satisfies $f+1$-accountable safety.*

Corollary 1 follows from Theorem 6.

Although finality signatures ensure accountable safety and key extraction for the adversarial validators identified as protocol violators, they do so by imposing a stronger so-called finality condition, *i.e.*, the existence of $2f+1$ finality signatures by the validators, as opposed to the original confirmation (decision) rule of Tendermint. We must thus ensure that the finality gadget retains the liveness of the Tendermint protocol under honest supermajority.

THEOREM 7 (LIVENESS). *If the number of adversarial validators is less than or equal to f, Tendermint with the finality gadget satisfies safety and after GST, liveness with finite latency (w.o.p.).*

PROOF. Let C_t^{val} denote the sequence of Tendermint blocks confirmed (decided) by an honest validator val following the original confirmation rule of Tendermint [15, Algorithm 1, line 49]. Note that the Tendermint protocol code executed by the honest validators is not affected by the finality gadget. Thus, when the number of adversarial validators is less than or equal to f, Tendermint satisfies agreement, validity, and after GST, termination by [15, Lemmas 3, 4, 7]. This implies that for all honest validators val and val′ and times t and t', (i) $C_t^{\mathsf{val}} \preceq C_{t'}^{\mathsf{val}'}$ or vice versa, (ii) if a block B appears in C_t^{val} at height h at some time t, then B appears within $C_{t'}^{\mathsf{val}'}$ at the same height by time $t' = \max(t, \mathrm{GST}) + \Delta$, (iii) these chains satisfy liveness per Definition 1. By property (i), for all times t and honest validators val, $C_t^{\mathsf{val}} \preceq C_t = \cup_{\text{honest val}'} C_t^{\mathsf{val}'}$, and for all times t and $t' > t$, $C_t \preceq C_{t'}$. Thus, if a block at height h conflicts with C_t at some time t, it eventually conflicts with all height h blocks at the honest validator val's chains C^{val}, and vice versa. Therefore, by Alg. 1, an honest validator sends a finality signature for each block in C_t by time $t' = \max(t, \mathrm{GST}) + \Delta$, and only for the blocks within C_t^{val} by time t'. Then, by the bound on the number of adversarial validators, each block $B \in C_t$ receives $2f+1$ finality signatures by round $\max(t, \mathrm{GST}) + \Delta$, which are observed by all clients at all times $t' \geq \max(t, \mathrm{GST}) + 2\Delta$, *i.e.*, all clients finalize the blocks in C_t by $\max(t, \mathrm{GST}) + 2\Delta$.

Finally, by (ii) and (iii), any transaction tx input to an honest validator at some time t appears in $C_{t'}$ for all rounds $t' \geq \max(t, \mathrm{GST}) + T_{\mathrm{cf}} + \Delta$. Hence, tx appears in all finalized chains $C_{t'}^{\mathsf{c}}$ for all clients c and times $t' \geq \max(t, \mathrm{GST}) + T_{\mathrm{cf}} + 2\Delta$, concluding the liveness argument. □

5.1.3 Performance. Each validator has to use a single DAPS per height while creating the finality signature at that height. This implies a linear communication complexity for the DAPS in the number of heights and validators, which is a small overhead on top of the complexity of Tendermint (*cf.* Section 6 for concrete numbers). Hierarchical deterministic wallets can be used to store a single DAPS key per validator.

5.1.4 Discussion. Our finality gadget can be composed with any SMR consensus protocol with a fixed-sized validator set to equip the protocol with accountable safety. Therefore, our remote staking protocol, whether it has smart or dumb contracts, can be instantiated with any such protocol as the consumer chain. Then, clients of the protocol could choose between outputting the full ledger, thus ensuring liveness in the absence of finality signatures, or its prefix that was attested by finality signatures, thus ensuring accountable safety (*cf.* [35, 36, 46] for the nested ledger paradigm).

In Appendix D, we formally prove that Tendermint [15] is not accountably-safe, prompting us to design our finality gadget. An alternative way to provide accountable safety to Tendermint would be to modify the protocol itself (*cf.* Appendix D.4). In this case, it is also possible to use DAPS directly for signing in-protocol messages to enable remote staking. We have opted to follow the finality signature approach for two reasons: (i) it is simple, and (ii) it adds DAPS *on top of Tendermint*, without changing the original protocol. The latter feature of the finality gadget helps its adoption by the existing blockchain projects, which appreciate modularity.

In terms of incentives, by delegating their stake to a validator, our architecture enables Bitcoin stake holders to earn staking rewards on the consumer chain (*i.e.*, *yield farm*), a form of investment that was not possible for bitcoins prior to our work.

5.2 Slashing Validators with the Bond Contract

In this section, we complete the description of the slashing mechanism by adding the bond contract to the finality gadget, and prove that the protocol achieves economic safety.

5.2.1 Bond contract. The bond contract requires the validators to put up bitcoin tokens as deposit, *i.e.*, bond their stake, in a *bond contract* deployed on Bitcoin. These deposits remain locked for a *predetermined* duration measured in the number of Bitcoin blocks, during which the validator must fulfill its duties towards the consumer chain.

5.2.2 Using covenants for slashing. The bond contract ensures that a validator's stake can only be sent to an unspendable output until its validator duties end, after which the validator can unbond by sending its stake to an address it controls. For this purpose, the contract uses a covenant along with a timelock to restrict the spending method until the validator's duties end. To slash a coin, it is sufficient to input a spending transaction (called the *slashing transaction*) to Bitcoin, upon which the contract sends the coin to the unspendable address (an OP_RETURN output) specified by the covenant (OP_CHECKTEMPLATEVERIFY). If the validator's secret key is exposed, anyone can use the exposed key to create and send a slashing transaction. Hence, a validator whose secret key is exposed cannot avoid slashing, even if it collaborates with some of the miners.

Algorithm 4 A simple bond contract implemented in Bitcoin Script using OP_CHECKTEMPLATEVERIFY. In a year, the validator can take their deposit back. Until then, if they leak their key, anyone can execute the slashing transaction.

```
OP_IF
    <1 year>
    OP_CHECKLOCKTIMEVERIFY OP_DROP
OP_ELSE
    <hash_of_slashing_transaction>
    OP_CHECKTEMPLATEVERIFY
OP_ENDIF

<validator_pubkey>
OP_CHECKSIG
```

5.2.3 Covenant emulation. Until covenants are enabled as part of Bitcoin script, we emulate their function with a *covenant committee* consisting of m members. We structure the bond contract as an $m+1$-out-of-$m+1$ multi-signature, such that $m+1$ signatures by the committee members and the staked validator are required to spend the deposit before the validator's duties end (Alg. 5). The committee co-signs a slashing transaction at the time of the creation of the bond contract, such that anyone can complete and execute it if the validator's secret key is exposed. The committee is trusted to never co-sign a different transaction collectively, as that would break the covenant. The committee members should ideally delete their signing keys after generating their signatures, to ensure that a future attacker cannot break the covenant, even if they compromise the committee. If at least one of the m members is honest and manages to keep its signing key private (existential honesty assumption), then the covenant becomes unbreakable (w.o.p.). The more committee members there are, the more plausible this existential honesty assumption becomes.

Anyone can join the covenant committee permissionlessly at the time of its formation. As it is used to ensure slashing when the validators violate the protocol rules, PoS chain users with high-value transactions (such as exchanges) are incentivized to join the committee to enforce its security. They do not have to trust anyone but themselves to delete their signing keys and guarantee that the covenant is unbreakable, thus removing any trust requirement (signing keys are deleted only after the multisig is created). As a further incentive, participation in the committee can be rewarded on Bitcoin using adaptor signatures or the consumer chain.

The committee can be represented in a space-efficient multi-signature scheme, such as MuSig2 [37]. The downside of a multi-signature is that if only a single member is offline or refuses to participate, then the committee cannot complete its signature. The chance of defection by a committee member increases as the committee size grows. In this case, the committee must exclude the members that halt the signing process. However, to sustain our 1-out-of-m assumption, an objective measure is required to distinguish between the case of a single malicious member halting the progress, and the case where $m-1$ malicious members try to exclude the only honest member from the committee. We can achieve the desired objectivity by requiring the committee members to publish their nonces, public keys and partial signatures on Bitcoin when the signature is not completed within some acceptable timeframe. This allows all users to observe which committee members published a correct signature on time, and which members refused to sign and thus must be excluded from the next signing attempt. This workaround ensures that a single member cannot disrupt the signing process for long, thus enabling the permissionless registration of the committee members.

With MuSig2 [37], the size of an emulated covenant on Bitcoin is $\sim$ 100 bytes, consisting of a 32-bytes aggregate public key and a 64-bytes signature. Optimistically, all committee members are honest, and the signature is promptly created off-chain. When parties must post their partial signatures to Bitcoin due to unresponsive members (worst-case), emulation would require 16 kBytes for $m = 100$, assuming 32-bytes keys, 64-bytes signatures and two 32-byte nonces for delinearization per member. Assuming that these partial signatures are posted as OP_RETURN transactions, which allow attaching 80 bytes of arbitrary data to the transaction output [5], posting this data costs less than 1000 USD as of April 15[4], an acceptable amount for securing large stake.

Further optimizations are possible to reduce the complexity of the aggregate signature generation. For instance, if many (*e.g.*, n) validators join the protocol together, each committee member can re-use the same key for emulating the covenant for all of the validators, reducing the worst-case on chain cost from $O(n \cdot m)$ to $O(m)$. Similarly, although MuSig2 for covenant emulation has two rounds – committing to the nonces (R) and signatures (s) – regular committee members can reduce this to a single round per aggregate

[4] Size of 1 OP_RETURN transaction is 205 bytes, it takes $\sim$ 30 satoshi per byte to have a transaction mined within three blocks with a latency of 30 minutes (on April 15) [2], and the average Bitcoin price on April 15 was 65, 753 USD [1].

signature (covenant emulation). Instead of committing to R and then s, each member can commit together with s also to its next nonce R_{next} for the next multisig, when the next validator joins and requires an emulated covenant. Then, all nonces are known to all parties before the next validator joins.

Algorithm 5 The bond contract, emulated with a deleted-key covenant. The committee pre-signs with a MuSig2 multi-signature.

```
OP_IF
    <1 year>
    OP_CHECKLOCKTIMEVERIFY OP_DROP
OP_ELSE
    <committee_pubkey>
    OP_CHECKSIGVERIFY
OP_ENDIF

<validator_pubkey>
OP_CHECKSIG
```

5.2.4 *From DAPS safety to economic safety with static stake.*

Theorem 8 (Economic Safety). *Equipped with covenants, the static-stake remote staking protocol with dumb contracts satisfies $(f+1)$-economic safety. In the absence of covenants, the static-stake remote staking protocol coupled with dumb contracts and using a covenant committee satisfies $(f+1)$-economic safety as long as one of the committee members is honest.*

Proof. By Theorem 6, Tendermint with the finality signature protocol satisfies $(f+1)$-DAPS safety. By Definition 5, when there is a safety violation, the secret signing keys of at least $f+1$ adversarial validators are extracted by an efficient forensic protocol. Then, when there is a safety violation, an honest client sends slashing transactions to the bond contracts of the identified $f+1$ validators, and these transactions are subsequently executed by Bitcoin. Therefore, in the event of a safety violation, $f+1$ adversarial validators get slashed. This holds for the covenant committee solution as well, as long as one of the committee members is honest. On the other hand, for any honest validator, given the set Q of message, context, signature tuples returned by the validator (*cf.* Alg. 8), $\forall(\mathsf{ct}, m, m')$ such that $(m, \mathsf{ct}, .) \in Q \wedge (m', \mathsf{ct}, .) \in Q$, it holds that $m = m'$. Thus, by existential unforgeability (Def. 7), no honest validator's Bitcoin stake can be slashed (w.o.p.). Therefore, the remote staking protocol instantiated with Tendermint satisfies $(f+1)$-economic safety. □

Recall that no PoS blockchain secured only by its native stake can slash the adversarial validators after a safety violation if their fraction exceeds 2/3 [23]. In this context, the remote staking protocol with a covenant committee improves on a standalone PoS blockchain by ensuring that at least 1/3 of the validators can be slashed after a safety violation as long as one of the committee members is honest. Indeed, if the committee members are the same entities as the validators, our solution would reduce the requirement of having over 1/3 honest validators for slashing to having a *single* honest validator.

5.3 Supporting Dynamic Stake

In Sections 5.1 and 5.2, we have shown how finality signatures guarantee the extraction of the secret signing keys of the adversarial validators, and how the adversarial validators' stake can be slashed on Bitcoin using the extracted keys in a static-stake protocol. Our complete design with dynamic stake combines this slashing mechanism with the protocol of Section 4, except that the smart contract on the provider chain is replaced with a covenant or a covenant committee, and the validators serve their duties for a predetermined number of Bitcoin blocks due to the use of timelocks on Bitcoin. The complete description along with the algorithms can be found in Appendix B. Its security is expressed as follows.

Theorem 9 (Economic Safety). *Equipped with covenants, the remote staking protocol with dumb contracts satisfies $(f+1)$-economic safety. In the absence of covenants, the remote staking protocol with dumb contracts and a covenant committee satisfies $(f+1)$-economic safety as long as one of the committee members is honest.*

Proof of Theorem 9 is given in Appendix G. It follows from a case-by-case analysis, where after a safety violation, $f+1$ adversarial validators' stake is slashed in all possible cases.

Theorem 10 (Liveness). *If the number of adversarial validators in any window of Bitcoin blocks is less than or equal to f, the remote staking protocol with dumb contracts satisfies liveness with finite latency.*

Proof of Theorem 10 is given in Appendix H. It shows that when the number of adversarial validators is sufficiently low, the timestamps on Bitcoin do not affect the Tendermint ledger output by the clients. Liveness then follows from the liveness of Tendermint.

6 IMPLEMENTATION

We implement a production-ready remote staking validator in 10,620 lines of Go that secures Tendermint using staked bitcoin. Our implementation covers the entire lifecycle of a validator, including submitting a Bitcoin transaction to lock up funds, maintaining DAPS key pairs, monitoring the Tendermint (consumer chain) consensus protocol, and creating finality signatures. Besides demonstrating the practicality of our construction, we use this implementation to evaluate the operational costs of running a validator, measured in its CPU and memory usage.

To simulate a production environment, we set up an end-to-end testbed with the following components: a private Bitcoin blockchain running `bitcoind`, a Tendermint blockchain implemented using the Cosmos SDK, a covenant committee, a monitoring program that slashes equivocating validators, and our validator implementation. The components reside on the same physical server, and are isolated in separate docker containers. We configure Tendermint to produce a block every 5 seconds. The validator implementation communicates with a Tendermint blockchain client, a separate process, and signs each block confirmed by it. In the steady state, the validator uses 179 MB of memory, and uses less than 10% of a core on a Xeon E5 2698 v4 CPU, which can be further optimized in a less portable implementation. This implies that the validator is lightweight, and fits in even the smallest cloud VM instances.

Clients of the consumer chain use the chain's original confirmation rule and the quorum of the DAPS signatures together to finalize

blocks. Thus, both the PoS validators and the clients run light clients of Bitcoin to verify that the DAPS signatures correspond to the Bitcoin addresses with stake. This adds little overhead for the clients (and validators) as Bitcoin is light-weight (24 MB/hour) compared to most PoS protocols (for Ethereum, 300 MB/hour)[5].

REFERENCES

[1] Bitcoin Price (I:BTCUSD) | YCHARTS. https://ycharts.com/indicators/bitcoin_price.
[2] Bitcoin Transaction Fee Estimator & Calculator. https://privacypros.io/tools/bitcoin-fee-estimator/.
[3] BitGo. https://www.bitgo.com/.
[4] Mesh security. https://github.com/osmosis-labs/mesh-security.
[5] OP_RETURN. https://en.bitcoin.it/wiki/OP_RETURN.
[6] Rootstock. https://rootstock.io/.
[7] Stacks: A bitcoin layer for smart contracts. https://stx.is/nakamoto.
[8] Private communication. 2022.
[9] Sunny Aggarwal. Mesh security talk at cosmoverse 2022. https://youtu.be/Z2ZBKo9-iRs?t=4937.
[10] Andreas Antanopolous. BIP119, EU regulatory attack, El Salvador, and much more in Q&A with aantonop. https://www.youtube.com/live/vAE5fOZ2Luw?feature=shared&t=575, 2022.
[11] Christian Badertscher, Peter Gazi, Aggelos Kiayias, Alexander Russell, and Vassilis Zikas. Ouroboros genesis: Composable proof-of-stake blockchains with dynamic availability. In *CCS*, pages 913–930. ACM, 2018.
[12] Dan Boneh, Sam Kim, and Valeria Nikolaenko. Lattice-based DAPS and generalizations: Self-enforcement in signature schemes. In *ACNS*, volume 10355 of *Lecture Notes in Computer Science*, pages 457–477. Springer, 2017.
[13] Ethan Buchman. Tendermint: Byzantine fault tolerance in the age of blockchains, 2016.
[14] Ethan Buchman, Rachid Guerraoui, Jovan Komatovic, Zarko Milosevic, Dragos-Adrian Seredinschi, and Josef Widder. Revisiting tendermint: Design tradeoffs, accountability, and practical use. In *DSN (Supplements)*, pages 11–14. IEEE, 2022.
[15] Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on BFT consensus. *CoRR*, abs/1807.04938, 2018.
[16] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *CoRR*, abs/1710.09437, 2017.
[17] Vitalik Buterin, Diego Hernandez, Thor Kamphefner, Khiem Pham, Zhi Qiao, Danny Ryan, Juhyeok Sin, Ying Wang, and Yan X. Zhang. Combining GHOST and casper. *CoRR*, abs/2003.03052, 2020.
[18] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *OSDI*, pages 173–186. USENIX Association, 1999.
[19] Anthony Towns Christian Decker. SIGHASH_ANYPREVOUT for taproot scripts. https://github.com/bitcoin/bips/blob/master/bip-0118.mediawiki, 2020.
[20] Phil Daian, Rafael Pass, and Elaine Shi. Snow white: Robustly reconfigurable consensus and applications to provably secure proof of stake. In *Financial Cryptography*, volume 11598 of *Lecture Notes in Computer Science*, pages 23–41. Springer, 2019.
[21] Evangelos Deirmentzoglou, Georgios Papakyriakopoulos, and Constantinos Patsakis. A survey on long-range attacks for proof of stake protocols. *IEEE Access*, 7:28712–28725, 2019.
[22] Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.
[23] Tim Roughgarden Eric Budish, Andrew Lewis-Pye. The Economic Limits of Permissionless Consensus, 2024. Keynote Speech by Tim Roughgarden at Financial Cryptography and Data Security 2024.
[24] Ethereum. PROOF-OF-STAKE (POS). https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/.
[25] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *EUROCRYPT (2)*, volume 9057 of *Lecture Notes in Computer Science*, pages 281–310. Springer, 2015.
[26] Ethan Heilman and Armin Sabouri. OP_CAT BIP Draft. https://github.com/EthanHeilman/op_cat_draft/blob/main/cat.mediawiki, 2023.
[27] James O'Beirne Jeremy Rubin. BIP119, CHECKTEMPLATEVERIFY. https://github.com/bitcoin/bips/blob/master/bip-0119.mediawiki, 2020.
[28] Interlay Labs. Interlay v2: Bitcoin finance, unbanked, 2023. https://gateway.pinata.cloud/ipfs/QmWp62gdLssFpAoG2JqK8sy3m3rTRUa8LyzoSY8ZFisYNB.
[29] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.
[30] Robin Linus. Stakechain: A bitcoin-backed proof-of-stake. In *Financial Cryptography Workshops*, volume 13412 of *Lecture Notes in Computer Science*, pages 3–14. Springer, 2022.
[31] Gregory Maxmell. CoinCovenants using SCIP signatures, an amusingly bad idea, 2023. https://bitcointalk.org/index.php?topic=278122.msg2970937#msg2970937.
[32] Malte Möser, Ittay Eyal, and Emin Gün Sirer. Bitcoin covenants. In *Financial Cryptography Workshops*, volume 9604 of *Lecture Notes in Computer Science*, pages 126–141. Springer, 2016.
[33] Joachim Neu, Srivatsan Sridhar, Lei Yang, and David Tse. Optimal flexible consensus and its application to ethereum. *CoRR*, abs/2308.05096, 2023. In IEEE S&P 2024.
[34] Joachim Neu, Ertem Nusret Tas, and David Tse. Snap-and-chat protocols: System aspects. *CoRR*, abs/2010.10447, 2020.
[35] Joachim Neu, Ertem Nusret Tas, and David Tse. Ebb-and-flow protocols: A resolution of the availability-finality dilemma. In *SP*, pages 446–465. IEEE, 2021.
[36] Joachim Neu, Ertem Nusret Tas, and David Tse. The availability-accountability dilemma and its resolution via accountability gadgets. In *Financial Cryptography*, volume 13411 of *Lecture Notes in Computer Science*, pages 541–559. Springer, 2022.
[37] Jonas Nick, Tim Ruffing, and Yannick Seurin. Musig2: Simple two-round schnorr multi-signatures. In *CRYPTO (1)*, volume 12825 of *Lecture Notes in Computer Science*, pages 189–221. Springer, 2021.
[38] Nomic. Nomic bitcoin bridge. https://www.nomic.io/.
[39] Andrew Poelstra. Cat and Schnorr Tricks I. https://medium.com/blockstream/cat-and-schnorr-tricks-i-faf1b59bd298, 2021.
[40] Andrew Poelstra. Cat and Schnorr Tricks II. https://medium.com/blockstream/cat-and-schnorr-tricks-ii-2f6ede3d7bb5, 2021.
[41] Bertram Poettering and Douglas Stebila. Double-authentication-preventing signatures. In *ESORICS (1)*, volume 8712 of *Lecture Notes in Computer Science*, pages 436–453. Springer, 2014.
[42] Polkadot. Polkadot consensus. https://wiki.polkadot.network/docs/learn-consensus.
[43] Rootstock. Powpeg: Building the most secure, permissionless and uncensorable bitcoin peg. https://dev.rootstock.io/rsk/architecture/powpeg/.
[44] Jeremy Rubin. Why CTV, why now? Was RE: Stumbling into a contentious soft fork activation attempt. https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2022-January/019736.html, 2022.
[45] Tim Ruffing, Aniket Kate, and Dominique Schröder. Liar, liar, coins on fire!: Penalizing equivocation by loss of bitcoins. In *CCS*, pages 219–230. ACM, 2015.
[46] Suryanarayana Sankagiri, Xuechao Wang, Sreeram Kannan, and Pramod Viswanath. Blockchain CAP theorem allows user-dependent adaptivity and finality. In *Financial Cryptography (2)*, volume 12675 of *Lecture Notes in Computer Science*, pages 84–103. Springer, 2021.
[47] Peiyao Sheng, Gerui Wang, Kartik Nayak, Sreeram Kannan, and Pramod Viswanath. BFT protocol forensics. In *CCS*, pages 1722–1743. ACM, 2021.
[48] Ertem Nusret Tas, David Tse, Fangyu Gai, Sreeram Kannan, Mohammad Ali Maddah-Ali, and Fisher Yu. Bitcoin-enhanced proof-of-stake security: Possibilities and impossibilities. In *SP*, pages 126–145. IEEE, 2023.
[49] EigenLayer Team. Eigenlayer: The restaking collective. https://docs.eigenlayer.xyz/overview/whitepaper.
[50] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan-Gueta, and Ittai Abraham. Hotstuff: BFT consensus with linearity and responsiveness. In *PODC*, pages 347–356. ACM, 2019.

[5] These numbers are for the network data requirements of the clients, not extra storage.

A TENDERMINT IN A NUTSHELL

Tendermint is a PBFT-style [18] SMR protocol designed for the partially synchronous network. It proceeds in *rounds*, each with a unique, known leader that proposes a block. There are $n = 3f + 1$ active validators. Each honest validator maintains five variables throughout the protocol execution: step, lockedValue, lockedRound, validValue and validRound. The variable step denotes the stage of the protocol execution within the current round. It can take the values Proposal, Prevote and Precommit.

At the beginning of the Proposal step, the leader sends a Proposal message, $\langle \text{Proposal}, h, r, v, vr \rangle$, (proposal for short) containing a block v of transactions. Here, h and r denote the leader's current height (*i.e.* consensus instance) and round number respectively, whereas vr denotes its validRound. Upon observing a proposal, each validator enters the Prevote step and sends a Prevote message, $\langle \text{Prevote}, h, r, s \rangle$, (prevote for short) for either the proposed block ($s = id(v)$), or a special *nil* value ($s = \perp$), depending on the proposal and its internal state. Here, $id(v)$ represents a succinct and binding commitment to the proposed block (*e.g.* its hash). If the validator

observes $2f + 1$ prevotes for a block v (or the *nil* value), it subsequently enters the Precommit step and sends a Precommit message, $\langle \text{Precommit}, h, r, id(v)\rangle$, (precommit for short) for that block (or the *nil* value). Finally, a validator or a client *finalizes* a block for height h upon observing $2f + 1$ precommits with height h for the block.

The lockedValue denotes the most recent block, *i.e.* the one with the largest round, for which the validator sent a precommit, and lockedRound denotes the round of this precommit. Similarly, validValue denotes the most recent block for which the validator has observed $2f + 1$ prevotes by distinct validators, and validRound denotes this round. If a validator has received a proposal $\langle \text{Proposal}, h, r, v, vr\rangle$ from the round leader before entering the Prevote step and its lockedRound $= -1$, *i.e.* it is not locked on any block, it sends a prevote for the proposed block. Otherwise, if its lockedRound > -1, *i.e.* it is locked on a block lockedValue, the validator checks if either v is the same as its lockedValue **(voting rule 1)** or if it has observed $2f+1$ round vr prevotes $\langle \text{Prevote}, h, vr, id(v)\rangle$ for v, such that $vr >$ lockedRound **(voting rule 2)**. If either of the voting rules is satisfied, it sends a prevote for the proposed block. Otherwise, it sends a prevote with the *nil* value.

B THE TIMESTAMPING PROTOCOL

Each client and validator downloads the consumer blocks and track the timestamps and the bond contract on the provider chain. When we talk about the remote staking protocol with smart contracts, we consider a consumer chain running a PBFT-style consensus protocol, where blocks are confirmed by a quorum of votes (*e.g.*, precommit votes in Tendermint, commit votes in HotStuff [50]). When we talk about the remote staking protocol with dumb contracts, we use Bitcoin as the provider chain and Tendermint with the finality gadget as the consumer chain. The protocol works in almost the same manner in both cases, and the differences will be explicitly highlighted in the description below. Before describing the details of the timestamping protocol, we recall the parameters k_c and k_f defined by Proposition 1. Intuitively, k_c denotes the number of provider blocks added to the confirmed provider chain during T_{cf} time, the liveness parameter of the provider chain. Similarly, k_f denotes the time, measured in the number of provider blocks, it takes for a message posted to the provider chain to appear in a confirmed block.

B.1 Determining the Validator Set

An honest validator includes the hash of the highest confirmed provider block in its view within the proposed consumer block. When an honest validator first receives a consumer block B proposed at a certain height, it checks if the provider block b referred by the hash in B's parent is confirmed in its view. If b becomes confirmed before the validator moves to the voting step of the protocol (Prevote in Tendermint), it continues to execute the consumer chain protocol as specified. Otherwise, it ignores block B.

When a client c first downloads a consumer block B, it checks if the block is *valid* in its view. The genesis consumer block, B_0, is assumed to be valid and specifies the initial validator set by referring to a provider block containing the bonding transactions for this initial set. Validity of any other consumer block B at height $|B|$ is determined by c according to the following rules: (i) there are $2f + 1$ confirming signatures for B (there are $2f + 1$ finality signatures for B with context $|B|$ in the case of dumb contracts) from the correct validator set for height $|B|$, (ii) the provider block b referred by B is confirmed in c's view, and (iii) B's parent is valid (Alg. 6). If so, c accepts B as a *valid* consumer block.

Algorithm 6 The algorithm used by a client c to determine if a consumer block B is available and valid. It takes the consumer chain C ending at B and the confirmed provider chain $\mathcal{B}$ in c's view as input and outputs true if B is available and valid. The function GetVals outputs the validator set determined for the next consumer chain height by an available and valid consumer chain C' and a confirmed provider chain $\mathcal{B}$ taken as input. It outputs $\perp$ if any provider block among those referred by the consumer blocks within C' is not in $\mathcal{B}$. The function Signed checks if there are $2f + 1$ finality signatures on a given consumer block by the specified validator set.

```
1:  function IsValid(C, B)
2:      if C = B_0 then   ▷ If C includes only the genesis consumer block
3:          return True
4:      end if
5:      if C[0] ≠ B_0 then
6:          return False
7:      end if
8:      B_0, ..., B_r ← C
9:      for i = 1 to r do
10:         vals ← GetVals(C[: i − 1], B)
11:         if ¬Signed(C[i], vals) ∨ vals = ⊥ then
12:             return False
13:         end if
14:     end for
15:     return True
16: end function
```

The validator set stays fixed during periods of m consecutive consumer chain heights, where m can be as little as 1. Suppose a client c wants to determine the validator set for period e after observing valid blocks for the periods $1, \ldots, e-1$. Let b be the highest confirmed provider block referred by the consumer blocks from periods $1, \ldots, e-1$ in c's view at some time t. Then, c determines the validator set for period e as those who have bonded their stake (on the provider chain) by the provider block b, and whose validator duties have not ended by b.

For ease of description, in the rest of this section, we assume that there are $n = 3f+1$ bonded validators, each with equal stake, at each period $e \in \mathbb{Z}^+$. We note that our protocol accommodates different numbers of validators at different periods with inhomogeneous stake amounts. In the latter case, the voting power of the validators must be scaled in proportion to the fraction of their stake within their period's total stake. Then, we can guarantee that if a safety violation is committed across blocks within a period e with a total stake of p, at least $p/3$ tokens belonging to the adversarial validators can be slashed.

B.2 Bonding and Unbonding

To join the validator set, a validator locks its stake in the provider chain's bond contract via a bonding transaction. Upon bonding its

Algorithm 7 The algorithm used by a client c to find the canonical consumer chain $\mathcal{C}_t^{\mathsf{c}}$ at time t. It takes as input a tree $\mathcal{T}$ of available and valid consumer blocks and the confirmed provider chain $\mathcal{B}$ in c's view at time t. The function GetCkpts outputs the ordered sequence of timestamps on the given provider chain $\mathcal{B}$. The function IsCor checks if a given timestamp is correct based on the height H of the canonical consumer chain output so far, the tracked current period per and the validator set identified by this canonical consumer chain. The function ProH returns the height of the provider block containing a given timestamp or referred by a given consumer block depending on its input. The function ConH returns the height of the specified consumer chain. The function GetCh returns the chain of available and valid consumer blocks behind a given timestamp using $\mathcal{T}$. It returns $\perp$ if there is an unavailable or invalid block in the consumer chain defined by the block at the preimage of the given timestamp. The function GetProH takes a consumer chain C and a provider chain $\mathcal{B}$ as input and returns the height of the highest provider block in $\mathcal{B}$ among those referred by the consumer blocks within C (if this highest provider block is not in $\mathcal{B}$, it returns $\perp$). The function IsLast returns true if a given consumer chain ends at a block that is the last block of its period. The function GetChildren returns the children of a given block.

```
1: function OutputConsumerCh(𝒯, ℬ)
2:     per ← 1
3:     ts_1, ..., ts_r ← GetCkpts(ℬ)
4:     C, H, vals ← B_0, 0, GetVals(B_0, ℬ)   ▷ H denotes the consumer height.
5:     h ← ProH(B_0)
6:     for i = 1 to r do                      ▷ Obtain the timestamped consumer chain
7:         if IsCor(ts_i, vals, H, per) ∧ ProH(ts_i) < h + k_d then
8:             C_i ← GetCh(𝒯, ts_i)
9:             if C_i = ⊥ then
10:                return C                   ▷ Safe-stop rule 1
11:            else if C ≤ C_i ∧ ConH(C_i) = H then
12:                if |ℬ| ≥ h+k_d+k_f ∧ ∃ts: (ProH(ts) < h+k_d+k_f ∧
                       ts conflicts with C_i) then
13:                    return C               ▷ Safe-stop rule 2
14:                else
15:                    C ← C_i                ▷ Update C.
16:                    H ← |C_i|
17:                    if IsLast(C_i) then
18:                        h ← GetProH(C_i, ℬ)
19:                        per ← per + 1
20:                        vals ← GetVals(C_i, ℬ)
21:                    end if
22:                end if
23:            end if
24:        end if
25:    end for
26:    chs ← GetChildren(𝒯, C[−1])
27:    chs ← {B: B ∈ chs ∧ |ℬ| < GetProH(B.C, ℬ) + k_d}
28:    while |chs| = 1 do
29:        C ← C ‖ chs                        ▷ Add the new child to C
30:        chs ← GetChildren(𝒯, chs)
31:        chs ← {B: B ∈ chs ∧ |ℬ| < GetProH(B.C, ℬ) + k_d}
32:    end while
33:    return C
34: end function
```

stake at some provider block b, it can act as a validator for the consumer chain heights described above while it continues its validator duties. In the case of a smart contract on the provider chain, the validator can request to unbond by sending a message to the bond contract. Its validator duties end at the confirmed provider block b' that includes the message. In the case of Bitcoin as the provider chain, its validator duties end at the provider block that extends b by some fixed amount K_a determined by the protocol. Afterwards, the validator can retrieve its stake at or after the provider block extending b' by k_u blocks (*i.e.*, extending b by $K_a + k_u$ blocks on Bitcoin), where $k_u = 2k_c + 4k_f$ and it is called the *unbonding delay*. To prevent early unbonding, the bond contract enforces a timelock on the bonded stake until the $(K_a + k_u)$-th block extending b (*cf.* Algs. 4 and 5).

B.3 Timestamping on the Provider Chain

Each honest validator periodically sends the *timestamp* of the last block of each period to the provider chain. To avoid duplicate timestamps, a single client or validator called the *watchtower* can be tasked with timestamping new blocks. The timestamp of a consumer block consists of the hash of the block, its height and the quorum of $2f + 1$ signatures on the block (finality signatures on the block with context equal to its height in the case of dumb contracts) by the corresponding validator set. Note that the period e of a consumer block can be found by dividing its height H with m (*i.e.*, $e = \lfloor H/m \rfloor + 1$).

Two timestamps are said to conflict if they both include (i) the same consumer block height H, (ii) different consumer block hashes, and (iii) $2f + 1$ signatures (finality signatures with height H as context in the case of dumb contracts) on their respective consumer block hashes.

B.4 Block Output Rules (Alg. 7)

When there is a posterior corruption attack, a client c might observe conflicting valid consumer blocks confirmed (finalized in the case of dumb contracts) by the same validator set. In this case, c wants to identify and output only the *canonical* consumer chain consisting of blocks signed earlier. Towards this goal, it first downloads the blocktree of all valid consumer blocks. Let ts_i, $i \in [r]$, denote the sequence of timestamps on the provider chain, listed from the genesis to the tip of the chain (denoted by $\mathcal{B}_t^{\mathsf{c}}$) in c's view at time t. Starting at the genesis consumer block, c constructs the canonical consumer chain (denoted by $\mathcal{C}_t^{\mathsf{c}}$) one block at a time, by sequentially processing these timestamps. For $i = 1, \ldots, r$, let C_i denote the chain ending at the consumer block (denoted by B_i), which is the preimage of the hash within ts_i, if B_i and its prefix are available and valid in c's view at time t. Suppose c has processed the timestamp sequence until some timestamp ts_j and constructed so far as its canonical consumer chain, the chain C of available and valid consumer blocks ending at some block B with consumer chain height H and period e. Define $\tilde{e} = e + 1$ if B is the last block of its period; and $\tilde{e} = e$ otherwise. Let $h_{\tilde{e}-1}$ denote the height of the highest confirmed provider block referred by the consumer blocks within the periods $1, \ldots, \tilde{e} - 1$. We call the next timestamp ts_{j+1} *correct*, if (i) the height H_{j+1} included in ts_{j+1} is larger than H and $\lfloor H_{j+1}/m \rfloor + 1 = \tilde{e}$, (ii) ts_{j+1} includes over $2f + 1$ signatures

on its consumer block hash (finality signatures on its consumer block hash with context equal to height H_{j+1} in the case of dumb contracts) by the validator set of period $\tilde{e}$, and (iii) ts_{j+1} appears at a provider chain height less than $h_{\tilde{e}-1} + k_d$, where $k_d = 2k_c + k_f$ is called the *timestamp delay* (Line 7, Alg. 7). The items (i) and (ii) above are checked for a timestamp by the function IsCor(.) in Alg. 7. Then;

(1) **Safe-stop Rule 1:** (Line 9, Alg. 7) If (i) the timestamp ts_{j+1} is correct, and (ii) a block in C_{j+1} is either unavailable or invalid in c's view, then c stops going through the sequence ts_i, $i \in [r]$, and outputs C as its canonical consumer chain[6].
(2) (Line 11, Alg. 7) If (i) the timestamp ts_{j+1} is correct, (ii) every block in C_{j+1} is available and valid in c's view, (iii) the chain C_{j+1} is of the height specified by ts_{j+1}, and (iv) $C \preceq C_{j+1}$ (*i.e.*, C_{j+1} is consistent with the consumer chain output so far);
   - **Safe-stop Rule 2:** (Line 12, Alg. 7, Fig. 2) If $|\mathcal{B}_t^{\mathsf{c}}| \geq h_{\tilde{e}-1} + k_d + k_f$ and there is a correct timestamp at a provider chain height less than $h_{\tilde{e}-1} + k_d + k_f$ conflicting with C_{j+1}, then c stops going through the sequence ts_i, $i \in [r]$, and outputs C as its canonical consumer chain.
   - **Update:** (Line 15, Alg. 7) If $|\mathcal{B}_t^{\mathsf{c}}| < h_{\tilde{e}-1} + k_d + k_f$, or if $|\mathcal{B}_t^{\mathsf{c}}| \geq h_{\tilde{e}-1} + k_d + k_f$ and there is no correct timestamp at provider chain heights less than $h_{\tilde{e}-1} + k_d + k_f$ conflicting with C_{j+1}, then c sets C_{j+1} as the new canonical chain ($C \leftarrow C_{j+1}$) and moves to ts_{j+2} as the next timestamp.
(3) **Ignore:** If none of the cases above are satisfied, c ignores ts_{j+1} and moves to ts_{j+2} as the next timestamp.

Unless one of the safe-stop rules is triggered, c processes all timestamps on its provider chain and identifies a timestamped canonical consumer chain ending at some block B_ℓ from period e_ℓ. Let $h_{\ell-1}$ denote the height of the highest confirmed provider block referred by the consumer blocks by the end of period $e_\ell - 1$. Then, starting at B_ℓ, c complements the timestamped canonical chain by outputting a chain of available and valid consumer blocks uniquely extending B_ℓ, as long as the height of c's confirmed provider chain is less than $h_{\ell-1} + k_d$ (Line 31, Alg. 7). This is for fast progress at the speed of the consumer chain even after the latest timestamps.

B.5 Enforcing Slashing on the Provider Chain

In certain cases, clients send timestamps to the provider chain in addition to the periodic timestamps for the last consumer block of every period, to warn other clients about a potential safety violation. If there were indeed a safety violation, these extra timestamps ensure the identification of the adversarial validators (extraction of the adversarial validators' secret keys in the case of dumb contracts) and thus the slashing of their stake. Let C denote the canonical consumer chain in c's view and e denote the period of the last block in C. Let h denote the height of the highest provider block in c's confirmed provider chain, among those referred by the consumer blocks (all of which are valid by definition) in C from periods $1, \ldots, e-1$. Then, if any of the following happens, c sends a timestamp to the provider chain for *all* of the consumer blocks within C that follow the last consumer block in C with a correct timestamp on the provider chain *before or at block* $h' - k_d - k_f$, where h' denotes the height of c's confirmed provider chain:

(1) Client c decides to go offline. (In this case, it must notify other clients about its view of the confirmed consumer blocks.)
(2) The safe-stop rule 1 is triggered for client c.
(3) The safe-stop rule 2 is triggered for client c.
(4) Client c does not observe a correct timestamp on its confirmed provider chain for the last consumer block B from period e before the confirmed provider chain reaches height $h + k_d$.
(5) Client c observes two correct timestamps for heights in period e on its confirmed provider chain before height $h + k_d$, and the blocks behind one of the timestamps is either unavailable, invalid, or conflicting with those of the earlier correct timestamps.

In the case of smart contracts, the extra timestamps sent due to the emergency conditions above could contain extra information pertaining to the consumer chain's consensus protocol; so that any client can create an evidence of protocol violation for the adversarial validators, and send this evidence to the bond contract to slash their stake. In the case of dumb contracts, upon obtaining two quorums of $2f+1$ finality signatures for the same height but two different block hashes, any client can extract the secret keys of $f+1$ validators using Alg. 3 and send this evidence to the respective bond contracts to slash their stake.

[6] Client c knows the correct validator set for all periods $e \leq \tilde{e}$. This is because; every block in its current canonical consumer chain C is available and valid in its view. In particular, if B is the last block of period e, c can infer the validator set of period $e+1$ from C.

C FORMAL DEFINITIONS FOR THE PROPERTIES OF DAPS

C.1 Correctness

Definition 6 (Correctness). $\forall m \in \mathcal{M}$:

$$\Pr\left[\mathsf{DAPS\text{-}Ver}(pk, m, \sigma, \mathsf{ct}) = 1 : \begin{array}{r} sk \leftarrow \mathsf{DAPS\text{-}KeyGen}(1^\kappa), \\ pk \leftarrow \mathsf{DAPS\text{-}PK}(1^\kappa), \\ \sigma \leftarrow \mathsf{DAPS\text{-}Sign}(sk, m, \mathsf{ct}) \end{array}\right] = 1$$

Intuitively, correctness guarantees that a correctly generated signature always passes verification.

C.2 Existential Unforgeability

Algorithm 8 Game for Strong Existential Unforgeability under Adaptive Chosen Message Attacks (sEUF-CMA).

function $\mathsf{sEUF\text{-}CMA}_{\mathcal{A}}(1^\kappa)$
  $sk \leftarrow \mathsf{DAPS\text{-}KeyGen}(1^\kappa)$;
  $pk \leftarrow \mathsf{DAPS\text{-}PK}(sk)$;
  $M \leftarrow \emptyset$
  $(m^*, \mathsf{ct}^*, \sigma^*) \leftarrow \mathcal{A}^{O(\cdot,\cdot)}(pk)$ ▷ $\mathcal{A}$ can call multiple times.
  return $(m^*, \mathsf{ct}^*, \sigma^*) \notin Q \wedge \mathsf{DAPS\text{-}Ver}(pk, m^*, \mathsf{ct}^*, \sigma^*) \wedge \forall(\mathsf{ct}, m, m').\ (m, \mathsf{ct}, .) \in Q \wedge (m', \mathsf{ct}, .) \in Q \implies m = m'$
end function
function $O(m, \mathsf{ct})$ ▷ Signing oracle
  $\sigma \xleftarrow{\$} \mathsf{DAPS\text{-}Sign}(sk, m, \mathsf{ct})$
  $Q \leftarrow Q \cup \{(m, \mathsf{ct}, \sigma)\}$
  return σ
end function

Definition 7 (Existential Unforgeability). $\forall$ *PPT* $\mathcal{A}$:

$$\Pr[\mathsf{sEUF\text{-}CMA}_{\mathcal{A}}(1^\kappa) = 1] < negl(\kappa)$$

Intuitively, existential unforgeability guarantees that signatures are, except with negligible probability, unforgeable when the secret key is unknown, even after querying for multiple signatures.

C.3 Extractability

Algorithm 9 Game for Extractability under Single Chosen Message Attacks (EXT-SCMA)

1: **function** EXT-SCMA$_{\mathcal{A}}(1^\kappa)$
2: $(pk, \text{ct}, m_1, \sigma_1, m_2, \sigma_2) \leftarrow \mathcal{A}$
3: **return** DAPS-Ver$(pk, m_1, \text{ct}, \sigma_1) \wedge$ DAPS-Ver$(pk, m_2, \text{ct}, \sigma_2) \wedge$ DAPS-PK(DAPS-Ext$(pk, m_1, \sigma_1, m_2, \sigma_2, \text{ct})) \neq pk \wedge m_1 \neq m_2$
4: **end function**

Definition 8 (EXT-SCMA security). *The* EXT-SCMA *game formalizes the extractability guarantee for the DAPS scheme.*

$$\forall \mathcal{A} \in \text{PPT}, \Pr[\text{EXT-SCMA}_{\mathcal{A}}(1^\kappa) = 1] < negl(\kappa)$$

Lastly, EXT-SCMA security guarantees that two valid signatures on distinct messages with the same key and context can be used to extract the secret key, except with negligible probability.

D LACK OF ACCOUNTABLE SAFETY IN TENDERMINT

We now explore why Tendermint lacks accountable safety.

D.1 Accountable Safety for Tendermint

If two clients finalize conflicting blocks B and B' at the same round, then they can identify the adversarial validators that sent precommits for both blocks by inspecting the $2f + 1$ precommits for B and B'. However, when the conflicting blocks are finalized at different rounds r and $r' > r$, they cannot use the quorum intersection argument directly on the two precommit quorums. To understand this, consider an honest validator that sent a precommit for B at round r. Even though the validator locked on B at round r and set its lockedValue $= B$ and lockedRound $= r$, it might have observed a quorum of $2f + 1$ prevotes for block B' at a later round $r^* > r$. In this case, upon observing the proposal $\langle \text{Proposal}, h, r', B', r^* \rangle$, the honest validator would send a prevote for block B' by **voting rule 2**, after which it could send a precommit. Then, the naive intersection argument between the precommit quorums would identify this honest validator as adversarial, which violates accountable safety.

To find the validators culpable for the safety violation in the example above, we consider the first round r^* such that a collection of $2f + 1$ prevotes from round r^*, *i.e.*, $\langle \text{Prevote}, h, r^*, id(B') \rangle$, is formed for block B'. The set of validators that sent these prevotes constitute the potential set of adversarial validators. Suppose these validators broadcast prevotes for some proposal $\langle \text{Proposal}, h, r^*, B', v, vr \rangle$. Now, since r^* is the first round greater than r, where a quorum of $2f + 1$ prevotes is formed for B', no validator could have observed a quorum for B' at any round $vr \in (r, r^*)$. Thus, the validators that were locked on B at round r should not have sent prevotes for B' as none of the voting rules could have been satisfied in their views. Sending a prevote in such circumstances is called the *amnesia* attack since the adversarial validators *forget* that they had an earlier lock on B (*cf.* [14]). Consequently, to determine the set of adversarial validators, clients must find the intersection of the validator sets that have sent the $2f + 1$ precommits for block B at round r and the $2f + 1$ prevotes for B' at round r^*.

D.2 Lack of Accountable Safety under Partial Synchrony

Unfortunately, the current version of Tendermint [15] does not allow clients to generate a proof of protocol violation in the case of an amnesia attack. This is due to the indistinguishability of two worlds with different sets of adversarial validators under partial synchrony.

Consider a client that aims to identify the culpable validators in the attack above (by calling the forensic protocol), after collecting transcripts and observing the quorum of $2f + 1$ round $r^* > r$ prevotes for B'. For this purpose, the protocol must ascertain that r^* is the earliest round, where a quorum of $2f + 1$ prevotes was formed for block B'. In world 1, this is indeed the case. Then, the protocol can identify the validators that sent both a round r^* prevote and a round r precommit for B as adversarial, since there is no set of $2f + 1$ prevotes for B' from any round $r'' < r^*$ that could have prompted these validators to release their locks on B. However, in world 2, there is a round $r'' < r^*$, in which the adversarial validators sent a quorum of $2f + 1$ round r'' prevotes for B' to the honest validators. No client (other than the honest validators) receives these prevotes for block B' due to partial synchrony. Thus, for the clients and the forensic protocol invoked by them, the two worlds are *indistinguishable*. Then, the adversary can convince the clients that an honest validator is a protocol violator by giving them the same proof output by the forensic protocol in world 1, which contradicts accountable safety.

Theorem 11. *Tendermint protocol does not provide accountable safety with resilience greater than one validator under a partially synchronous network.*

Formal proof of Theorem 11 is presented in the full version of this paper.

In Tendermint, proposals do not include the $2f + 1$ prevotes that justify the leader's validValue. The protocol instead expects the validators to receive these prevotes from the network, which happens in a timely manner under synchrony. However, Theorem 11 holds even if these prevotes are broadcast alongside the proposals (as in HotStuff); since its proof already assumes that the clients expect to see the round vr prevotes that justify a proposal $\langle \text{Proposal}, h, r = 2, B', vr \rangle$ before considering the proposal itself.

D.3 Lack of Accountable Safety under Synchrony

If the network is known to become synchronous when the forensic protocol is invoked, then the protocol can distinguish the two worlds above with different sets of honest validators by querying the honest validators and learning about the $2f + 1$ prevotes from round r'' in world 2. However, this is not sufficient to provide accountable safety, which requires the forensic protocol to generate a transferable proof of protocol violation. As it is not possible to create a *proof of absence*, each client must check for themselves which world they are in, *i.e.*, they must verify the absence or presence of

the $2f + 1$ prevotes from some round $r'' < r^*$ by communicating with the honest validators. This observation is formalized by the following theorem:

Theorem 12. *Tendermint protocol does not provide accountable safety with resilience greater than one validator, even if the network is known to become synchronous when the forensic protocol is invoked.*

Formal proof of Theorem 12 is presented in the full version of this paper.

D.4 Tendermint Made Accountably-safe

Inspired by the HotStuff-view protocol in [47], we can change Tendermint so that each Prevote message includes the validRound number vr within the proposal it supports. For instance, if a validator sends a prevote for the proposal $\langle \text{Proposal}, h, r^*, B', vr \rangle$, then it includes vr to its prevote as shown: $\langle \text{Prevote}, h, 2, id(B'), vr \rangle$. This small change suffices to make Tendermint accountably-safe and the proof of accountable safety proceeds similar to [47, Theorem 5.1].

E PROOF OF THEOREM 11

Proof. Towards contradiction, suppose Tendermint provides accountable safety with resilience of greater than one validator. Consider rounds $r = 0, 1, 2$ and 3 of some height h before GST. There are $3f + 1$ validators. Let P, Q and R denote disjoint sets of f validators each. Let x denote the remaining validator. We next consider the following two worlds.

World 1: Validators in R and x are adversarial and the rest are honest.

Round 0: At round 0, the adversary delivers only the messages among the validators in $P \cup R \cup x$. A new block B is proposed at round 0, and gathers $2f + 1$ prevotes and precommits from the validators in $P \cup R \cup x$. However, the honest validators in P do not observe the precommits by those in R. Thus, even though they lock on B, they do not decide B.

Round 1: At round 1, the adversary delivers only the messages among the validators in $Q \cup R \cup x$. A new block B' is proposed by an honest validator in Q and gathers f round 1 prevotes from the validators in Q. The adversarial validators in $R \cup x$ do not send round 1 prevotes for B'. Hence, the honest validators in Q send precommits with the *nil* value at round 1, and B' cannot be decided by the round 1 prevotes and precommits.

Round 2: At round 2, the adversary again delivers only the messages among the validators in $Q \cup R \cup x$. The adversarial leader x sends the proposal $\langle \text{PROPOSAL}, h, r = 2, B', vr = -1 \rangle$. The block B' gathers $2f + 1$ round 2 prevotes $\langle \text{PREVOTE}, h, r = 2, id(B') \rangle$ from the validators in $Q \cup R \cup x$; however, the adversarial validators in $R \cup x$ do not show their prevotes to the honest validators in Q. Hence, the honest validators in Q send precommits with the *nil* value at round 2, and B' cannot be decided by the round 2 prevotes and precommits.

Round 3: Finally, at round 3, the adversary delivers only the messages among the validators in $P \cup R \cup x$. An adversarial validator sends the proposal $\langle \text{PROPOSAL}, h, r = 3, B', vr = 2 \rangle$, and the adversary delivers the $2f + 1$ round 2 prevotes for B' to the honest validators in P. Hence, the validators in P unlock from B and, along with the adversarial validators in $R \cup x$, send prevotes and precommits for B'.

Clients in World 1: A client c_1 decides B at the end of round 0 upon observing the round 0 prevotes and precommits for B by the validators in $P \cup R \cup x$. A different client c_2 decides B' at the end of round 3 upon observing the messages sent by the validators in rounds 1, 2 and 3. Since Tendermint is accountably-safe with a resilience of greater than one validator, upon collecting the messages received by the clients, the forensic protocol outputs at least one validator from the set R (otherwise, it must have identified an honest validator which would imply a contradiction).

World 2: Validators in P and x are adversarial and the rest are honest.

Round 0: At round 0, the adversary delivers only the messages among the validators in $P \cup R \cup x$. A new block B is proposed at round 0, and gathers $2f + 1$ prevotes and precommits from the validators in $P \cup R \cup x$. However, the honest validators in R do not observe the precommits by those in R. Thus, even though they lock on B, they do not decide B.

Round 1: At round 1, the adversary delivers only the messages among the validators in $P \cup Q \cup x$. A new block B' is proposed by an honest validator in Q. The block B' gathers $2f + 1$ round 1 prevotes from the validators in $P \cup Q \cup x$; however, the adversarial validators in $P \cup x$ do not show their prevotes to the honest validators in Q. Hence, the honest validators in Q send precommits with the *nil* value at round 1, and B' cannot be decided by the round 1 prevotes and precommits.

Round 2: At round 2, the adversary delivers only the messages among the validators in $Q \cup R \cup x$. The adversarial leader x sends two proposals: $\langle \text{PROPOSAL}, h, r = 2, B', vr = -1 \rangle$ to the validators in Q and $\langle \text{PROPOSAL}, h, r = 2, B', vr = 1 \rangle$ to the validators in R. It also shows the $2f + 1$ round 1 prevotes for B' to the validators in R. Consequently, the block B' gathers $2f + 1$ round 2 prevotes $\langle \text{PREVOTE}, h, r = 2, id(B') \rangle$ from the validators in $Q \cup R \cup x$; however, the adversarial validator x does not show its prevote to the honest validators in $Q \cup R$. Hence, the honest validators in $Q \cup R$ send precommits with the *nil* value at round 2, and B' cannot be decided by the round 2 prevotes and precommits.

Round 3: Finally, at round 3, the adversary delivers only the messages among the validators in $P \cup R \cup x$. An adversarial validator sends the proposal $\langle \text{PROPOSAL}, h, r = 3, B', vr = 2 \rangle$, and the adversary delivers the $2f + 1$ round 2 prevotes for B' to the honest validators in R. Hence, all validators in $P \cup R \cup x$, send prevotes and precommits for B'.

Clients in World 2: A client c_1 decides B at the end of round 0 upon observing the round 0 prevotes and precommits for B by the validators in $P \cup R \cup x$. A different client c_2 decides B' at the end of round 3 upon observing all round 1, 2 and 3 messages, except the round 1 prevotes by the validators in $P \cup x$ and the round 2 proposal $\langle \text{PROPOSAL}, h, r = 2, B', vr = 1 \rangle$ by x. The adversarial validators in $P \cup x$ send the same messages to the forensic protocol as they do in world 1. Hence, the forensic protocol receives the same set of messages as in world 1 and identifies x and the same subset of the validators in R as in world 1 as protocol violators with overwhelming probability. Since the validators in R are honest in world 2, this is a contradiction with the definition of accountable safety. □

If the network were known to become synchronous when the forensic protocol is invoked, the forensic protocol would receive the $2f + 1$ round 1 prevotes by the validators in $P \cup Q \cup x$ from the honest validators in R (who observed these round 1 prevotes in round 2) and identify those in P as protocol violators in world 2.

F PROOF OF THEOREM 12

Proof. Towards contradiction, suppose Tendermint provides accountable safety with resilience greater than one validator. At the invocation of the forensic protocol, the network has become synchronous. We next construct the following two worlds inspired by the proof of Theorem 11:

World 1: This is the same as world 1 described by the proof of Theorem 11. The forensic protocol does not receive any round 1 messages from the validators in P and generates a proof that irrefutably identifies a validator in R as a protocol violator.

World 2: This is the same as world 2 described by the proof of Theorem 11, except that since the network has become synchronous, the forensic protocol has also received the round 1 prevotes by the validators in $P \cup x$ and the round 2 proposal $\langle \text{PROPOSAL}, h, r = 2, B', vr = 1 \rangle$. Thus, the set of messages received by the forensic protocol in world 2 is a superset of the messages received in world 2 of the proof of Theorem 11, which is the same as the messages received in world 1. This implies that given these messages, an adversarial client can generate the same proof as the one generated in world 1, which irrefutably identifies a validator in R as a protocol violator. However, since the validators in R are honest in world 2, this is a contradiction with the definition of accountable safety. □

G PROOF OF THEOREMS 3 AND 9

Proof of Theorems 3 and 9. Suppose there are two clients c_1 and c_2 such that the canonical consumer chains $C_{t_1}^{c_1}$ and $C_{t_2}^{c_2}$ are not consistent. Let B_1 denote the consumer block with the smallest height in $C_{t_1}^{c_1}$ among those conflicting with $C_{t_2}^{c_2}$. Similarly, let B_2 denote the block with the smallest height in $C_{t_2}^{c_2}$ among those conflicting with $C_{t_1}^{c_1}$. Let B_0 denote the common parent of B_1 and B_2.

Suppose c_1 first outputted B_1 at some time t_a, and c_2 first outputted B_2 at some time t_b as part of its canonical consumer chain. The validator set for the height of B_1 and B_2 is determined by the highest provider block $b \in \mathcal{B}_{t_a}^{c_1}, \mathcal{B}_{t_b}^{c_2}$, with height h, among those referred by the consumer blocks ending at the largest completed period at or before B_0. Next, we consider the following cases:

Case A: $|\mathcal{B}_{t_a}^{c_1}| \geq h + k_d$ and $|\mathcal{B}_{t_b}^{c_2}| \geq h + k_d$. Then, both c_1 and c_2 must have respectively output B_1 and B_2 at Line 15, Alg. 7 upon observing the correct timestamps $ts_1, ts_2 \in \mathcal{B}_{t_a}^{c_1}, \mathcal{B}_{t_b}^{c_2}$ at heights less than $h + k_d$. Without loss of generality, suppose ts_1 appears in the prefix of ts_2. In this case, if every block in the consumer chain determined by ts_1 is available and valid in c_2's view at time t_b, then c_2 would also output B_1 upon observing ts_1. Thus, there must be a block within the consumer chain determined by ts_1 that is unavailable or invalid in c_2's view at time t_b. However, in this case, the safe-stop rule 1 is triggered for c_2 upon observing ts_1, and it does not output B_2 (Line 9, Alg. 7). Hence, case A cannot happen.

Case B: $|\mathcal{B}_{t_a}^{c_1}| < h + k_d$ and $|\mathcal{B}_{t_b}^{c_2}| < h + k_d$. Then, one of the following cases must have happened:

- **Case 1:** Safe-stop rule 1 is triggered for c_1 before its provider chain reaches height $h + k_d$.
- **Case 2:** Client c_1 decides to go offline before its provider chain reaches height $h + k_d$.
- **Case 3:** Neither of the cases 1 and 2 happen until c_1's provider chain reaches height $h + k_d$. However, c_1 does not observe any correct timestamp for B_1 or its descendants by the time its provider chain reaches height $h + k_d$.
- **Case 4:** Neither of the cases 1 and 2 happen until c_1's provider chain reaches height $h + k_d$. Client c_1 observes a correct timestamp ts_1 for B_1 or its descendants on its provider chain at a height less than $h + k_d$, and every block timestamped by ts_1 is available and valid in c_1's view.

If cases 1, 2 or 3 happen, then c_1 sends timestamps to the provider chain for *all* of the blocks within its canonical consumer chain that follow the last consumer block with a correct timestamp on the provider chain at least before h, *i.e.*, at least all blocks following B_0.

- **Case I:** Safe-stop rule 1 is triggered for c_2 before its provider chain reaches height $h + k_d$.
- **Case II:** Client c_2 decides to go offline before its provider chain reaches height $h + k_d$.
- **Case III:** Neither of the cases I and II happen until c_2's provider chain reaches height $h + k_d$. However, c_2 does not observe any correct timestamp for B_2 or its descendants by the time its provider chain reaches height $h + k_d$.
- **Case IV:** Neither of the cases I and II happen until c_1's provider chain reaches height $h + k_d$. Client c_2 observes a correct timestamp ts_2 for B_2 or its descendants on its provider chain at a height less than $h + k_d$, and every block timestamped by ts_2 is available and valid in c_2's view.

If cases I, II or III happen, c_2 sends timestamps to the provider chain for *all* of the blocks within its canonical consumer chain that follow the last consumer block with a correct timestamp on the provider chain at least before h, *i.e.*, at least all blocks following B_0.

Then, we can deduce the following:

- **(1 and I), (1 and II), (1 and III), (2 and I), (2 and II), (2 and III), (3 and I), (3 and II), (3 and III):** In these cases, all online clients learn about the conflicting blocks B_1 and B_2, before the confirmed provider chain in its view reaches height $h + k_d + k_f$.
- **(4 and I), (4 and II), (4 and III):** In these cases, either c_1 learns about the conflicting blocks B_1 and B_2, or goes offline, before the confirmed provider chain reaches height $h + k_d + k_f$ in its view. In the latter case, c_1 sends timestamps to the provider chain for all of its consumer blocks following B_0, and an online client learns about the conflicting blocks B_1 and B_2 before the confirmed provider chain reaches height $h + k_d + 2k_f$ in its view.
- **(1 and IV), (2 and IV), (3 and IV):** In these cases, either c_2 learns about the conflicting blocks B_1 and B_2, or goes offline, before the confirmed provider chain reaches height $h+k_d+k_f$ in its view. In the latter case, c_2 sends timestamps to the provider chain for all of its consumer blocks following B_0, and an online client learns about the conflicting blocks B_1 and B_2 before the confirmed provider chain reaches height $h + k_d + 2k_f$ in its view.

- **(4 and IV):** In this case, both clients observe two correct timestamps with the same period on their confirmed provider chains before height $h + k_d$, and the timestamps attest to either conflicting, unavailable or invalid consumer blocks in their views. In this case, they both send timestamps to the provider chain for all of their consumer blocks following B_0. Then, at least one online client learns about the conflicting blocks B_1 and B_2 before the confirmed provider chain reaches height $h + k_d + k_f$ in its view.

Case C: $|\mathcal{B}^{c_1}_{t_a}| < h+k_d$ and $|\mathcal{B}^{c_2}_{t_b}| \geq h+k_d$. In this case, c_2 must have output B_2 at Line 15, Alg. 7 upon observing a timestamp $ts_2 \in \mathcal{B}^{c_2}_{t_b}$ at a height less than $h + k_d$. Then, depending on which of the four cases 1-2-3-4 is true for c_1, we investigate the following cases:

- **1-2-3 and $|\mathcal{B}^{c_2}_{t_b}| < h + k_d + k_f$:** Then, either c_2 learns about the conflicting blocks B_1 and B_2, or goes offline, before the confirmed provider chain reaches height $h + k_d + k_f$ in its view. In the latter case, an online client learns about the conflicting blocks B_1 and B_2 before the confirmed provider chain reaches height $h + k_d + 2k_f$ in its view.
- **1-2-3 and $|\mathcal{B}^{c_2}_{t_b}| \geq h + k_d + k_f$:** In this case, c_2 observes a timestamp on its confirmed provider chain before height $h + k_d + k_f$ that conflicts with B_2. Then, c_2 does not output B_2 upon observing the timestamp $ts_2 \in \mathcal{B}^{c_1}_{t_b}$ due to the safe-stop rule 2 (Line 12, Alg. 7). Hence, this case cannot happen (w.o.p.).
- **4 and $|\mathcal{B}^{c_2}_{t_b}| < h + k_d + k_f$:** In this case, c_1 observes two correct timestamps on its confirmed provider chain for the same period before height $h + k_d$, and the conflicting consumer chains attested by the two timestamps are available and valid in c_1's view by definition of case 4. Then, it sends timestamps to the provider chain for all of the blocks following B_0 on its canonical consumer chain. Thus, either c_2 learns about the conflicting blocks B_1 and B_2, or goes offline (sending timestamps for all of the blocks following B_0 on its canonical consumer chain), before the confirmed provider chain reaches height $h + k_d + k_f$ in its view. In the latter case, at least one online client learns about the conflicting blocks B_1 and B_2 before the confirmed provider chain reaches height $h + k_d + 2k_f$ in its view.
- **4 and $|\mathcal{B}^{c_2}_{t_b}| \geq h + k_d + k_f$:** In this case, c_1 observes two correct timestamps on its confirmed provider chain for the same period before height $h + k_d$, and the conflicting consumer chains attested by the two timestamps are available and valid in c_1's view. Then, it sends timestamps to the provider chain for all of the blocks following B_0 on its canonical consumer chain, which appear on the provider chain before it reaches height $h + k_d + k_f$. Thus, c_2 does not output B_2 upon observing the timestamp $ts_2 \in \mathcal{B}^{c_1}_{t_b}$ due to the safe-stop rule 2 (Line 12, Alg. 7). Hence, this case cannot happen.

Case D: $|\mathcal{B}^{c_1}_{t_a}| \geq h + k_d$ and $|\mathcal{B}^{c_2}_{t_b}| < h + k_d$. This is the same as case C, with the roles of c_1 and c_2 reversed.

Finally, we observe that in all possible cases, an online client c learns about the conflicting blocks B_1 and B_2 at the same height h', along with either (i) sufficient evidence to identify the adversarial validators that have confirmed the two blocks in the case of smart contracts, or (ii) two quorums of $2f + 1$ height h' finality signatures $\langle \mathsf{Final}, h', id(B_1)\rangle$ and $\langle \mathsf{Final}, h', id(B_2)\rangle$ for these blocks in the case of dumb contract, both before the confirmed provider chain reaches height $h+k_d+2k_f$. Upon obtaining the two quorums or the evidence from the online client, the forensic protocol identifies $f+1$ adversarial validators as protocol violators either by the accountable safety of the consumer chain, or by intersecting the two finality signature quorums as they have satisfied the condition in Alg. 3. In the latter case, by the extractability property (Def. 8), the forensic protocol can extract their secret signing keys (w.o.p.), before the confirmed provider chain reaches height $h + k_d + 2k_f$ in c's view. Then, in either case, c sends a slashing transaction to the bond contract, which is confirmed in the provider chain before it reaches height $h+k_d+3k_f \leq h+2k_c+4k_f < h+k_u$ in the view of any client. Since none of the $f+1$ validators identified by the forensic protocol could have spent their stake before the provider block at height $h+k_u$ due to the timelock, $f + 1$ adversarial validators get slashed. Morever, in the case of dumb contracts, since honest validators send at most one finality signature per height, for any honest validator, given the set Q of message, height, signature tuples returned by the validator, $\forall(h, B, B')$ such that $(id(B), h, .) \in Q \wedge (id(B'), h, .) \in Q$, it holds that $id(B) = id(B')$. Thus, by Defs. 2 and 7, no honest validator's stake can be slashed in either of the smart or dumb contract cases. Therefore, the remote staking protocol satisfies $(f + 1)$-economic safety. □

H PROOF OF THEOREMS 4 AND 10

For the liveness result below, we assume a synchronous network or a partially synchronous network, where GST is sufficiently bounded. An arbitrarily large GST would prevent liveness of the underlying consumer chain protocol for extended durations, during which the validators assigned to the pending period of m consumer blocks could unbond on the provider chain before the period is completed, which can only happen after GST. This issue can be avoided if the honest validators can designate when they would like to withdraw as in the smart contracts case and opt to remain as part of the validator set.

We also assume that the number m of heights at each consumer chain period is large enough; such that every period has at least one honest proposer w.o.p. If m is small, the proof remains mostly unchanged, except that the inductive step argument on the presence of an honest proposer in period m would have to refer to sufficiently many consecutive periods preceding m.

Proof of Theorems 4 and 10. Based on Proposition 1, we prove liveness by induction on the periods of consumer blocks extending the genesis block B_0. Let h denote the height of the provider block b_0 referred by B_0, and suppose the height of the longest confirmed provider chain held by the clients is h at the start of the protocol execution. Over $2f + 1$ validators within the initial validator set S_0 are honest.

Induction Hypothesis: Only a single valid consumer block can become confirmed (and gather $2f + 1$ finality signatures in the case of dumb contracts) at any height of period m. Safe-stop rules cannot be triggered for any client by the timestamps from periods $1, \ldots, m$. Correct timestamps of the available and valid consumer blocks from period m appear on the provider chain.

Base step: Only a single valid consumer block can be confirmed (and gather $2f + 1$ finality signatures) at any consumer chain height

of the first period. Moreover, all timestamps sent to the provider chain attest to available and valid blocks, since the number of adversarial validators within S_0 is at most f; and the safe-stop rule 1 rule cannot be triggered for any client. Therefore, all consumer blocks of period $m = 1$ become confirmed (and gather $2f + 1$ finality signatures) within $\Theta(T_{\text{cf}})$ time by the security of Tendermint ([15, Lemmas 3, 4, 7]), during which the confirmed provider chain advances less than k_c blocks in the view of any client. Thus, by the time all relevant honest validators have entered period 2, the highest provider block b_1 referred by the consumer blocks of the first period is at height at least h, and is at most k_c deep in the confirmed provider chain of any client. As $k_c < k_u$, no validator could have unbonded by this time. Furthermore, a timestamp of the blocks in the first period appears on the confirmed provider chains of all clients before height $h + k_c + k_f < h + k_d$, and all blocks attested by the timestamps of the first period are available, valid and consistent; implying that the clients keep outputting confirmed consumer blocks, and the safe-stop rule 2 cannot be triggered for any client.

Inductive step: Suppose that by the time all relevant honest validators have entered period m, the highest provider block b_{m-1} (at height h_{m-1}) referred by the blocks within the past periods $1, \ldots, m-1$ is at most k_c deep in the confirmed provider chain of any client. Also assume that the safe-stop rules cannot be triggered for any client by the timestamps with periods $1, \ldots, m-1$, and there is a single chain of available, valid and confirmed consumer blocks for the periods $1, \ldots, m-1$ in all clients' views. At least $2f+1$ of the validators assigned to period m are honest by assumption. Therefore, only a single valid consumer block can become confirmed (and gather $2f+1$ finality signatures) at any height of period m. Moreover, all timestamps sent to the provider chain attest to available and valid blocks, since the number of adversarial validators within the validator set for period m is at most f; and the safe-stop rule 1 cannot be triggered for any client by timestamps with period m.

All consumer blocks of period m become confirmed (and gather $2f + 1$ finality signatures) within $\Theta(T_{\text{cf}})$ time by the security of Tendermint ([15, Lemmas 3, 4, 7]), during which the confirmed provider chain advances less than k_c blocks in the view of any client. As $k_c < k_u$, no validator could have unbonded during this time. Since one of the block proposers at each period is honest (w.o.p.) and refers to the tip of its provider chain, by the time all relevant honest validators have entered period $m+1$, the highest provider block b_m referred by the consumer blocks of periods $1, \ldots, m$ is at most k_c deep in the confirmed provider chain of any client. Furthermore, a timestamp of the blocks in period m appears on the confirmed provider chain of all clients before height $h_{m-1} + 2k_f + k_c < h + k_d$, and all blocks attested by the timestamps of period m and earlier periods are available, valid and consistent; implying that the clients keep outputting confirmed consumer blocks, and the safe-stop rule 2 cannot be triggered for any client.

Finally, since there is an honest block among the m finalized blocks of any periods w.o.p., safe-stop rules cannot be triggered for any client, and correct timestamps of the available and valid blocks from each period appear periodically on the provider chain. Therefore, liveness is satisfied w.o.p. Note that in this *normal path*, validators do not send extra timestamps to the provider chain as cases 1-3-4 in the proof of Theorems 3 and 9 are never triggered. □