Accountable Light Client Systems for PoS Blockchains

Oana Ciobotaru, Web 3 Foundation
Fatemeh Shirazi, Independent
Alistair Stewart, Web 3 Foundation
Sergey Vasilyev, Web 3 Foundation

ABSTRACT

A major challenge for blockchain interoperability is having an on-chain light client protocol that is both efficient and secure. We present a protocol that provides short proofs about the state of a decentralised consensus protocol while being able to detect misbehaving parties. To do this naively, a verifier would need to maintain an updated list of all participants' public keys which makes the corresponding proofs long. Existing solutions either are not able to detect misbehaving parties (i.e. lack accountability) or are not efficient. We define and design a committee key scheme with short proofs that does not include any of the individual participants' public keys in plain which makes it very efficient. Our committee key scheme, in turn, uses a custom designed SNARK which has a fast prover time. Our committee key scheme can be used in an accountable light client system as the main cryptographic core for building bridges between proof-of-stake blockchains. By allowing a large number of participants, our scheme allows decentralization and interoperability without compromise. Finally, we implement a prototype of our custom SNARK for which we provide benchmarks.

1 INTRODUCTION

Blockchain systems rely on consensus among a number of participants, where the size of this number is important for decentralisation and the foundation of blockchain security. To know that a transaction is valid, one needs to follow the consensus of the blockchain. However, following consensus can become expensive in terms of bandwidth, storage and computation. Depending on the consensus type, these challenges can be aggravated when the size of participants' set becomes bigger or when the participants' set changes frequently. Light clients (such as SPV clients in Bitcoin [1] or inter-blockchain bridge components that support interoperability) are designed to allow resource constrained users to follow consensus of a blockchain with minimal cost. We are interested in blockchains that use Byzantine agreement type consensus protocols, particularly proof of stake systems like Polkadot [2], Ethereum [3] or many other systems [4–6]. These protocols may have a large number of consensus participants, from 1000s to 100000s, and in such PoS protocols, the set of participants often changes regularly.

Following the consensus protocols in the examples above entails proving that a large subset of a designated set of participants, which are called validators, signed the same message (e.g., a block header). Existing approaches have limiting shortcomings as follows: 1) verifying all signatures which has a large communication overhead for large validator sets; 2) verifying a single aggregatable signature, by computing an aggregate public key from the signer's public keys, has the shortcoming that any verifier still needs to know the entire list of public keys and this, again, has expensive communication if the list changes frequently; 3) verifying a threshold signature which has two shortcomings: first, such a signature does not reveal the set of signers impacting the security of PoS systems; second, it requires an interactive setup which becomes expensive if the validator set is large or changes frequently.

Our Approach: Committee Key Schemes. We introduce a committee key scheme which allows to succinctly prove that a subset of signers signed a message using a commitment to a list of all the signers' public keys. Our primitive is an extension of an aggregatable signature scheme and it allows us to prove the desired statement, in turn, by proving the correctness of an aggregate public key for the subset of signers. In more detail, the committee key scheme defines a committee key which is a commitment to all the signers' public keys. It generates a succinct proof that a particular subset of the list of public keys signed a message. The proof can be verified using the committee key. Because of the way the aggregatable signature scheme works, we need to specify the subset of signers; for this purpose we use a bitvector. More precisely, if the owner of the mth public key in the list of public keys signed the message, then the mth bit of this bitvector is 1 otherwise is 0. Using the committee key, the proof and the bitvector, a light client can verify that the corresponding subset of validators to whom the public keys belong (as per our use case) signed the message. Although the bitvector has length proportional to the number of validators, it is still orders of magnitude more succinct than giving all the public keys or signatures. Public keys or signatures are usually 100s of bits long and as a result this scheme reduces the amount of data required by a factor of 100 times or more. We could instantiate our committee key scheme using any universal SNARK scheme and suitable commitment scheme. However, to avoid long prover times for large validator sets, we use optimized custom SNARKs. We have implemented this scheme (Section 3) and it gives fast enough proving times for the use cases we consider: a prover with commodity hardware can generate these custom SNARK proofs in real time, i.e., as fast as the consensus generates instances of this problem.

Application: Accountable Light Client Systems. Light clients allow resource constrained devices such as browsers or phones to follow a decentralised consensus protocols. A blockchain is also resource constrained and hence could benefit from a light client system. In this case a light client verifier (e.g, smart contracts on Ethereum) allows building trustless bridges protocols between blockchains. Currently, computation and storage costs on existing blockchains are much higher than those in a browser on a modern phone. If such a bridge is responsible for securing assets with high total value, then the corresponding light

client system which defines such a light client verifier must be secure as well as efficient. Using the primitives and techniques described in this work, one can design a light client system with the following properties: accountability, asynchronous safety and incrementability reviewed below.

Accountability. Our light client system is accountable, i.e., if the light client verifier is misled and the transcript of its communication is given to the network then one can identify a large number (e.g., 1/3) of misbehaving consensus participants (e.g., validators in our case). Identifying misbehaving consensus participants is challenging in the light client system context when we want to send minimal data to the light client verifier. However, identifying misbehaviour is necessary for any proof of stake protocols including Polkadot and Ethereum whose security relies on identifying and punishing misbehaving consensus participants.

Asynchronous Safety. Our light client system has asynchronous safety i.e., under the consensus' honesty assumptions, our light client verifier cannot be misled even if it has a restricted view of the network, e.g., only connecting to one node, which may be malicious. This is because our light client system inherits the property of asynchronous safety from the Byzantine agreement protocol of the blockchain. Such light client systems would not be possible for consensus based on longest chains.

Succinctness. Our light client system is incremental - i.e its succinct state is incrementally updated - it is optimised to make these updates efficiently, which is particularly relevant for the bridge application, as opposed to trying to optimise verifying consensus decisions from the blockchain genesis.

1.1 Impact on decentralisation

For a blockchain network, having a large number of validators contributes greatly to better decentralization. This leads to better security both in terms of less point of physical failure and being able to distribute control over consensus which makes collusion harder. Some protocols have restricted their validator numbers to make light clients or bridges more efficient, e.g., by being able to run a DKG for threshold signatures (e.g., Dfinity [7]) or obtaining Byzantine agreement with all validators on every block (e.g., Cosmos [5]). More efficient light clients for blockchains with large validator sets offer both decentralisation and interoperability (bridging) without compromise.

1.2 Relevance to Bridge Security

In this section we review the impact of our scheme on bridge security. Blockchain bridges are protocols that allow value transfer between blockchains. Bridges have frequently been the target of attacks. We note that $1.2 billion has been stolen in attacks on insecure bridges during first 8 months of 2022 alone [8, 9]. Of the top 10 crypto thefts of all times, $1.6bn out of $3.4bn come from bridge attacks [9]. These confirm that bridges have frequently been a weaker point, compared to the security of the blockchains themselves and they carry a lot of economical value. An ideal bridge would be as secure as the least secure of the two blockchains. The most secure bridges use in-chain light client systems, e.g., Cosmos IBC protocol [10], to achieve this. Each bridged chain follow the other chain's consensus on-chain. To simplify, we will consider an on-chain light client of chain B on chain A, although B will also have the same for A. If B's consensus and the on-chain logic of A are secure, then adversary cannot convince the logic of A that B decided some event that B's clients do not agree as decided. This translates to the adversary for example not being able to create value on A without having locked any value on B.

A main reason why bridges might not use this approach is efficiency. Smart contracts and other on-chain logic is an extremely resource constrained environment compared to browsers or phones that light clients might target. One approach for efficiency is to design B's consensus so that the light clients are cheaper, for example by reducing the validator numbers. Cosmos chains currently have 33-175 validators [11],[12]. Many chains have many more, e.g., Ethereum's hundreds of thousands of validators, for more decentralisation and security. Alternatively, the light client can use threshold signature may be used however that means not having the same accountability guarantees and also limits validator numbers in practice, both discussed elsewhere.

Another approach to reducing on-chain complexity is optimism. Entities make a claim on chain A that something happened on chain B and this is accepted if no entity makes an on-chain challenge within a certain time, claiming that this is incorrect and triggering a more expensive procedure. A bridge that uses this approach in Optics for bridging Celo to other chains [13]. A less extreme example of this approach is NEAR's Rainbow bridge [14], where signatures are stored but not checked unless the correctness of a signature is challenged. The optimism approach relies on the censorship resistance of blockchain for security. In practice, blockchains may be censored for a period of time by an attacker with enough resources. An example of this was the result of the first round of Fomo3D on Ethereum [15], a smart contract that would pay a jackpot, a large amount (in the end 10,469 ETH), to the last user to pay the contract when no user does so for 30 seconds. The jackpot grew to such a large amount that it was worth a user buying up all the block space for 30 seconds [15]. For a claim and challenge protocol, the challenge is itself quite computationally expensive, so it may be sufficient to increase the cost of computation, the gas price on Ethereum, to make such a challenge unprofitable. Security against this attack requires a large reward for challenges or a long challenge period. For example the rainbow bridge has an 8 hour challenge period [16]. Long challenge periods would mean that bridge operations take a long time with consequences for usability.

Stakers in proof of stake protocol have an incentive for the chain (chain A) using that protocol to keep working, however they may not have stake in a chain (chain B) bridged to their chain. As a result, they may have no particular incentive in the correct functioning of a light client of chain A on chain B and so not to mislead the light client. In the case when the protocol of chain A has slashing, if an accountable light client on chain B is misled, one can prove to chain A, using information that is publicly available on chain B, of validator of chain A misbehaving in a way that will result in those validators being slashed on chain A. This gives the bridge similar economic security to chain A itself.

Protocols Cardano and Algorand

1.3 Applicability of Our Scheme

Our scheme is applicable to proof of stake blockchains where if something is decided by the chain, then a message is signed by some threshold fraction of a validator set, defined as a set of nodes or their public keys, which changes at well-defined times, those changes being signed by an appropriate threshold of the existing set. As mentioned such chains as Polkadot, the many Cosmos chains, or Ethereum fit this model. Our scheme is not applicable to chains using proof of work or many other proof of X schemes. Nor proof of stake protocols when only random validators or random subsets of validators decide something and the whole set never votes, such as protocols using the longest chain rule without a finality gadget.
Our scheme might well require a hard fork to be applied to many blockchains, especially those that have not implemented the required cryptography. It should be easily implementable for chains that use BLS signatures for consensus but those using signatures that do not support aggregation (e.g., the many using Ed25519), would need to use SNARKs with much slower prover time (e.g., zkBridge [17] for Ed25519)). To naively implement our scheme, we would also want validators to compute and sign the commitment to the next set. We note however that this is not strictly necessary, as the commitment could be computed on chain, maybe in a smart contract, as long as light client proof of the result of this computation can be constructed. This would result in longer proofs that cover validator set changes. For blockchains with expensive on-chain computation, native code support for the cryptography we use e.g., with precompiles for smart contracts might be required. It is planned to make the required changes to Polkadot and implement this scheme for it. We discuss in detail what would be required for a light client of Ethereum in Appendix Section M.

Structure. The paper is organised as follows. In Section 2, we sketch our proposed protocols and compare them to existing work. In Section 4, we give cryptographic preliminaries necessary for later sections. In Section 5 we describe our custom SNARKs and our committee key scheme. In Section 3 we give benchmarks for our custom SNARKs implementations. We conclude in Section 6. Our paper includes an extensive appendix for more details.

2 OUR SOLUTION

In this section we present a sketch of our solution for both the committee key scheme and the accountable light client system, then describe the technical challenges and contributions and finish with an overview of related work.

2.1 Sketch of Committee Key Scheme

Suppose that a prover wants to prove to a verifier that a subset S of some set T of signers with equal stakes have signed a message. One obvious approach would be using BLS aggregatable signatures with the following steps:

a. Verifier knows all public keys $\{pk_i\}_{i\in T}$ of signers.
b. Prover sends the verifier an aggregatable signature σ and a representation of the subset S.
c. Verifier computes the aggregate public key $apk = \sum_{i\in S} pk_i$ of the public keys of signers in S. Then it verifies the aggregatable signature σ for the aggregate public key apk and it accepts if the verification succeeds.

However, we can represent a subset S of a list of signers compactly using a bitvector b: the ith signer in the list is in S if and only if the ith bit of b is 1. Our committee key scheme describes an alternative approach:

a'. Verifier knows a commitment C to the list of public keys $(pk_i)_{i\in T}$.
b'. Prover sends the verifier an aggregatable signature σ, a bitvector b representing S, an aggregate public key apk and π, a succinct proof that $apk = \sum_i b_i pk_i$ i.e., that apk is the aggregate public key for the subset of signers in S given by the bitvector b; all of the public keys in S are a subset of the list of public keys committed to using C.
c'. The verifier using C, apk and the bitvector b checks if π is valid. It then verifies σ against apk and accepts if both steps succeed.

With the above committee key scheme, if C and π are constant size, the communication cost becomes $O(1) + |T|$ bits instead of $|T|$ public keys. So far we have implicitly assumed validators have equal stakes. One can generalise our alternative approach introduced above to validators with unequal stakes by including at 2.1, a'., a commitment to all stakes and to 2.1, b'., a claimed total signing stake that can be proved via a scalar product between stakes of the signing validators and the bitvector. Moreover, apk is appropriately replaced by the scalar product between signing validators' stakes and their respective public keys. The bitvector cannot be removed as it is needed for ensuring accountability of our light client system.

2.2 From CKS to Accountable Light Client

Below we sketch how a light client verifier uses our committee key scheme. Suppose that a light client verifier wants to know some information $info_n$ about the state of a blockchain at block number n without having to download the entire blockchain. Another entity, a full node, who knows all the data of the blockchain and is following the consensus, should be able to convince the light client verifier using a computational proof that $info_n$ was indeed decided.

We assume that $info_n$ can be proven from a commitment to the state at block number n that is signed by validators, here we assume that this commitment is a block hash H_n. To convince the light client verifier that H_n was decided, the full node needs to convince the light client verifier that a threshold number t of validators from the current validator set signed H_n, where t depends on the type of consensus. Byzantine fault tolerant based consensus often uses t to be over 2/3 of the total number of validators.

Keeping Track of the Validator Set: A light client verifier must be initialised with a committee key cpk_1 corresponding to the genesis validator set with keys $\mathbf{pk_1}$. At the end of each epoch, i.e., the time a validator set needs to be updated, the validators set of epoch i, with keys $\mathbf{pk_i}$ sign a message (i, cpk_{i+1}) where

cpk_{i+1} is a commitment to the validator set for the next epoch $\mathbf{pk_{i+1}}$. The light client verifier keeps track of cpk_i for each epoch. A light client proof must include a committee key scheme proof that a bitvector of validators, with a threshold number of 1s, with keys committed to in cpk_i signed (i, cpk_{i+1}). To convince a light client verifier knowing only cpk_1 of something in block n, all such proofs up to the epoch containing block n must be included. For an incremental light client system, such as one on a bridge, these validator set update proofs only need to be given once an epoch.

Proving the General Claim $info_n$: Once the light client verifier is convinced of cpk_{n-1} for the epoch $n-1$ and t of the validators in epoch $n-1$ signed H_n, it needs a committee key scheme proof for cpk_n and a bitfield with t ones that t validators signed H_n. Finally, such a proof needs the opening of the commitment H_n to $info_n$.

Accountability: Now suppose that a full node obtains a light client proof for something that contradicts something it sees as decided by the blockchain. For our bridge use case, all light client proofs will be publicly available on another blockchain. We assume that we can express this contradiction in terms of a pair of messages that should never be signed by an honest validator, and that any validators doing so can be punished. These we call incompatible messages. In this example, such pairs of messages should include validator sets commitment to different commitments (i, cpk_{i+1}) and (i, cpk'_{i+1}), $cpk_{i+1} \neq cpk'_{i+1}$ and similarly distinct H_n and H'_n. If a light client proof contains a message signed by a committee key which is a commitment to the public keys of a known set of validators which is incompatible with a message the same set signed on the blockchain, then the signature in the proof is a valid BLS signature with the claimed set of signers and so the full node should be able to report that public keys that signed both messages misbehaved. If an incompatible message was signed by a committee key which doesn't correspond to the claimed epoch's validator set, then at some point previously the light client proof must have shown that the committee key for some correct validator set signed the wrong committee key for the next set which is a message that is incompatible with the correct committee key that they signed on the real chain. Note that the accountability of our light client system instantiation relies on the accountability of the underlying consensus protocol. Indeed, our light client is accountable only if signatures on incompatible messages are enough for consensus accountability e.g., in Casper FFG [18] and it is not directly applicable to consensus protocols where forensics (such as in [19]) are required for accountability, e.g., Polkadot's GRANDPA, Section 4.1 [20]. If the consensus protocol is not accountable with signatures, then the consensus protocol needs to be modified by adding another layer (e.g. ABC[21], Polkadot's BEEFY [22]). .

Efficiency Gain: If one follows the obvious approach described above using BLS aggregation and aims to convince the light client verifier that $info_n$ is decided, then one needs to send $O(v)$ public keys for each validator set change, where v is the upper bound on the size of the validator set. Using our succinct committee key scheme however, one requires only a constant size proof and v bits for each validator set change to convince the light client that $info_n$ was decided. Since a public key or signature typically takes 100s of bits, our approach achieves much smaller proof sizes. More details our achieved efficiency are available in Section 3.

Formalisation: We give a formal model for the security properties of our accountable light client in Appendix I.3.

2.3 Our Custom SNARKs

Here we discuss how we use custom SNARKs with efficient prover time to implement our committee key scheme. While we achieved very fast proving time in our SNARKs implementation, this came at the cost of not using a general purpose SNARK protocol, in turn leading to a more involved security model and the necessity of additional security proofs.

The public inputs for our SNARKs are: an aggregate public key apk, a commitment C to the list of public keys $(pk_i)_{i\in T}$ and a bitvector $(b_i)_{i\in T}$ succinctly representing a subset S of public keys. Our SNARKs provers output a proof that $apk = \sum_{i\in T} b_i pk_i$ and that C is the commitment to the list of public keys $(pk_i)_{i\in T}$. However the list itself is a witness for the relations defining our SNARKs and so the verifiers do not need it and do not have to parse or check anything based on this possibly long list. We detail below two further optimisations of our custom SNARKs.

Commit and Prove SNARKs: Our SNARKs are an instance of commit and prove SNARKs (see Section 2.4.3). The underlying commitment scheme used for computing the public input commitment C is the same as the (polynomial) commitment scheme used in the rest of our SNARK(s). Hence, we do not need to add a witness for C to the SNARK constraint system in the same way we would have to if our commitment scheme were, e.g., to use a hash function. The constraints for checking a hash inside our custom SNARKs would increase the size of the constraint system so much that it would lead to several orders of magnitude increase in our prover time. The trade-off for our SNARKs design (i.e., with a commitment as part of the public input) is that we cannot use an existing SNARK compiler as a black box.

Constraint System Simplicity: Our constraint system is simple enough such that our custom SNARKs do not require a permutation argument or a matrix-vector product argument which general proving systems need to bind together gates. In fact, the underlying circuit for our SNARKs can be described as an affine addition gate with a couple of constraints added to avoid the incompleteness of our addition formulae. This simplification leads to smaller proof sizes and faster proving times.

2.4 Related Work

2.4.1 Naive Approaches and Their Use in Blockchains. There are a number of approaches commonly used in practice to verifying that a subset of a large set signed a message.

Verify All Signatures. One could verify a signature for each signing validator. This is what participants do in protocols like Polkadot [2], with 297 validators (or Kusama with 1000 validators) and Tendermint [5], which is frequently used with 100 validators). The Tendermint light client system, which is accountable and uses the verification of all individual signatures

approach, is used in bridges in the IBC protocol[10]. This approach becomes prohibitively expensive for a light client verifier when there are 1000s or millions of signatures.

Aggregatable Signatures. One could use an aggregatable signature scheme like BLS [23, 24] and reduce this to verifying one signature, but that requires calculating an aggregate public key. This aggregate key is different for every subset of signers and needs to be calculated from the public keys. This is what Ethereum does, which currently has 415,278 validators. However for a light client verifier, it is expensive to keep a list of 100,000s of public keys updated. As such only full nodes of Ethereum use this approach and instead light clients verifiers of Ethereum [25] follow signatures of randomly selected subsets of validators of size 512. This means that the resulting light client system is not accountable because these 512 validators are only backed by a small fraction of the total stake.

Threshold Signatures. Alternatively a threshold signature scheme may be used, with one public key for the entire set of validators. This approach was adopted by Dfinity [26]. Threshold signature schemes used in practice use secret sharing for the secret key corresponding to the single public key. This gives the schemes two downsides. Firstly, they require a communication-heavy distributed key generation protocol for the setup which is difficult to scale to large numbers of validators. Indeed, despite recent progress [26–28], it is still challenging to implement setup schemes for threshold signatures across a peer-to-peer network with a large number of participants, which is what many blockchain related use cases require. Moreover, such a setup may need repeating whenever the signer set changes. Secondly, for secret sharing based threshold signature schemes, the signature does not depend on the set of signers and so we cannot tell which subset of the validators signed a signature i.e., they are not accountable. Dfinity [26] uses a re-shareable BLS threshold signature, where the threshold public key remains the same even when the validator set changes. Such a signature scheme provides the light client verifier with a constant size proof, even over many validator set changes, but means that the proof not only does not identify which of a particular set of validators are misbehaving, but also we cannot say when this misbehaviour happened i.e., which validator set misbehaved. This is because the signature would be the same for any threshold subset of any validator set.

It is worth noting that if a protocol has already implemented aggregatable BLS signatures, our committee key scheme can be used without altering the consensus layer. Indeed it may be easier to alter a protocol that uses individual signatures to use aggregatable BLS signatures than to implement threshold signatures from scratch because the latter requires waiting for an interactive setup before making validator set changes.

2.4.2 Using SNARKs to Roll up Consensus. Celo [6] and Mina [29] blockchains have associated light client verifiers which allow their resource constrained users to efficiently and securely sync from the beginning of the blockchain to the latest block.

Plumo [30]. is the most relevant comparison to our scheme. It also tackles the problem we consider, i.e., that of proving validator set changes. In more detail, Plumo uses a Groth16 SNARK [31] to prove that enough validators signed a statement using BLS signatures from a set of the public keys. In Celo [6], the blockchain that designed and plans to use Plumo, validators may change every epoch which is about a day long and the Plumo's SNARK iteratively proves 120 epochs worth of validator set changes. Since in Celo there are no more than 100 validators in a validator set at any one time, the respective public keys are used in plain as public input for Plumo's SNARK, as opposed to a succinct polynomial commitment in the case of our custom SNARKs. All of the above increase the size of Plumo's prover circuit. Since Plumo is designed to help resource constrained light clients sync from scratch, it is not an impediment that the Plumo SNARK cannot be efficiently generated, i.e., in real time. In the case of a light client verifier for bridges (i.e., the most resource constrained application), we expect it to be in sync at all times and, by design, we care only about one validator set change at a time. Our slimmed down and custom SNARK not only can be generated in real time, but, also due to the use of specialised commitments schemes for public keys, our validator sets can scale up to much larger sizes as well without impacting the efficiency of our system.

Mina [29]. achieves light clients with $O(1)$ sized light client proofs using recursive SNARKs. This requires some nodes have a large computational overhead to produce proofs. Also because this requires verifying consensus with small circuits, they do not use the consensus paradigm discussed above where a majority of validators sign, and instead use a longest chain rule version of proof of stake [29]. Their protocol is not accountable because, as with Dfinity above, it is not possible to tell from the proof which validators signed off on a fork, nor when this happened. Another downside is that because the proof only shows the length of a chain (and its block density), similar to a Bitcoin SPV proof, a light client needs to be connected to an honest node to tell if a block is in the longest chain. If the client is connected to a single malicious node, it could be given a proof for a shorter fork and not see any proofs of chains the fork choice rule would prefer.

2.4.3 Commit-and-Prove and Related Approaches. Our custom SNARKs are an instance of the commit-and-prove paradigm [32–35] which is a generalisation for zero-knowledge proofs/arguments in which the prover proves statements about values that are committed.

In this context, ECLIPSE [36] presents a compiler that starts off with popular SNARKs (e.g., Sonic, PLONK, Marlin) and via a new general compilation method generates CP-variants for these SNARKs. Our proposed compiler uses as a first step the standard PLONK compiler. As a second step we simply re-cast the SNARK resulted in the first step as a SNARK for a new relation. The security of the re-casting holds under mild conditions that deterministically relate some polynomials processed by the verifier in the ranged polynomial protocol (before applying PLONK compiler) to some public inputs. To our knowledge, our re-casting conditions are less stringent than the conditions needed in [36].

We cannot use the ECLIPSE compilation technique either in full or in part to compile our custom SNARKs since the types of NP relations derived after ECLIPSE compilation are simply

incompatible with ours. While in the case of ECLIPSE, the witnesses for the NP relations before compilation remain witnesses also for the relations after compilation, in our case, some part of the public input before compilation becomes witness after the re-casting of the SNARK for a new NP relation. Thus, overall, ECLIPSE and the current work solve different problems. Finally, our compilation method requires only the PLONK compiler without additional computational steps so it is more efficient than the one in [36].

3 IMPLEMENTATION

We implemented and benchmarked the protocol. The implementation allows us to evaluate the performance of our protocol and serve as prototype for future deployment. The implementation is open source and publicly available at https://github.com/CCS23-anonymous/light-client. It is written in Rust and uses the Arkworks library.

Table 1 gives the prover and verifier time for the two SNARK schemes (basic accountable and packed accountable, see Section 5) with $v = n - 1 = 2^{10} - 1$, $v = n - 1 = 2^{16} - 1$ and $v = n - 1 = 2^{20} - 1$ signers. The benchmarks were run on commodity hardware, with an 2.2 GHz i7 and 16GB RAM. We remind the reader that by v we denote the maximum number of validators in our system and that n was defined in Section 4.6.1.

These signer numbers are approximately the range of the number of validators that we aimed our implementation at e.g. the Kusama blockchain (https://kusama.network/) has 1000 validators which is also the number that Polkadot is aiming for, and Ethereum 2 has about 348,000 validators and it has been suggested that there will be no more than 2^{19} [37].

At $v = n - 1 = 1023$, the prover can generate a proof in any scheme in well under a second, which is short enough to generate a proof for every block in most prominent blockchains. Even for $v = n - 1 = 2^{20} - 1$, the prover time is under 6.4 minutes, the time for an Ethereum 2 epoch, the time that validators finalise the chain. For verification time, the basic accountable scheme is slower, considerably so for larger signer numbers.

Table 2 gives the number of operations the prover and verifier use. Table 3 gives the proof constituents and also the total proof and input sizes in bits. The basic accountable scheme's verifier performance at large numbers is so slow because it includes $O(n)$ field operations, which dominate the running time, however at 1023 signers it gives the smallest size. The packed accountable scheme, which includes $O(n/\lambda)$ field operations, fairs better on the benchmarks for large signer sets. The prover is considerably slower for the latter scheme because it needs to do additional operations. At larger signer sizes, the proof size is dominated by the bitfield.

4 PRELIMINARIES

We assume all algorithms receive an implicit security parameter λ. An efficient algorithm is one that runs in uniform probabilistic polynomial time (PPT) in the length of its input and λ. We assume the correct parameters for the curves, groups, pairings, the group generators, etc. have been generated and shared with all parties before running any algorithm or protocol. A function $f(\lambda)$ is negligible in λ, written as $negl(\lambda)$, if $1/f(\lambda)$ grows faster than any polynomial in λ and is overwhelming in λ if $1 - f(\lambda) = negl(\lambda)$. poly($\lambda$) is some polynomial in λ and e.w.n.p. means except with probability $negl(\lambda)$. We write $y = A(x; r)$ when algorithm A on input x and randomness r, outputs y. We write $y \leftarrow A(x)$ for picking randomness r uniformly at random and setting $y = A(x; r)$. We denote by $|S|$ the cardinality of set S. $\mathbb{F}_{<d}[X]$ is the set of all polynomials of degree less than d over the field $\mathbb{F}$. For any integer $n \geq 1$, we denote by $[n]$ the set $\{1, \ldots, n\}$.

4.1 Pairings

If E is an elliptic curve defined over a prime field $\mathbb{F}_p$ of large characteristic p, we denote by $E(\mathbb{F}_p)$ the abelian group containing all the points $(x, y) \in (\mathbb{F}_p)^2$ on the curve along with the point at infinity. We will work with pairing friendly curves i.e., those with a secure [38, 39] efficiently computable, bilinear, non-degenerate mapping from a prime order subgroup of $E(\mathbb{F}_p)$ and a subgroup of the curve over the extension field. We will work with a *pairing-friendly two-chain*, i.e., a pair of pairing friendly elliptic curves $E_{inn} = E(\mathbb{F}_p)$ (*the inner curve*) and $E_{out} = E'(\mathbb{F}_r)$ (*the outer curve*), such that the pairing e_{inn} on E_{inn} works on subgroups of order r. $\mathbb{F}_p$ is the *base field* of $E_{inn} = E(\mathbb{F}_p)$ and $\mathbb{F}_r$ is its *scalar field*. We write $\mathbb{G}_{1,inn}$, $\mathbb{G}_{2,inn}$, $\mathbb{G}_{T,inn}$, $\mathbb{G}_{1,out}$, $\mathbb{G}_{2,out}$, $\mathbb{G}_{T,out}$ for cyclic subgroups of E_{inn}, $E(\mathbb{F}_{p^l})$,$\mathbb{F}_{p^k}$, E_{out}, $E'(\mathbb{F}_{r^{l'}})$, $\mathbb{F}_{r^{k'}}$ respectively for suitable l, k, l', k with the two pairings $e_{inn} : \mathbb{G}_{1,inn} \times \mathbb{G}_{2,inn} \rightarrow \mathbb{G}_{T,inn}$ and by $e_{out} : \mathbb{G}_{1,out} \times \mathbb{G}_{2,out} \rightarrow \mathbb{G}_{T,out}$. We write $g_{1,inn}$, $g_{2,inn}$, $g_{T,inn}$, $g_{1,out}$, $g_{2,out}$, $g_{T,out}$ respectively for randomly chosen generators of these groups. We use additive notation for group operations and write $[x]_{1,inn} = x \cdot g_{1,inn}$, $[x]_{2,inn} = x \cdot g_{2,inn}$. Concretely, our implementation uses BLS12-377 [40] and BW6-761 [41] for E_{inn} and E_{out}.

4.2 Secure Signature Aggregation

An aggregatable signature scheme compresses signatures using different signing keys into one signature. In this work we use an aggregatable signature scheme making explicit use of the proofs-of-possession (PoPs) [42]. For our concrete instantiation, we use aggregatable BLS signatures with an efficient aggregation procedure, i.e., by adding together keys and by multiplying together signatures, and protect against rogue key attacks [42] using PoPs. This is in contrast to other aggregation procedures that do not require PoPs for security but incur a higher computational cost (e.g., due to the use of multi-scalar multiplication [24]). For our concrete use case of accountable light clients systems, our efficient signature aggregation method results in a simple and more efficient SNARK which compensates for the cost of having to work with PoPs.

Definition 4.1. (Aggregatable Signature Scheme) An aggregatable signature scheme consists of the following tuple of algorithms (*AS.Setup*, *AS.GenKeypair*, *AS.VerifyPoP*, *AS.Sign*, *AS.AggKeys*, *AS.AggSig*, *AS.Verify*) such that for implicit security parameter λ:

Scheme	$v = 2^{10} - 1$		$v = 2^{16} - 1$		$v = 2^{20} - 1$	
	prover	verifier	prover	verifier	prover	verifier
Basic Accountable	564ms	26ms	22s	46ms	332s	231ms
Packed Accountable	830ms	29ms	31s	33ms	447s	57ms

Table 1: Proof and verifier times for the different schemes and numbers of signers

Scheme	Prover operations	Verifier operations
Basic Accountable	$12 \times FFT(N) + FFT(4N) + 9ME(N)$	$2P + 11E + O(n)F$
Packed Accountable	$18 \times FFT(N) + FFT(4N) + 12ME(N)$	$2P + 16E + O(n/\lambda + log(n))F$

Table 2: Expensive prover and verifier operations. $FFT(M)$ is an FFT of size M. $ME(M)$ is a multi-scalar multiplication of size M. P is a pairing, E is a single scalar multiplication and F is a field operation.

Scheme	Proof	Input	Actual proof + input size in bits		
			$v = 2^{10} - 1$	$v = 2^{16} - 1$	$v = 2^{20} - 1$
Basic Accountable	$5\mathbb{G}_{1,out} + 5\mathbb{F}$	$2\mathbb{G}_{1,out} + 1\mathbb{G}_{1,inn} + n$ bits	9088	73600	1056640
Packed Accountable	$8\mathbb{G}_{1,out} + 8\mathbb{F}$	$2\mathbb{G}_{1,out} + 1\mathbb{G}_{1,inn} + n$ bits	12544	77056	1060096

Table 3: Proof/input constituents and total proof/input size for implementation.

- $pp \leftarrow AS.Setup(aux_{AS})$: a setup algorithm that, given an auxiliary parameter aux_{AS}, outputs public protocol parameters pp.
- $((pk, \pi_{PoP}), sk) \leftarrow AS.GenKeypair(pp)$: a key pair generation algorithm that outputs a secret key sk, and the corresponding public key pk together with a proof of possession π_{PoP} for the secret key.
- $0/1 \leftarrow AS.VerifyPoP(pp, pk, \pi_{PoP})$: a public key verification algorithm that, given a public key pk and a proof of possession π_{PoP}, outputs 1 if π_{PoP} is valid for pk and 0 otherwise.
- $\sigma \leftarrow AS.Sign(pp, sk, m)$: a signing algorithm that, given a secret key sk and a message $m \in \{0,1\}^*$, returns a signature σ.
- $apk \leftarrow AS.AggKeys(pp, (pk_i)_{i=1}^u)$: a public key aggregation algorithm that, given a vector of public keys $(pk_i)_{i=1}^u$, returns an aggregate public key apk.
- $asig \leftarrow AS.AggSig(pp, (\sigma_i)_{i=1}^u)$: a signature aggregation algorithm that, given a vector of signatures $(\sigma_i)_{i=1}^u$, returns an aggregate signature $asig$.
- $0/1 \leftarrow AS.Verify(pp, apk, m, asig)$: a signature verification algorithm that, given an aggregate public key apk, a message $m \in \{0,1\}^*$, and an aggregate signature σ, returns 1 or 0 to indicate validity.

We say AS is an aggregatable signature scheme if it satisfies *perfect completeness* and *unforgeability* as standard security definitions (see appendix A for full details) and, additionally, *perfect completeness* for aggregation defined below.

Perfect Completeness for Aggregation An aggregatable signature scheme AS has perfect completeness for aggregation if, for every adversary $\mathcal{A}$

$$Pr[AS.Verify(pp, apk, m, asig) = 1 \mid pp \leftarrow AS.Setup(aux_{AS}),$$
$$((pk_i)_{i=1}^u, m, (\sigma_i)_{i=1}^u) \leftarrow \mathcal{A}(pp) \text{ such that}$$
$$\forall i \in [u], AS.Verify(pp, pk_i, m, \sigma_i) = 1,$$
$$apk \leftarrow AS.AggKeys(pp, (pk_i)_{i=1}^u),$$
$$asig \leftarrow AS.AggSigs(pp, (\sigma_i)_{i=1}^u)] = 1.$$

4.2.1 An Aggregatable Signature Instantiation. In the following, we instantiate the aggregatable signature definition given above with a scheme inspired by the BLS signature scheme [23] and its follow-up variants [24, 42].

Instantiation 4.2. *(Aggregatable Signatures) For aggregatable signatures, our implementation uses an instantiation of BLS signatures using proofs-of-possession which are $\mathbb{G}_{2,inn}$ elements, where the public keys are in $\mathbb{G}_{1,inn}$ and the signatures are in $\mathbb{G}_{2,inn}$. The public key aggregation is a simple sum of the public keys and the signature aggregation is a simple sum of the individual signatures. We instantiate E_{inn} with BLS12-377 [40]. Full details can be found in Appendix A.*

4.3 Committee Key Scheme

Bellow we introduce the notion of committee key scheme for aggregatable signatures (*CKS*). This notion, for an appropriate instantiation (Section 5.3), builds upon aggregatable signature schemes (Section 4.2) allowing a prover to convince a verifier that an alleged aggregated signature for subset of signers out of an all possible set of signers represented by a bitvector and a proof together with a key summarising an all possible set of signers' public keys (in the following called *committee key*) are valid. The notion of *CKS* and its instantiation can be used, in turn, for an accountable light client scheme instantiation (*LCI*) as sketched in Section 2.1. Before providing formal definition for *CKS*, we include some intuition for its chosen security properties.

- *Perfect completeness*: If all of *CKS.Verify* inputs except for an aggregated signature have been generated honestly and if the signature is accepted by *AS.Verify* (Definition 4.1), then *CKS.Verify* on the signature and honest inputs accepts; this is the counter-notion to perfect completeness for aggregation (Definition 4.1); it is used for proving *LCI perfect completeness*.
- *Soundness*: An adversary cannot output a *CKS* verifying proof and alleged aggregated signature pair without the aggregated signature being accepted by *AS.Verify*; this property is crucial for proving *LCI accountability completeness*.
- *Unforgeability*: It is similar to the underlying aggregatable signature scheme unforgeability and it is a direct consequence of the signature's scheme unforgeability and *CKS* soundness; it shows *CKS* has further security properties beyond those of an argument system. *CKS* unforgeability is used for proving *LCI accountable soundness*.

Definition 4.3. (Committee Key Scheme for Aggregatable Signatures) Let *AS* be an aggregatable signature scheme that fulfils Definition 4.1. A committee key scheme for aggregatable signatures consists of the following tuple of algorithms (*CKS.Setup*, *CKS.GenCommitteeKey*, *CKS.Prove*, *CKS.Verify*) such that for implicit security parameter λ:

- $(pp, rs_{vk}, rs_{pk}) \leftarrow CKS.Setup(\upsilon)$: a setup algorithm that, given an upper bound $\upsilon \in \mathbb{N}$, $\upsilon = \text{poly}(\lambda)$ outputs some public parameters pp and proving and verification keys rs_{pk} and rs_{vk}, respectively, where $pp \leftarrow AS.Setup(aux_{AS})$, for some aux_{AS} chosen by the aggregated signature *AS*.
- $ck \leftarrow CKS.GenCommitteeKey(rs_{pk}, (pk_i)_{i=1}^{u})$: a committee key generation algorithm that, given a proving key rs_{pk} and a list of public keys, outputs a committee key ck, where $u \leq \upsilon$.
- $\pi \leftarrow CKS.Prove(rs_{pk}, ck, (pk_i)_{i=1}^{u}, (bit_i)_{i=1}^{u})$: a proving algorithm that, given a proving key rs_{pk}, a committee key ck, a list of public keys and a bitvector $(bit_i)_{i=1}^{u} \in \{0,1\}^u$, outputs a proof π, where $u \leq \upsilon$.
- $0/1 \leftarrow CKS.Verify(pp, rs_{vk}, ck, m, asig, \pi, \textbf{bitvector})$: a verification algorithm that, given public parameters pp, a verification key rs_{vk}, a committee key ck, a message m, a signature $asig$, a proof π and a vector **bitvector** $\in \{0,1\}^*$, outputs 1 if the verification succeeds and 0 otherwise.

We say (*CKS.Setup*, *CKS.GenCommitteeKey*, *CKS.Prove*, *CKS.Verify*) is a committee key scheme for aggregatable signatures if it satisfies *perfect completeness* and *soundness* as defined below.

Perfect Completeness A committee key scheme for aggregatable signatures (*CKS.Setup*, *CKS.GenCommitteeKey*, *CKS.Prove*, *CKS.Verify*) has perfect completeness if for any message $m \in \{0,1\}^*$, for any vector of public keys $(pk_i)_{i=1}^{u}$ generated using $AS.GenKeypair(pp)$, for any bitmask $(bit_i)_{i=1}^{u} \in \{0,1\}^u$, for any aggregated signature $asig$, it holds that:

$$
\begin{aligned}
&Pr[AS.Verify(pp, apk, m, asig) = 1 \implies \\
&CKS.Verify(pp, rs_{vk}, ck, m, asig, \pi, (bit_i)_{i=1}^{u}) = 1 | \\
&(pp, rs_{vk}, rs_{pk}) \leftarrow CKS.Setup(\upsilon), \\
&ck \leftarrow CKS.GenCommitteeKey(rs_{pk}, (pk_i)_{i=1}^{u})), \\
&\pi \leftarrow CKS.Prove(rs_{pk}, ck, (pk_i)_{i=1}^{u}, (bit_i)_{i=1}^{u}), \\
&apk \leftarrow AS.AggKeys(pp, (pk_i)_{i:bit_i=1})] = 1
\end{aligned}
$$

Soundness A *CKS* for aggregatable signatures (*CKS.Setup*, *CKS.GenCommitteeKey*, *CKS.Prove*, *CKS.Verify*) has soundness if for every efficient adversary $\mathcal{A}$ it holds that:

$$
\begin{aligned}
&Pr[CKS.Verify(pp, rs_{vk}, ck, m, asig, \pi, (bit_i)_{i=1}^{u}) = 1 \implies \\
&AS.Verify(pp, apk, m, asig) = 1 | (pp, rs_{vk}, rs_{pk}) \leftarrow CKS.Setup(\upsilon), \\
&(pk_i)_{i=1}^{u}, (bit_i)_{i=1}^{u}, asig, \pi, m \leftarrow \mathcal{A}(pp, rs_{vk}, rs_{pk}), \\
&ck \leftarrow CKS.GenCommitteeKey(rs_{pk}, (pk_i)_{i=1}^{u}), \\
&apk \leftarrow AS.AggKeys(pp, (pk_i)_{i:bit_i=1})] = 1 - negl(\lambda)
\end{aligned}
$$

Next, we define an additional security property, namely *unforgeability*, which intuitively ensures that ...

Unforgeability For a committee key scheme for aggregatable signatures (*CKS.Setup*, *CKS.GenCommitteeKey*, *CKS.Prove*, *CKS.Verify*) the advantage of an adversary $\mathcal{A}$ against unforgeability is defined by $Adv_{\mathcal{A}}^{forgecomkey}(\lambda) = \Pr[Game_{\mathcal{A}}^{forgecomkey}(\lambda) = 1]$, where

$$
\begin{aligned}
&Game_{\mathcal{A}}^{forgecomkey}(\lambda): \\
&(pp, rs_{vk}, rs_{pk}) \leftarrow CKS.Setup(\upsilon), \\
&((pk^*, \pi_{PoP}^*), sk^*) \leftarrow AS.GenKeypair(pp), Q \leftarrow \emptyset \\
&((pk_i, \pi_{PoP,i})_{i=1}^{u}, (bit_i)_{i=1}^{u}, asig, \pi, m) \leftarrow \mathcal{A}^{OSign}(pp, rs_{vk}, \\
&rs_{pk}, (pk^*, \pi_{PoP}^*)) \\
&\textit{If } (\nexists i \in [u], pk^* = pk_i \wedge bit_i = 1) \vee m \in Q, \textit{ then return } 0 \\
&\textit{For } i \in [u] \\
&\quad \textit{If } AS.VerifyPoP(pp, pk_i, \pi_{PoP,i}) = 0 \textit{ return } 0 \\
&ck \leftarrow CKS.GenCommitteeKey(rs_{pk}, (pk_i)_{i=1}^{u}) \\
&\textit{Return } CKS.Verify(pp, rs_{vk}, ck, m, asig, \pi, (bit_i)_{i=1}^{u})
\end{aligned}
$$

and

$$
\begin{aligned}
&OSign(m_j): \\
&\sigma_j \leftarrow AS.Sign(pp, sk^*, m_j); Q \leftarrow Q \cup \{m_j\}; \textit{Return } \sigma_j
\end{aligned}
$$

A committee key scheme for aggregatable signatures is unforgeable if for all efficient adversaries $\mathcal{A}$ it holds that $Adv_{\mathcal{A}}^{forgecomkey}(\lambda) \leq negl(\lambda)$.

Corollary 4.4. *Let AS be an aggregatable signature scheme that fulfils definition 4.1. If CKS is a committee key scheme for aggregatable signatures that fulfils Definition 4.3, then CKS is unforgeable, as defined above.*

Proof Sketch. Assume by contradiction there exists an efficient adversary $\mathcal{A}$ such that $Adv_{\mathcal{A}}^{forgecomkey}(\lambda)$ is non-negligible. Using $\mathcal{A}$ and the soundness property of a committee key scheme, one can construct in a straightforward manner an efficient adversary $\mathcal{A}'$ such that $Adv_{\mathcal{A}'}^{forge}(\lambda) \geq Adv_{\mathcal{A}}^{forgecomkey}(\lambda) - negl(\lambda)$.

This, in turn, implies that $Adv^{forge}_{\mathcal{A}'}$ is non-negligible which contradicts the unforgeability property of aggregatable signature scheme *AS*. Thus, our assumption is false and our statement holds. □

4.4 Conditional NP Relations

By $\mathcal{R} = \{(x; w) : p(x, w) = 1\}$ we denote the binary relation such that (x, w) fulfil predicate $p(x, w) = 1$. We say $\mathcal{R}$ is an NP relation if predicate p can be checked in polynomial time in the length of both inputs x and w and $\mathcal{L}(\mathcal{R}) = \{x \mid \exists w \text{ s.t. } (x, w) \in \mathcal{R}\}$ is an NP language w.r.t. predicate p. In such a case we call x an *instance* and w a *witness*.

To model a specific property of our NP relations, we introduce further notation which we call *conditional NP relation*, we denote it by

$$\mathcal{R}^c = \{(x; w) : (p_1(x, w) = 1 \mid c(x, w) = 1) \ \wedge \ p_2(x, w) = 1\}$$

and we interpret it as the NP relation containing the pairs of inputs and witnesses (x, w) such that $c(x, w) = 1$, $p_1(x, w) = 1$ and $p_2(x, w) = 1$ hold. However, to prove that $(x, w) \in \mathcal{R}^c$ we assume/take it as a given that $c(x, w) = 1$ and we are left to prove only that $p_1(x, w) = 1$ and $p_2(x, w) = 1$ hold. The reason we separate predicate $c(x, w)$ from predicate $p_1(x, w)$ in the definition of $\mathcal{R}^c$ is that predicate $c(x, w)$ may be inefficient to prove inside a proof system (e.g., in our case, inside a SNARK); using the above separation, one can delegate the verification of $c(x, w)$ to a trusted party.
We explicitly include in the definition of any NP relation $\mathcal{R}$ or $\mathcal{R}^c$ the corresponding domain for each type of public input and that input is parsed by the honest parties without additional checks. When we make a statement about an NP relation we can interpret that as one about a conditional relation $\mathcal{R}^c$, where c is the predicate that always outputs 1.

4.5 SNARKs

For proving some security properties in this work, we define and use an extension of commonly employed SNARKs definitions. We call our new notion *hybrid model SNARK*. It is related to the notion of SNARKs with online-offline verifiers [35] where the verifier's computation is split into two parts; however it differs from that since in our case the second part of the verifier's computation is replaced by some portion of public input. In turn, the public input portion may depend on the witness of some NP relation, hence that portion cannot be calculated by the respective verifier himself, and, is computed instead by some (deterministic) process, external to both the prover and the verifier involved. We describe hybrid model SNARKs in full in Appendix Section B and we use them for the security proof in section F.

4.6 Polynomial Protocols, Polynomial Commitments and Lagrange Bases

To prove the security of our custom SNARKs, we start by defining custom vector-based conditional NP relations and we describe for each of them a ranged polynomial protocol. Extending the notion introduced in PLONK [43], we give an updated definition of ranged polynomial protocols in Appendix C. We also use KZG polynomial commitments [44], their batched version and their security definitions as described in PLONK.

4.6.1 Lagrange Bases. For finite field $\mathbb{F}$ we denote by H a subgroup of the multiplicative group of $\mathbb{F}$ such that $n = |H|$ is a large power of 2. Let ω be an n-th root of unity in $\mathbb{F}$ such that ω is a generator of H. A Lagrange base is the polynomial set $\{L_i(X)\}_{0 \le i \le n-1}$, where $\forall i, 0 \le i \le n-1$, $L_i(X)$ is the unique polynomial in $\mathbb{F}_{<n}[X]$ s. t. $L_i(\omega^i) = 1$ and $L_i(\omega^j) = 0, \forall j \ne i$. We denote by block a power of 2 such that block $< n$ and use block for defining one of our conditional NP relations in Section 5. We assume $n = \text{poly}(\lambda)$ and block $= \Theta(\lambda)$ and $|\mathbb{F}| = 2^{\Theta(\lambda)}$.

5 CUSTOM SNARKS FOR AGGREGATION

We construct two related SNARKS, each of them allowing a prover to convince an efficient verifier that an alleged aggregated public key has indeed been computed correctly as an aggregate of a vector of public keys for which two succinct commitments (to the x and y affine coordinates of points) are publicly known. The differences between the two constructions stem from how a *bitvector* with one bit associated to each public key (necessary to signal the inclusion or omission of the respective public key w.r.t. the aggregate key) is used as part of the verifier's public input. We describe a *basic accountable SNARK* (the bitvector is represented as a sequence of 0/1 field elements) and a *packed accountable SNARK* (the bitvector is partitioned into equal blocks of consecutive binary bits which are represented by a field element per block). We finally transform basic and packed accountable SNARKs into SNARKs for building accountable light client systems.
To compile our desired SNARKs we proceed as follows:

- In Sections 5.1 and **??** we define vector-based conditional NP relations $\mathcal{R}^{incl}_{\text{ba}}$ (i.e., basic accountable) and $\mathcal{R}^{incl}_{\text{pa}}$ (packed accountable) and we design two ranged polynomial protocols for these relations. The ranged polynomial protocol notion originates in [43]; we review it (including a refinement) in Section C;
- In Section F we define a two-steps PLONK-inspired compiler which we use to compile the ranged polynomial protocols into SNARKs for two novel mixed vector and trusted polynomial commitments conditional NP relations which we denote by $\mathcal{R}^{incl}_{\text{ba},com}$ and $\mathcal{R}^{incl}_{\text{pa},com}$, respectively.
- In Section 5.3 we give an instantiation for committee key scheme for aggregatable signatures which uses our SNARKS and our instantiation for BLS aggregatable signatures from Section A.0.1.

We define our conditional NP relations over $\mathbb{F}$, i.e., the base field of E_{inn}. Our SNARKs provers' circuits are defined as well over $\mathbb{F}$ as the scalar field of E_{out}. The vector of public keys, which is part of the public input for both of our relations $\mathcal{R}^{incl}_{\text{ba}}$ and $\mathcal{R}^{incl}_{\text{pa}}$, and is denoted by $\mathbf{pk} = (pk_0, \ldots, pk_{n-2})$, is a vector of pairs with each component in $\mathbb{F}$. This vector has size $n - 1$ (n defined in Section 4.6.1). For $\mathcal{R}^{incl}_{\text{ba}}$ we denote the n components bitvector by $\mathbf{bit} = (bit_0, \ldots, bit_{n-1})$ (meaning that each component belongs to the set $\{0, 1\} \subset \mathbb{F}$), while the $\mathcal{R}^{incl}_{\text{pa}}$ relation is defined

using the *compacted bitvector* $\mathbf{b'} = (b'_0, \ldots, b'_{\frac{n}{\text{block}}-1})$ of $\frac{n}{\text{block}}$ field elements, each of which is block binary bits long (block has been defined in Section 4.6.1). Each of the bits in the bit representation of these field elements signals the inclusion (or exclusion) of the index-wise corresponding public keys into the aggregated public key *apk*. The last bit of field element $b'_{\frac{n}{\text{block}}-1}$ as well as the *n*-th component bit_{n-1} do not correspond to any public key, but, as will become clear in the following, they have been included for easier design of constraints.

We denote by H the multiplicative subgroup of $\mathbb{F}$ generated by ω as defined in Section 4.6.1. We denote by $incl(a_0, \ldots, a_{n-2})$ the inclusion predicate that checks if $(a_0, \ldots, a_{n-2}) \in \mathbb{G}^{n-1}_{1,inn}$. Moreover let $h = (h_x, h_y)$ be some fixed, publicly known element in $E_{inn} \setminus \mathbb{G}_{1,inn}$. We denote by (a_x, a_y) the affine representation in x and y coordinates of $a \in E_{inn}$ and by $\oplus$ the point addition in affine coordinates on the elliptic curve E_{inn}. We denote by $\mathbb{B} = \{0, 1\} \subset \mathbb{F}$.

5.1 Basic Accountable Protocol

Conditional Basic Accountable Relation $\mathcal{R}^{incl}_{\text{ba}}$

$$\mathcal{R}^{incl}_{\text{ba}} = \{(\mathbf{pk} \in (\mathbb{F}^2)^{n-1}, \mathbf{bit} \in \mathbb{B}^n, apk \in \mathbb{F}^2; \_) : apk = \sum_{i=0}^{n-2} [bit_i] \cdot pk_i \mid \mathbf{pk} \in \mathbb{G}^{n-1}_{1,inn}\}$$

where $\mathbf{pk} = (pk_0, \ldots, pk_{n-2})$ and $\mathbf{bit} = (bit_0, \ldots, bit_{n-1})$.

Next, we introduce the following Lagrange interpolation polynomials of degree at most $n-1 = |H|$ over cyclic group H (Section 4.6.1): $b(X)$ - interpolates the bits of bitvector *bit*; $pkx(X)$, $pky(X)$ - interpolate all public keys' x and y coordinates, respectively; $kaccx(X)$, $kaccy(X)$ - interpolate x and y coordinates, respectively, of the iterative partial aggregate sum of the actual signing validators' public keys. We also define five polynomial identities $id_1(X), \ldots, id_5(X)$ supporting the following intuition: $id_1(X)$, $id_2(X)$ ensure the x and, respectively, the y coordinates of the iterative partial aggregate sums of actual signing validators public keys (up to each index $i \leq n-2$) follow formulas $(*)$, $(**)$ from Observation 1 which gives all possible cases of complete curve point addition when the second curve point is multiplied by a bit; $id_3(X)$, $id_4(X)$ ensure first partial aggregate sum is h and the total aggregate sum is $h + apk$; this is necessary in order to ensure the addition of the public keys (i.e., elliptic curve points) never falls into condition (3) defined in Observation 1, which recursively implies the partial aggregate sum at every step is a well defined curve point, hence, it is a suitable input for the next step; $id_5(X)$ ensures $b(X)$ evaluates to bits over H. Together, $id_1(X)$ to $id_4(X)$ define the H-ranged polynomial protocol $\mathscr{P}_{\text{ba}}$ for relation $\mathcal{R}^{incl}_{\text{ba}}$; $id_5(X)$ will be used to prove a more general result, applicable also for the H-ranged polynomial protocol $\mathscr{P}_{\text{pa}}$ for relation $\mathcal{R}^{incl}_{\text{pa}}$ (Section ??). In more detail, we have:

Polynomials as Computed by Honest Parties

$$\mathsf{b}(X) = \sum_{i=0}^{n-1} bit_i \cdot \mathsf{L_i}(X);$$

$$\mathsf{pkx}(X) = \sum_{i=0}^{n-2} pkx_i \cdot \mathsf{L_i}(X); \mathsf{pky}(X) = \sum_{i=0}^{n-2} pky_i \cdot \mathsf{L_i}(X)$$

$$\mathsf{kaccx}(X) = \sum_{i=0}^{n-1} kaccx_i \cdot \mathsf{L_i}(X); \mathsf{kaccy}(X) = \sum_{i=0}^{n-1} kaccy_i \cdot \mathsf{L_i}(X),$$

where $(pkx_0, \ldots, pkx_{n-2})$ and $(pky_0, \ldots, pky_{n-2})$ are computed such that $\forall i \in \{0, \ldots, n-2\}$, pk_i is interpreted as a pair (pkx_i, pky_i) with its components in $\mathbb{F}$; we also have $(kaccx_0, kaccy_0) = (h_x, h_y)$ and $(kaccx_{i+1}, kaccy_{i+1}) = (kaccx_i, kaccy_i) \oplus bit_i(pkx_i, pky_i)$, $\forall i < n-1$.

Polynomial Identities

$$id_1(X) = (X - \omega^{n-1}) \cdot [b(X) \cdot ((kaccx(X) - pkx(X))^2 \cdot (kaccx(X) + pkx(X) + kaccx(\omega \cdot X)) - (pky(X) - kaccy(X))^2) + (1 - b(X)) \cdot (kaccy(\omega \cdot X) - kaccy(X))]$$

$$id_2(X) = (X - \omega^{n-1}) \cdot [b(X) \cdot ((kaccx(X) - pkx(X)) \cdot (kaccy(\omega \cdot X) + kaccy(X)) - (pky(X) - kaccy(X)) \cdot (kaccx(\omega \cdot X) - kaccx(X))) + (1 - b(X)) \cdot \cdot (kaccx(\omega \cdot X) - kaccx(X))]$$

$$id_3(X) = (kaccx(X) - h_x) \cdot L_0(X) + (kaccx(X) - (h \oplus apk)_x) \cdot L_{n-1}(X)$$

$$id_4(X) = (kaccy(X) - h_y) \cdot L_0(X) + (kaccy(X) - (h \oplus apk)_y) \cdot L_{n-1}(X)$$

$$id_5(X) = b(X)(1 - b(X)).$$

$id_5(X)$ is not strictly needed for $\mathcal{R}^{incl}_{\text{ba}}$, but for the section **??**.

H-ranged Polynomial Protocol $\mathscr{P}_{\text{ba}}$ for Conditional NP Relation $\mathcal{R}^{incl}_{\text{ba}}$ describes the interaction of the prover $\mathcal{P}_{poly}$, the verifier $\mathcal{V}_{poly}$ and the trusted third party $\mathcal{I}$ in accordance to Definition C.1 from Section C.

$\mathcal{P}_{poly}$ and $\mathcal{V}_{poly}$ know public input $\mathbf{bit} \in \mathbb{B}^n$, $\mathbf{pk} \in (\mathbb{F}^2)^{n-1}$ and $apk \in (\mathbb{F})^2$ which are interpreted as per their domains.

(1) $\mathcal{V}_{poly}$ computes $b(X)$, $pkx(X)$, $pky(X)$.
(2) $\mathcal{P}_{poly}$ sends polynomials $kaccx(X)$ and $kaccy(X)$ to $\mathcal{I}$.
(3) $\mathcal{V}_{poly}$ asks $\mathcal{I}$ to check whether the following polynomial relations hold over range H

$$id_i(X) = 0, \forall i \in [4].$$

(4) $\mathcal{V}_{poly}$ accepts if $\mathcal{I}$'s checks verify.

We show $\mathscr{P}_{\text{ba}}$ is an H-ranged polynomial protocol for conditional NP relation $\mathcal{R}^{incl}_{\text{ba}}$. For this, we first prove:

Claim 5.1. *Assume that* $\forall i < n-1$ *such that* $bit_i = 1$, $pk_i = (pkx_i, pky_i) \in \mathbb{G}_{1,inn}$. *If polynomial identities* $id_i(X) = 0, \forall i \in [5]$, *hold over range* H *and the polynomial* $b(X)$ *has been constructed via interpolation on* H *such that* $b(\omega^i) = bit_i, \forall i < n$ *then* $bit_i \in \mathbb{B} = \{0, 1\} \subset \mathbb{F}, \forall i < n$ $(kaccx_0, kaccy_0) = (h_x, h_y)$, $(kaccx_{n-1}, kaccy_{n-1}) = (h_x, h_y) \oplus (apk_x, apk_y)$, $(kaccx_{i+1}, kaccy_{i+1}) = (kaccx_i, kaccy_i) \oplus bit_i(pkx_i, pky_i)$, $\forall i < n-1$.

Proof. Everything but the last property in the claim is easy to derive from polynomial identities $id_3(X) = 0, id_4(X) = 0, id_5(X) = 0$ holding over H. To prove the remaining property, we remind the incomplete addition formulae for curve points in affine coordinates, over elliptic curve in short Weierstrasse form and state:

Observation 1: Suppose that $bit \in \{0, 1\}$, (x_1, y_1) is a point on an elliptic curve in short Weierstrasse form, and, if $bit = 1$, so is (x_2, y_2). We claim that the following equations:

$$bit((x_1 - x_2)^2(x_1 + x_2 + x_3) - (y_2 - y_1)^2) + (1 - bit)(y_3 - y_1) = 0 \ (*)$$

$$bit((x_1 - x_2)(y_3 + y_1) - (y_2 - y_1)(x_3 - x_1)) + (1 - bit)(x_3 - x_1) = 0 \ (**)$$

hold if and only if one of the following three conditions hold

(1) $bit = 1$ and $(x_1, y_1) \oplus (x_2, y_2) = (x_3, y_3)$ and $x_1 \neq x_2$
(2) $bit = 0$ and $(x_3, y_3) = (x_1, y_1)$
(3) $bit = 1$ and $(x_1, y_1) = (x_2, y_2)$[1].

It is easy to see that each of the conditions 1,2,3 above implies equations $(*)$ and $(**)$. For the implication in the opposite direction, if we assume that $(*)$ and $(**)$ hold, then

Case a: For $bit = 0$, the first term of each equation $(*)$ and $(**)$ vanishes, leaving us with $y_3 - y_1 = 0$ and $x_3 - x_1 = 0$ which are equivalent to condition 2.

Case b: For $bit = 1$ and $x_1 = x_2$, by simple substitution in $(*)$ and $(**)$, we obtain $y_1 = y_2$, i.e., condition 3.

Case c: For $bit = 1$ and $x_1 \neq x_2$, then we can substitute $\beta = \frac{y_2 - y_1}{x_2 - x_1}$ into equations $(*)$ and $(**)$, leaving us with

$$x_1 + x_2 + x_3 = \beta^2 \text{ and } y_3 + y_1 = \beta(x_3 - x_1).$$

which are the usual formulae for short Weierstrass form addition of affine coordinate points when $x_1 \neq x_2$ so this is equivalent to Condition 1.

We apply the above *Observation* by noticing that if $id_1(X)$ and $id_2(X)$ hold over H, then $(*)$ and $(**)$ hold with (x_1, y_1) substituted by $(kaccx_i, kaccy_i)$, (x_2, y_2) substituted by (pkx_i, pky_i), (x_3, y_3) substituted by $(kaccx_{i+1}, kaccy_{i+1})$ and bit substituted by bit_i for $0 \leq i \leq n - 2$. Moreover, since $(kaccx_0, kaccy_0) = (h_x, h_y) \in E_{inn} \setminus \mathbb{G}_{1,inn}$ and if $(pkx_i, pky_i) \in \mathbb{G}_{1,inn}$ whenever $bit_i = 1$, then $\forall i < n-1$ equations $(*)$ and $(**)$ obtained after the substitution defined above are equivalent to either condition 1 or condition 2, but never condition 3, so the result of the sum (i.e., $(kaccx_{i+1}, kaccy_{i+1})$, $0 \leq i \leq n-2$) is, by induction, at each step a well-defined point on the curve. □

Corollary 5.2. *Assume $\forall i < n - 1$ such that $bit_i = 1$, $pk_i = (pkx_i, pky_i) \in \mathbb{G}_{1,inn}$. If the polynomial identities $id_i(X) = 0, \forall i \in [4]$, hold over range H and $bit_i \in \mathbb{B}$, $\forall i < n - 1$ and $b(X) = \sum_{i=0}^{n-1} bit_i \cdot L_i(X)$ then:*
$(kaccx_0, kaccy_0) = (h_x, h_y)$, $(kaccx_{n-1}, kaccy_{n-1}) = (h_x, h_y)$
$\oplus\ (apk_x, apk_y)$, $(kaccx_{i+1}, kaccy_{i+1}) = (kaccx_i, kaccy_i)$
$\oplus\ bit_i(pkx_i, pky_i)$, $\forall i < n - 1$.

Proof. The proof follows trivially from the general result stated by Claim 5.1. □

[1]Note that under condition 3, (x_3, y_3) can be any point whatsoever, maybe not even on the curve. The same holds true for (x_2, y_2) under the condition 2.

Lemma 5.3. *$\mathscr{P}_{\text{ba}}$ as described above is an H-ranged polynomial protocol for conditional NP relation $\mathcal{R}_{\text{ba}}^{incl}$.*

Proof. If $(\mathbf{bit}, \mathbf{pk}, apk) \in \mathcal{R}_{\text{ba}}^{incl}$ holds, meaning that $\mathbf{bit} \in \mathbb{B}^n$ and $\mathbf{pk} \in \mathbb{G}_{1,inn}^{n-1}$ and $apk = \sum_{i=0}^{n-2}[bit_i] \cdot pk_i$ hold, then it is easy to see that the honest prover $\mathcal{P}_{poly}$ in $\mathscr{P}_{\text{ba}}$ will convince the honest verifier $\mathcal{V}_{poly}$ in $\mathscr{P}_{\text{ba}}$ to accept with probability 1 so perfect completeness holds. For knowledge-soundness, if the verifier $\mathcal{V}_{poly}$ in $\mathscr{P}_{\text{ba}}$ accepts, then the extractor $\mathcal{E}$ is trivial since $\mathcal{R}_{\text{ba}}^{incl}$ has no witness. We need to show that if $\mathbf{pk} \in \mathbb{G}_{1,inn}^{n-1}$ and the verifier in $\mathscr{P}_{\text{ba}}$ accepts, then $(\mathbf{bit}, \mathbf{pk}, apk) \in \mathcal{R}_{\text{ba}}^{incl}$ holds, which given our definition for conditional relation is equivalent to proving that $apk = \sum_{i=0}^{n-2}[bit_i] \cdot pk_i$ holds. This is due to Corollary 5.2. □

5.2 Our Custom SNARKs

To design our accountable light client systems as introduced in Section 2.2 and fully detailed in Section I, we need to further refine and then compile relations $\mathcal{R}_{\text{ba}}^{incl}$ and $\mathcal{R}_{\text{pa}}^{incl}$ into appropriate SNARKs such that the long public input $\mathbf{pk} \in (\mathbb{F}^2)^{n-1}$ is replaced by a pair of succinct commitments and, $\mathbf{pk}$ becomes a witness for the resulting refined relations. As far as we are aware, such a compiler does not exist. Thus, we design a two-step compiler to obtain SNARKs for relations with such properties. We are interested in relations below.

$$\mathcal{R}_{\text{ba},com}^{incl} = \{(\mathbf{C} \in C, \mathbf{bit} \in \mathbb{B}^n, apk \in \mathbb{F}^2; \mathbf{pk}) : apk = \sum_{i=0}^{n-2}[bit_i] \cdot pk_i \mid \mathbf{pk} \in \mathbb{G}_{1,inn}^{n-1} \ \wedge \ \mathbf{C} = \mathbf{Com}(\mathbf{pk})\}$$

$$\mathcal{R}_{\text{pa},com}^{incl} = \{(\mathbf{C} \in C, \mathbf{b}' \in \mathbb{F}^{\frac{n}{|\text{block}|}}_{\text{block}}, apk \in \mathbb{F}^2; \mathbf{pk}, \mathbf{bit}) : apk = \sum_{i=0}^{n-2}[bit_i] \cdot pk_i \mid \mathbf{pk} \in \mathbb{G}_{1,inn}^{n-1} \ \wedge \ \mathbf{bit} \in \mathbb{B}^n \ \wedge \ \wedge\ b'_j = \sum_{i=0}^{\text{block}-1} 2^i \cdot bit_{\text{block}\cdot j+i}, \forall j < \frac{n}{\text{block}} \ \wedge \ \mathbf{C} = \mathbf{Com}(\mathbf{pk})\}$$

The two-step compiler is described in Section F, with the first step being the standard PLONK compiler [43] and the second step being amenable for compiling into SNARKs a generalisation of the two NP relations just detailed above.

5.3 Our Instantiation for CKS

Given relations $\mathcal{R}_{\text{ba},com}^{incl}$ and $\mathcal{R}_{\text{pa},com}^{incl}$ described in short in Section 5.2 and , we present an instantiation for committee key scheme defined in Section 4.3; this is used to build an accountable light client system . We instantiate u and v from Section 4.3 as $u = n - 1$, ($n = |H|$ from Section 4.6.1) and $v \in \mathbb{N}, n - 1 \leq v$, $v = \text{poly}(\lambda)$, where v is the maximum number of validators.

Instantiation 5.4. *(Committee Key Scheme for Aggregatable Signatures) In our implementation we use the following instantiation of Definition 4.3 for one of $\mathcal{R} \in \{\mathcal{R}_{\text{ba},com}^{incl}, \mathcal{R}_{\text{pa},com}^{incl}\}$:*

- *$CKS_{\mathcal{R}}.Setup(v)$ calls algorithms:*

(1) $pp \leftarrow AS.Setup(aux_{AS} = \upsilon + 1)$ *with* $AS.Setup$ *part of Instantiation A.2 and* $\mathbb{G}_{1,inn}$ *part of* pp *(see notation in Section A.0.1);*
(2) $srs \leftarrow SNARK.Setup(aux_{SNARK} = (\upsilon, 3\upsilon))$ *with* $srs = ([1]_{1,out}, [\tau]_{1,out}, \ldots, [\tau^{3\upsilon}]_{1,out}, [1]_{2,out}, [\tau]_{2,out})$;
(3) $(rs_{pk}, rs_{vk}) \leftarrow SNARK.KeyGen(srs, \mathcal{R})$ *with* $(rs_{pk}, rs_{vk}) = (([1]_{1,out}, [\tau]_{1,out}, \ldots, [\tau^{3\upsilon}]_{1,out}), ([1]_{1,out}, [1]_{2,out}, [\tau]_{2,out}))$ *where the notation* $[\ldots]_{1,out}$ *and* $[\ldots]_{2,out}$ *was defined in Section 4.1.*

- $ck \leftarrow CKS_{\mathcal{R}}.GenCommitteeKey(rs_{pk}, (pk_i)_{i=1}^{n-1})$, *where* $CKS_{\mathcal{R}}.GenCommitteeKey$ *first checks whether* $(pk_i)_{i=1}^{n-1} \in \mathbb{G}_{1,inn}^{n-1}$; *if this does not hold, it outputs* $\perp$; *otherwise,*
  $CKS_{\mathcal{R}}.GenCommitteeKey$ *continues as:*
  Let $\mathbf{pkx} = (pkx_1, \ldots, pkx_{n-1})$, $\mathbf{pky} = (pky_1, \ldots, pky_{n-1})$, $\forall i \in [n-1]$, $pk_i = (pkx_i, pky_i) \in \mathbb{F}^2$.
  Let $pkx(X) = \sum_{i=0}^{n-2} pkx_{i+1} \cdot L_i(X)$, $pky(X) = \sum_{i=0}^{n-2} pky_{i+1} \cdot L_i(X)$. *Let* $[pkx]_{1,out} = pkx(\tau) \cdot [1]_{1,out}$, $[pky]_{1,out} = pky(\tau) \cdot [1]_{1,out}$. *Output* $ck = ([pkx]_{1,out}, [pky]_{1,out})$.
  Note that $\mathbb{F}$ *and* $\{L_i(X)\}_{i=1}^{n-2}$ *are as defined in Section 4.6.1.*
- $\pi \leftarrow CKS_{\mathcal{R}}.Prove(rs_{pk}, ck, (pk_i)_{i=1}^{n-1}, (bit_i)_{i=1}^{n-1})$ *where* $\pi = (\pi_{SNARK}, apk)$ *and* $CKS_{\mathcal{R}}.Prove$ *calls* $apk = \sum_{i=1}^{n-1} bit_i \cdot pk_i \leftarrow AS.AggregateKeys(pp, (pk_i)_{i:bit_i=1})$ *as defined in Instantiation A.2 and* $\pi_{SNARK} \leftarrow SNARK.Prove(rs_{pk}, (x, w), \mathcal{R})$, *for* $\mathcal{R} \in \{\mathcal{R}^{incl}_{\text{ba},com}, \mathcal{R}^{incl}_{\text{pa},com}\}$ *where*

$$\begin{cases} (x = (ck, (bit_i)_{i=1}^{n-1}||0, apk), w = ((pk_i)_{i=1}^{n-1}) & \text{if } \mathcal{R} = \mathcal{R}^{incl}_{\text{ba},com}, \\ (x = (ck, \mathbf{b}', apk), w = ((pk_i)_{i=1}^{n-1}, (bit_i)_{i=1}^{n-1}||0) & \text{if } \mathcal{R} = \mathcal{R}^{incl}_{\text{pa},com}, \end{cases}$$

where $\mathbf{b}'$ *is the vector of field elements formed from blocks of size* block *of bits from vector* $(bit_i)_{i=1}^{n-1}||0$ *and* block *is the highest power of 2 smaller than the size of a field element in* $\mathbb{F}$.

- $0/1 \leftarrow CKS_{\mathcal{R}}.Verify(pp, rs_{vk}, ck, m, asig, \pi, \textbf{bitvector})$ *parses* π *to retrieve* π_{SNARK} *and* apk *and it calls* $AS.Verify(pp, apk, m, asig)$ *as defined in Instantiation A.2 and it also calls* $SNARK.Verify(rs_{vk}, x, \pi_{SNARK}, \mathcal{R})$ *(where* π_{SNARK}, x *and* $\mathcal{R}$ *are as defined in the paragraph above with the only difference that* $(bit_i)_{i=1}^{n-1}$ *represents the first* $n-1$ *bits of* **bitvector**, *padded with* 0s, *if not sufficiently many exist in* **bitvector**); *it outputs* 1 *if both algorithms output* 1 *and it outputs* 0 *otherwise.*

Theorem 5.5. *The committee key scheme* $CKS_{\mathcal{R}}$ *in Instantiation 5.4 is secure with respect to Definition 4.3.*

Proof. We give a full proof in Appendix H. □

6 CONCLUSIONS

In this work we have defined, designed, proved and implemented the first accountable light client system. This is provably secure and very efficient and can be further integrated as the core building block into secure bridges between SNARK friendly PoS blockchains. Our work is planned to be integrated into a bridge between Polkadot and Kusama, as part of the Polkadot ecosystem.

REFERENCES

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *Decentralized Business Review*, p. 21260, 2008.
[2] J. Burdges, A. Cevallos, P. Czaban, R. Habermeier, S. Hosseini, F. Lama, H. K. Alper, X. Luo, F. Shirazi, A. Stewart, and G. Wood, "Overview of polkadot and its design considerations," 2020. https://arxiv.org/abs/2005.13456.
[3] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, 2014.
[4] J. Kwon and E. Buchman, "Cosmos whitepaper: A network of distributed ledgers." https://v1.cosmos.network/resources/whitepaper.
[5] E. Buchman, J. Kwon, and Z. Milosevic, "The latest gossip on bft consensus," 2018. https://arxiv.org/abs/1807.04938.
[6] eLabs Team, "The celo protocol: A multi-asset cryptographic protocol for decentralized social payments." https://celo.org/papers/whitepaper.
[7] T. D. Team, "The internet computer for geeks," 2022. https://internetcomputer.org/whitepaper.pdf.
[8] "Over $1 billion stolen from bridges so far in 2022 as harmony's horizon bridge becomes latest victim in $100 million hack," 2022. bit.ly/3fvlIME.
[9] "Nomad loses $156 million in seventh major crypto bridge exploit of 2022," 2022. https://hub.elliptic.co/analysis/nomad-loses-156-million-in-seventh-major-crypto-bridge-exploit-of-2022/.
[10] C. Goes, "The interblockchain communication protocol: An overview," 2020. https://arxiv.org/abs/2006.15918.
[11] Mintscan, "Validator dashboard for nyx," Accessed 19.01.2023. https://www.mintscan.io/nyx/validators.
[12] Mintscan, "Validator dashboard for cosmos hub," Accessed 19.01.2023. https://www.mintscan.io/cosmos/validators.
[13] C. Organization, "Optics," Accessed 19.01.2023. https://docs.celo.org/protocol/bridge/optics.
[14] N. Foundation, "Rainbow bridge," Accessed 19.01.2023. https://wiki.near.org/getting-started/rainbow-bridge.
[15] SECBIT, "How the winner got fomo3d prize — a detailed explanation," 2018. https://medium.com/coinmonks/how-the-winner-got-fomo3d-prize-a-detailed-explanation-b30a69b7813f.
[16] N. Foundation, "Rainbow bridge faq," Accessed 19.01.2023. https://rainbowbridge.app/faq.
[17] T. Xie, J. Zhang, Z. Cheng, F. Zhang, Y. Zhang, Y. Jia, D. Boneh, and D. X. Song, "zkbridge: Trustless cross-chain bridges made practical," *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022.
[18] V. Buterin and V. Griffith, "Casper the friendly finality gadget," *arXiv preprint arXiv:1710.09437*, 2017. https://arxiv.org/abs/1710.09437.
[19] S. Kannan, K. Nayak, P. Sheng, P. Viswanath, and G. Wang, "Bft protocol forensics," 2020. https://arxiv.org/abs/2010.06785.
[20] A. Stewart and E. Kokoris-Kogia, "Grandpa: a byzantine finality gadget," 2020. https://arxiv.org/abs/2007.01560.
[21] P. Civit, S. Gilbert, V. Gramoli, R. Guerraoui, and J. Komatovic, "As easy as abc: Optimal (a) ccountable (b) yzantine (c) onsensus is easy!," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 560–570, IEEE, 2022.
[22] "Beefy." https://github.com/paritytech/grandpa-bridge-gadget/blob/master/docs/beefy.md.
[23] D. Boneh, B. Lynn, and H. Shacham, "Short signatures from the weil pairing," in *ASIACRYPT 2001*, pp. 514–532, 2001.
[24] D. Boneh, M. Drijvers, and G. Neven, "Compact multi-signatures for smaller blockchains," in *ASIACRYPT 2018*, pp. 435–464, 2018.
[25] "Minimal light client," 2021. Commit of 14th Sept 2021, https://github.com/ethereum/annotated-spec/blob/master/altair/sync-protocol.md.
[26] J. Groth, "Non-interactive distributed key generation and key resharing." ePrint 2021/339, 2021.
[27] K. Gurkan, P. Jovanovic, M. Maller, S. Meiklejohn, G. Stern, and A. Tomescu, "Aggregatable distributed key generation." ePrint 2021/005, 2021.
[28] C. Gentry, S. Halevi, and V. Lyubashevsky, "Practical non-interactive publicly verifiable secret sharing with thousands of parties." ePrint 2021/1397, 2021.
[29] J. Bonneau, I. Meckler, V. Rao, and E. Shapiro, "Coda: Decentralized cryptocurrency at scale." ePrint 2020/352, 2020.
[30] A. Gabizon, K. Gurkan, P. Jovanovic, G. Konstantopoulos, A. Oines, M. Olszewski, M. Straka, E. Tromer, and P. Vesely, "Plumo: Towards scalable interoperable blockchains using ultralight validation systems." 3rd ZKStandards Workshop, 2020.
[31] J. Groth, "On the size of pairing-based non-interactive arguments," in *EUROCRYPT 2016*, pp. 305–326, 2016.
[32] J. Kilian, "Uses of randomness in algorithms and protocols." PhD Thesis, 1990. https://core.ac.uk/download/pdf/4425126.pdf.
[33] R. Canetti, Y. Lindell, R. Ostrovsky, and A. Sahai, "Universally composable two-party and multi-party secure computation," in *STOC02*, 2002.

[34] D. Benarroch, M. Campanelli, D. Fiore, J. Kim, J. Lee, H. Oh, and A. Querol, "Proposal: Commit-and-prove zero-knowledge proof systems and extensions," 2021. 4rd ZKStandards Workshop.

[35] D. Fiore, C. Fournet, E. Ghosh, M. Kohlweiss, O. Ohrimenko, and B. Parno, "Hash first, argue later: Adaptive verifiable computations on outsourced data," in *CCS'16*, pp. 1304–1316, 2016.

[36] D. F. Aranha, E. M. Bennedsen, M. Campanelli, C. Ganesh, C. Orlandi, and A. Takahashi, "Eclipse: Enhanced compiling method for pedersen-committed zksnark engines." Cryptology ePrint Archive, Paper 2021/934, 2021. https://eprint.iacr.org/2021/934.

[37] "Simplified active validator cap and rotation proposal," 2022. https://ethresear.ch/t/simplified-active-validator-cap-and-rotation-proposal/9022.

[38] S. Yonezawa, "Pairing-friendly curves," 2020. https://tools.ietf.org/id/draft-yonezawa-pairing-friendly-curves-02.html.

[39] S. Galbraith, K. Paterson, and N. Smart, "Pairings for cryptographers." ePrint 2006/165, 2006.

[40] S. Bowe, A. Chiesa, M. Green, I. Miers, P. Mishra, and H. Wu, "Zexe: Enabling decentralized private computation," in *Security and Privacy 2020*, pp. 947–964, 2020.

[41] Y. E. Housni and A. Guillevic, "Optimized and secure pairing-friendly elliptic curves suitable for one layer proof composition." ePrint 2020/351, 2020.

[42] T. Ristenpart and S. Yilek, "The power of proofs-of-possession: Securing multi-party signatures against rogue-key attacks," in *EUROCRYPT 2007*, pp. 228–245, 2007.

[43] A. Gabizon, Z. J. Williamson, and O. Ciobotaru, "Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge." ePrint 2019/953, 2019.

[44] A. Kate and I. G. Gregory M Zaverucha, "Constant-size commitments to polynomials and their applications," in *ASIACRYPT10*, pp. 177–194, 2010.

[45] J. Groth, M. Kohlweiss, M. Maller, S. Meiklejohn, and I. Miers, "Updatable and universal common reference strings with applications to zk-snarks," in *CRYPTO 2018*, pp. 698–728, 2018.

[46] G. Fuchsbauer, E. Kiltz, and J. Loss, "The algebraic group model and its applications," in *CRYPTO 2018*, pp. 33–62, 2018.

[47] N. Bitansky, R. Canetti, O. Paneth, and A. Rosen, "On the existence of extractable one-way functions," in *STOC '14*, pp. 505–514, 2014.

[48] E. Boyle and R. Pass, "Limits of extractability assumptions with distributional auxiliary input," in *ASIACRYPT 2015*, pp. 236–261, 2015.

[49] M. Bellare, J. A. Garay, and T. Rabin, "Fast batch verification for modular exponentiation and digital signatures." ePrint 1998/007, 1998.

[50] A. Kattis, K. Panarin, and A. Vlasov, "Redshift: Transparent snarks from list polynomial commitment iops." ePrint 2019/1400, 2019.

[51] S. Bowe, J. Grigg, and D. Hopwood, "Recursive proof composition without a trusted setup." ePrint 2019/1021, 2019.

[52] A. Fiat and A. Shamir, "How to prove yourself: Practical solutions to identification and signature problems," in *CRYPTO' 86*, pp. 186–194, 1987.

[53] D. Pointcheval and J. Stern, "Security proofs for signature schemes," in *EUROCRYPT '96*, pp. 387–398, 1996.

[54] C. Ganesh, H. Khoshakhlagh, M. Kohlweiss, A. Nitulescu, and M. Zajac, "What makes fiat–shamir zksnarks (updatable srs) simulation extractable?." ePrint 2021/511, 2021.

[55] Y. E. Housni and A. Guillevic, "Bw6 over bls12-381," 2021. https://ethresear.ch/t/bw6-over-bls12-381/10321.

[56] F. of Ethereum Magician and E. C. Herders, "Ethereum improvement proposals." https://eips.ethereum.org.

[57] V. Buterin, "Minimal light client," 2021. https://github.com/ethereum/annotated-spec/blob/master/altair/sync-protocol.md.

[58] V. Buterin, D. Hernandez, T. Kamphefner, K. Pham, Z. Qiao, D. Ryan, J. Sin, Y. Wang, and Y. X. Zhang, "Combining ghost and casper," *arXiv*, 2020. https://arxiv.org/abs/2003.03052.

[59] B. Bünz, L. Kiffer, L. Luu, and M. Zamani, "Flyclient: Super-light clients for cryptocurrencies." Cryptology ePrint Archive, Paper 2019/226, 2019. https://eprint.iacr.org/2019/226.

APPENDICES

A AGGREGATABLE SIGNATURE SCHEME DEFINITION

Definition A.1. (Aggregatable Signature Scheme) An aggregatable signature scheme consists of the following tuple of algorithms ($AS.Setup$, $AS.GenKeypair$, $AS.VerifyPoP$, $AS.Sign$, $AS.AggKeys$, $AS.AggSigs$, $AS.Verify$) such that for implicit security parameter λ:

- $pp \leftarrow AS.Setup(aux_{AS})$: a setup algorithm that, given an auxiliary parameter aux_{AS}, outputs public protocol parameters pp.
- $((pk, \pi_{PoP}), sk) \leftarrow AS.GenKeypair(pp)$: a key pair generation algorithm that outputs a secret key sk, and the corresponding public key pk together with a proof of possession π_{PoP} for the secret key.
- $0/1 \leftarrow AS.VerifyPoP(pp, pk, \pi_{PoP})$: a public key verification algorithm that, given a public key pk and a proof of possession π_{PoP}, outputs 1 if π_{PoP} is valid for pk and 0 otherwise.
- $\sigma \leftarrow AS.Sign(pp, sk, m)$: a signing algorithm that, given a secret key sk and a message m in $\{0,1\}^*$, returns a signature σ.
- $apk \leftarrow AS.AggKeys(pp, (pk_i)_{i=1}^u)$: a public key aggregation algorithm that, given a vector of public keys $(pk_i)_{i=1}^u$, returns an aggregate public key apk.
- $asig \leftarrow AS.AggSigs(pp, (\sigma_i)_{i=1}^u)$: a signature aggregation algorithm that, given a vector of signatures $(\sigma_i)_{i=1}^u$, returns an aggregate signature $asig$.
- $0/1 \leftarrow AS.Verify(pp, apk, m, asig)$: a signature verification algorithm that, given an aggregate public key apk, a message $m \in \{0,1\}^*$, and an aggregate signature σ, returns 1 or 0 to indicate if the signature is valid.

We say ($AS.Setup$, $AS.GenKeypair$, $AS.VerifyPoP$, $AS.Sign$, $AS.AggKeys$, $AS.AggSigs$, $AS.Verify$) is an aggregatable signature scheme if it satisfies *perfect completeness* and *perfect completeness for aggregation* and *unforgeability* as defined below.

Perfect Completeness An aggregatable signature scheme AS has perfect completeness if for any message $m \in \{0,1\}^*$ and any $u \in \mathbb{N}$ it holds that:

$$
\begin{aligned}
&Pr[AS.Verify(pp, apk, m, asig) = 1 \wedge \\
&\forall i \in [u]\ AS.VerifyPoP(pp, pk_i, \pi_{PoP,i}) = 1 \mid \\
&pp \leftarrow AS.Setup(aux_{AS}), \\
&((pk_i, \pi_{PoP,i}), sk_i) \leftarrow AS.GenerateKeypair(pp),\ i = 1, \ldots, u \\
&apk \leftarrow AggregateKeys(pp, (pk_i)_{i=1}^u), \\
&\sigma_i \leftarrow AS.Sign(pp, sk_i, m),\ i = 1, \ldots, u, \\
&asig \leftarrow AS.AggregateSignatures(pp, (\sigma_i)_{i=1}^u)] = 1.
\end{aligned}
$$

We note that an aggregatable signature scheme with perfect completeness implies the underlying signature scheme has perfect completeness.

Perfect Completeness for Aggregation An aggregatable signature scheme AS has perfect completeness for aggregation if, for every adversary $\mathcal{A}$

$$
\begin{aligned}
&Pr[AS.Verify(pp, apk, m, asig) = 1 \mid pp \leftarrow AS.Setup(aux_{AS}), \\
&((pk_i)_{i=1}^u, m, (\sigma_i)_{i=1}^u) \leftarrow \mathcal{A}(pp)\ such\ that\ \forall i \in [u], \\
&AS.Verify(pp, pk_i, m, \sigma_i) = 1, \\
&apk \leftarrow AS.AggKeys(pp, (pk_i)_{i=1}^u), \\
&asig \leftarrow AS.AggSigs(pp, (\sigma_i)_{i=1}^u)] = 1.
\end{aligned}
$$

Unforgeable Aggregatable Signature For an aggregatable signature scheme AS, the advantage of an adversary against unforgeability is defined by

$$Adv_{\mathcal{A}}^{forge}(\lambda) = Pr[Game_{\mathcal{A}}^{forge}(\lambda) = 1]$$

where

$$
\begin{aligned}
&Game^{forge}_{\mathcal{A}}(\lambda):\\
&pp \leftarrow AS.Setup(aux_{AS})\\
&((pk^*, \pi^*_{PoP}), sk^*) \leftarrow AS.GenerateKeypair(pp)\\
&Q \leftarrow \emptyset\\
&((pk_i, \pi_{PoP,i})_{i=1}^{u}, m, asig) \leftarrow \mathcal{A}^{OSign}(pp, (pk^*, \pi^*_{PoP}))\\
&If\ pk^* \notin \{pk_i\}_{i=1}^{u} \vee m \in Q,\ then\ return\ 0\\
&For\ i \in [u]\\
&\quad If\ AS.VerifyPoP(pp, pk_i, \pi_{PoP,i}) = 0\ return\ 0\\
&apk \leftarrow AS.AggKeys(pp, (pk_i)_{i=1}^{u})\\
&Return\ AS.Verify(pp, apk, m, asig)
\end{aligned}
$$

and

$$
\begin{aligned}
&OSign(m_j):\\
&\sigma_j \leftarrow AS.Sign(pp, sk^*, m_j)\\
&Q \leftarrow Q \cup \{m_j\}\\
&Return\ \sigma_j
\end{aligned}
$$

and $\mathcal{A}^{OSign}$ denotes the adversary $\mathcal{A}$ with access to oracle *OSign*.

We say an aggregatable signature scheme is unforgeable if for all efficient adversaries $\mathcal{A}$ it holds that $Adv^{forge}_{\mathcal{A}}(\lambda) \leq negl(\lambda)$.

A.0.1 An Aggregatable Signature Instantiation. In the following, we instantiate the aggregatable signature definition given above with a scheme inspired by the BLS signature scheme [23] and its follow-up variants [24, 42].

Instantiation A.2. *(Aggregatable Signatures) In our implementation we call aggregatable signatures the following instantiation of aggregatable signatures definition. Note that in our implementation we instantiate E_{inn} with BLS12-377 [40].*

- *$(\mathbb{G}_{1,inn}, g_{1,inn}, \mathbb{G}_{2,inn}, g_{2,inn}, \mathbb{G}_{T,inn}, e_{inn}, H_{inn}, H_{PoP})$ from pp where $pp \leftarrow AS.Setup(aux_{AS})$, where $\mathbb{G}_{1,inn}$, $g_{1,inn}$, $\mathbb{G}_{2,inn}$, $g_{2,inn}$, $\mathbb{G}_{T,inn}$, e_{inn} were defined in Section 4.1 and $H_{inn} : \{0,1\}^* \to \mathbb{G}_{2,inn}$ and $H_{PoP} : \{0,1\}^* \to \mathbb{G}_{2,inn}$ are two hash functions. The auxiliary parameter aux_{AS} is such that there exists $N \in \mathbb{N}$, N is the first component of the vector aux_{AS} and there exists a subgroup of size at least N in the multiplicative group of $\mathbb{F}$, where $\mathbb{F}$ is the base field of E_{inn}, but also the size of the subgroup $\in O(N)$.*
- *$(pk, sk, \pi_{inn}) \leftarrow AS.GenKeypair(pp)$, where $sk \xleftarrow{\$} \mathbb{Z}^*_r$ and $pk = sk \cdot g_{1,inn} \in \mathbb{G}_{1,inn}$ and $\pi_{inn} \leftarrow sk \cdot H_{PoP}(pk)$ and r was defined in Section 4.1 as the characteristic of the scalar field of E_{inn}.*
- *$0/1 \leftarrow AS.VerifyPoP(pp, pk, \pi_{inn})$, where AS.VerifyPoP outputs 1 if*

  $$e_{inn}(g_{1,inn}, \pi_{inn}) = e_{inn}(pk, H_{PoP}(pk))$$

  holds and 0 otherwise. Note that implicitly, as part of running AS.VerifyPoP, one checks that $pk \in \mathbb{G}_{1,inn}$ also holds.
- *$\sigma \leftarrow AS.Sign(pp, sk, m)$: where $\sigma = sk \cdot H_{inn}(m) \in \mathbb{G}_{2,inn}$.*
- *$apk \leftarrow AS.AggKeys(pp, (pk_i)_{i=1}^{u})$, where $apk = \sum_{i=1}^{u} pk_i$. Note that AS.AggKeys checks whether $((pk_i)_{i=1}^{u}) \in \mathbb{G}^{u}_{1,inn}(*)$ and, if that is not the case, it outputs $\perp$; if $(*)$ holds, the algorithm AS.AggKeys continues with the computations described above.*
- *$asig \leftarrow AS.AggSigs(pp, (\sigma_i)_{i=1}^{u})$, where $asig = \sum_{i=1}^{u} \sigma_i$.*
- *$0/1 \leftarrow AS.Verify(pp, apk, m, asig)$, where AS.Verify outputs 1 if $apk \neq \perp$ and $apk \in \mathbb{G}_{1,inn}$ and $e_{inn}(apk, H_{inn}(m)) = e_{inn}(g_{1,inn}, asig)$; otherwise, it outputs 0.*

B HYBRID MODEL SNARKS

When proving the security of our arguments, we use an extension of some of the more commonly employed SNARK definitions which we call a "a hybrid model SNARK". This resembles the existing notion of SNARKs with online-offline verifiers as described in [35], where the verifier computation is split into two parts: during the offline phase some computation (possibly of commitments) happens; this computation takes some public inputs as parameters and, when not performed by the verifier, it may also be performed (in part) by the prover. The online phase is the main computation performed by the verifier. In the case of our hybrid model SNARKs, however, the input to the offline counterpart described above (which we call the *PartInput* algorithm) may even be the witness or a part of the witness for the respective relation. For our custom SNARKs, *PartInput* produces part of the public input used by the verifier; since for our use case, *PartInput* does handle a portion of the witness, this operation cannot be performed by the verifier for

that relation. Moreover, in our instantiation, *PartInput* produces computationally binding commitment schemes that are opened by the prover. Both of these properties are not explicitly part of our general definition for hybrid model SNARKs, but they are crucial and explicitly assumed and used in proving the security for our compiler's second step (see Appendix F).

The two SNARKs we design in this work have access to a *structured reference string* (srs) of the form $(\{[\tau^i]_1\}_{i=0}^{d}, \{[\tau^i]_2\}_{i=0}^{1})$ where τ is a random (and allegedly secret) value in $\mathbb{F}$ and d is bounded by a polynomial in λ. Such an srs is universal and updatable [45]. We introduce a generalisation of the usual SNARK definition which we call a *hybrid model SNARK* inspired by online-offline SNARKs [35]. We further refine it as described below:

Definition B.1. (Hybrid Model SNARK) A hybrid model *succinct non-interactive argument of knowledge for relation* $\mathcal{R}$ is a tuple of PPT algorithms ($SNARK.Setup$, $SNARK.KeyGen$, $SNARK.Prove$, $SNARK.Verify$, $SNARK.PartInputs$) such that for implicit security parameter λ:

- $srs \leftarrow SNARK.Setup(aux_{SNARK})$: a setup algorithm that on input auxiliary parameter aux_{SNARK} from some domain $\mathcal{D}$ outputs a universal structured reference string tuple srs,
- $(srs_{pk}, srs_{vk}) \leftarrow SNARK.KeyGen(srs, \mathcal{R})$: a key generation algorithm that on input a universal structured reference string srs and an NP relation $\mathcal{R}$ outputs a *proving key* and a *verification key* pair (srs_{pk}, srs_{vk}),
- $\pi \leftarrow SNARK.Prove(srs_{pk}, (x, w), \mathcal{R})$: a proof generation algorithm that on input a proving key srs_{pk} and a pair $(x, w) \in \mathcal{R}$ outputs *proof* π,
- $0/1 \leftarrow SNARK.Verify(srs_{vk}, x, \pi, \mathcal{R})$: a proof verification algorithm that on input a verification key srs_{vk}, an instance x and a proof π outputs a bit that signals acceptance (if output is 1) or rejection (if output is 0),
- $(x_1, state_2) \leftarrow SNARK.PartInputs(srs, state_1, \mathcal{R})$: a deterministic public inputs generation algorithm that takes as input a universal structured reference string srs, an NP relation $\mathcal{R}$ and state $state_1$ and outputs updated state $state_2$ and partial public input x_1,

and satisfies completeness, knowledge soundness w.r.t. $SNARK.PartInputs$ and succinctness as defined below:

Perfect Completeness holds if an honest prover will always convince an honest verifier: for all $(x, w) \in \mathcal{R}$ and for all $aux_{SNARK} \in \mathcal{D}$

$$Pr[SNARK.Verify(srs_{vk}, x, \pi, \mathcal{R}) = 1 \mid srs \leftarrow SNARK.Setup(aux_{SNARK}),$$
$$(srs_{pk}, srs_{vk}) \leftarrow SNARK.KeyGen(srs, \mathcal{R}), \pi \leftarrow SNARK.Prove(srs_{pk}, (x, w), \mathcal{R})] = 1.$$

Notation We denote by $State_{\mathcal{R}}$ the set of all states $state_1$ such that given some relation $\mathcal{R}$ and any possible srs, for any output x_1 of $SNARK.PartInputs(srs, state_1, \mathcal{R})$ with $state_1 \in State_{\mathcal{R}}$, there exists x_2 and w with $(x = (x_1, x_2), w) \in \mathcal{R}$; we further assume $State_{\mathcal{R}} \neq \emptyset$.

Knowledge-soundness with respect to $SNARK.PartInputs$ holds if there exists a PPT extractor $\mathcal{E}$ such that for all PPT adversaries $\mathcal{A}$, for all $aux_{SNARK} \in \mathcal{D}$ and for all $state_1 \in State_{\mathcal{R}}$

$$Pr[(x = (x_1, x_2), w) \in \mathcal{R} \wedge 1 \leftarrow SNARK.Verify(srs_{vk}, x = (x_1, x_2), \pi, \mathcal{R}) \mid$$
$$srs \leftarrow SNARK.Setup(aux_{SNARK}), (srs_{pk}, srs_{vk}) \leftarrow SNARK.KeyGen(srs, \mathcal{R}),$$
$$(x_1, state_2) \leftarrow SNARK.PartInput(srs, state_1, \mathcal{R}), (x_2, \pi) \leftarrow \mathcal{A}(srs, state_2, \mathcal{R}),$$
$$w \leftarrow \mathcal{E}^{\mathcal{A}}(srs, state_2, \mathcal{R})]$$

is overwhelming in λ, where by $\mathcal{E}^{\mathcal{A}}$ we denote the extractor $\mathcal{E}$ that has access to all of $\mathcal{A}$'s messages during the protocol with the honest verifier.

Succinctness holds if the size of the proof π is poly(λ) and $SNARK.Verify$ runs in time poly($\lambda + |x|$).

Firstly, if x_1, $state_1$ and $state_2$ are the empty strings, we obtain a more standard SNARK definition. Secondly, $\mathcal{R}$ is not a component of the vector aux_{SNARK} so even if $SNARK.Setup$ has aux_{SNARK} as parameter, it is universal, i.e., it can be used to derive proving and verification keys for circuits of any size up to a polynomial in the security parameter λ, independently of any specific NP relation. Thirdly, for the SNARKs we design, the size of the key used by the honest verifier is much smaller than the size of the honest prover's key. To capture this special case we made the separation clear between the two keys; however, a potential adversarial prover has access to the complete srs key. Moreover, our SNARKs are secure in the *AGM* model [46], i.e., security is w.r.t. *AGM* adversaries only and by $\mathcal{E}^{\mathcal{A}}$ we denote the extractor $\mathcal{E}$ that has access to all of $\mathcal{A}$'s messages during the protocol with the honest verifier including the coefficients of the linear combinations of group elements used by the adversary at any protocol step for outputting new group elements at the next step. Finally, the auxiliary input (i.e., $state_2$) is required to be drawn from a "benign distribution" or else extraction may be impossible [47, 48].

We did not include the notion of zero-knowledge since it is not required.

C RANGED POLYNOMIAL PROTOCOLS FOR NP RELATIONS

In the following, we keep the convention that all algorithms receive an implicit security parameter λ. The definition below is a natural extension of the notions of polynomial protocols and polynomial protocols for relations from Section 4 of PLONK [43] to polynomial protocols over ranges for conditional NP relations with additional refinements required by our specific use case; these refinements are incorporated into steps 4, 5 and 6 as follows:

Definition C.1. (Polynomial Protocols over Ranges for Conditional NP Relations) Assume three parties, a prover $\mathcal{P}_{poly}$, a verifier $\mathcal{V}_{poly}$ and a trusted party $\mathcal{I}$. Let $\mathcal{R}^c$ be a conditional NP relation (with c being a predicate) and let x be a public input both of which have been given to $\mathcal{P}_{poly}$ and $\mathcal{V}_{poly}$ by an *InitGen* efficient algorithm. For positive integers d, D, t, l, u, e and for set $S \subset \mathbb{F}$, an S-ranged (d, D, t, l, u, e)-polynomial protocol $\mathscr{P}_{\mathcal{R}^c}$ for relation $\mathcal{R}^c$ is a multi-round protocol between $\mathcal{P}_{poly}$, $\mathcal{V}_{poly}$ and $\mathcal{I}$ such that:

(1) The protocol $\mathscr{P}_{\mathcal{R}^c}$ definition includes a set of pre-processed polynomials $g_1(X), \ldots, g_l(X) \in \mathbb{F}_{<d}[X]$.
(2) The messages of $\mathcal{P}_{poly}$ are sent to $\mathcal{I}$ and are of the form $f(X)$ for $f(X) \in \mathbb{F}_{<d}[X]$.
If $\mathcal{P}_{poly}$ sends a message not of this form, the protocol is aborted.
(3) The messages from $\mathcal{V}_{poly}$ to $\mathcal{P}_{poly}$ are random coins.
(4) $\mathcal{V}_{poly}$ may perform arithmetic computations using input x and the random coins used in the communication with $\mathcal{P}_{poly}$. Let $(res_1, \ldots, res_u)$ be the results of those computations which $\mathcal{V}_{poly}$ sends to $\mathcal{I}$.
(5) Using vectors which are part of input x and/or other ad-hoc vectors which $\mathcal{V}_{poly}$ deems useful, $\mathcal{V}_{poly}$ may compute interpolation polynomials $s_1(X), \ldots, s_e(X)$ over domain S such that $s_1(X), \ldots, s_e(X) \in \mathbb{F}_{<d}[X]$. $\mathcal{V}_{poly}$ sends $s_1(X), \ldots, s_e(X)$ to $\mathcal{I}$.
(6) At the end of the protocol, suppose $f_1(X), \ldots, f_t(X)$ are the polynomials that were sent from $\mathcal{P}_{poly}$ to $\mathcal{I}$. $\mathcal{V}_{poly}$ may ask $\mathcal{I}$ if certain polynomial identities hold between

$$\{f_1(X), \ldots, f_t(X), g_1(X), \ldots, g_l(X), s_1(X), \ldots, s_e(X)\}$$

over set S (i.e., if by evaluating all the polynomials that define the identity at each of the field elements from S we obtain a true statement). Each such identity is of the form

$$F(X) := G(X, h_1(v_1(X)), \ldots, h_M(v_M(X))) \equiv 0,$$

for some $h_i(X) \in \{f_1(X), \ldots, f_t(X), g_1(X), \ldots, g_l(X), s_1(X), \ldots, s_e(X)\}$,
$G(X, X_1, \ldots, X_M) \in \mathbb{F}[X, X_1, \ldots, X_M], v_1(X), \ldots, v_M(X) \in \mathbb{F}_{<d}[X]$ such that $F(X) \in \mathbb{F}_{<D}[X]$ for every choice of $f_1(X), \ldots, f_t(X)$ made by P_{poly} when following the protocol correctly. Note that some of the coefficients in the identities above may be from the set $\{res_1, \ldots, res_u\}$.
(7) After receiving the answers from I regarding the polynomial identities, $\mathcal{V}_{poly}$ outputs acc if all identities hold over set S, and outputs rej otherwise.

Additionally, the following properties hold:

Perfect Completeness: If $\mathcal{P}_{poly}$ follows the protocol correctly and uses a witness ω with $(x, \omega) \in \mathcal{R}^c$, $\mathcal{V}_{poly}$ accepts with probability one.
Knowledge Soundness: There exists an efficient algorithm E, that given access to the messages of $\mathcal{P}_{poly}$ to $\mathcal{I}$ it outputs ω such that, for any strategy of $\mathcal{P}_{poly}$, the probability of $\mathcal{V}_{poly}$ outputting acc at the end of the protocol and, simultaneously, $(x, \omega) \in \mathcal{R}^c$ is overwhelming in λ.

Our definition for polynomial protocols over ranges does not include a zero-knowledge property as it is not required in our current work.

Given the definition for polynomial protocols over ranges for conditional relations as detailed above, we are now ready to state the following result. The proof follows with only minor changes from that of Lemmas 4.5. and 4.7. from [43].

Lemma C.2. *(Compilation of Ranged Polynomial Protocols for Conditional NP Relations into Hybrid Model SNARKs using PLONK) Let $\mathscr{P}_{\mathcal{R}^c}$ be a public coin S-ranged (d, D, t, l, u, e)-polynomial protocol for relation $\mathcal{R}^c$ where only one identity is checked by $\mathcal{V}_{poly}$ and predicate c from the definition of $\mathcal{R}^c$ needs to be fulfilled only by a part x_1 of the public input of the relation $\mathcal{R}^c$. Then one can construct a hybrid model SNARK protocol $\mathscr{P}^*_{\mathcal{R}^c}$ for relation $\mathscr{P}_{\mathcal{R}^c}$ with SNARK.PartInput as defined below and with $\mathscr{P}^*_{\mathcal{R}^c}$ secure in the AGM under the $2d$-DLOG assumption[2] such that:*

(1) The prover $\mathbf{P}$ in $\mathscr{P}^*_{\mathcal{R}^c}$ requires $\mathsf{e}(\mathscr{P}_{\mathcal{R}^c})$ $\mathbb{G}_{1,out}$-exponentiations where $\mathsf{e}(\mathscr{P}_{\mathcal{R}^c})$ is define analogously as in PLONK (see preamble of Section 4.2.), however it additionally takes into account polynomials $s_1(X), \ldots, s_e(X)$.
(2) The total prover communication consists of $t + t^*(\mathscr{P}_{\mathcal{R}^c}) + 1$ $\mathbb{G}_{1,out}$-elements and M $\mathbb{F}$-elements, where $t^*(\mathscr{P}_{\mathcal{R}^c})$ is defined identically as in PLONK (see preamble of Section 4.2.).
(3) The verifier $\mathbf{V}$ in $\mathscr{P}^*_{\mathcal{R}^c}$ requires $t + t^*(\mathscr{P}_{\mathcal{R}^c}) + 1$ $\mathbb{G}_{1,out}$-exponentiations, two pairings and one evaluation of the polynomial G, and, additionally, the verifier in $\mathscr{P}^*_{\mathcal{R}^c}$ computes e polynomial commitments to polynomials in the set $\{s_1(X), \ldots, s_e(X)\}$.

[2]Definition 2.1. in PLONK [43] formally describes the $2d$-DLOG assumption.

(4) For x_1 part of $state_1$, the algorithm for computing partial inputs is defined as

$$SNARK.PartInput(srs, state_1, \mathcal{R}^c)$$
$$If\ c(x_1) = 0$$
$$\quad Return$$
$$Else$$
$$\quad Return(state_1, x_1)$$

D POSTPONED PROOFS FOR PACKED ACCOUNTABLE RANGED POLYNOMIAL PROTOCOL

We give below the missing proofs from Section **??**:

Claim D.1. *If the polynomial identities $id_6(X) = 0, id_7(X) = 0$ hold over range H, then, e.w.n.p. , we have $c_{a,i} = 2^{i \mod \text{block}} \cdot r^{i \div \text{block}}$, $\forall i < n$ and $\text{sum} = \sum_{i=0}^{n-1} b_i \cdot c_{a,i}$, where $b_i = b(\omega^i), \forall i < n$. If, additionally, identity $id_5(X) = 0$ holds over H, r has been randomly chosen in $\mathbb{F}$, $\text{sum} = \sum_{j=0}^{\frac{n}{\text{block}}-1} b'_j r^j$ (as computed by $\mathcal{V}_{poly}$) and $bit_i \in \mathbb{B}, \forall i < n$ and $b'_j = \sum_{k=0}^{\text{block}-1} 2^k \cdot bit_{\text{block}\cdot j+k}, \forall 0 \le j \le \frac{n}{\text{block}} - 1$ (due to the input $(b'_0, \ldots, b'_{\frac{n}{\text{block}}-1})$ being interpreted by the verifier $\mathcal{V}_{poly}$ as in $\mathbb{F}^{\frac{n}{\text{block}}}_{|\text{block}|}$), then e.w.n.p. , $b_i = bit_i, \forall i < n$.*

Proof. To prove the first part of the claim, assume by contradiction that $c_{a,0} = k \neq 1$. Then, by induction, since $id_6(X) = 0$ on H,

$$c_{a,i} = k \cdot 2^{i \mod \text{block}} \cdot r^{i \div \text{block}}, \forall 0 < i < n.$$

Additionally, the property

$$c_{a,0} = c_{a,n-1} \cdot (2 + (\frac{r}{2^{\text{block}-1}} - 2) \cdot 1) + (1 - r^{\frac{n}{\text{block}}}) \tag{1}$$

holds (again, from $id_6(X) = 0$ on H). However, substituting $c_{a,0} = k$ and $c_{a,n-1} = k \cdot 2^{\text{block}-1} \cdot r^{\frac{n}{\text{block}}-1}$ in (1), we obtain $k = k \cdot 2^{\text{block}-1} \cdot r^{\frac{n}{\text{block}}-1} \cdot \frac{r}{2^{\text{block}-1}} + 1 - r^{\frac{n}{\text{block}}}$ which is equivalent to $k(1 - r^{\frac{n}{\text{block}}}) = 1 - r^{\frac{n}{\text{block}}}$, and, due to Schwartz-Zippel Lemma and the fact that degree n is negligibly smaller compared to the size of $\mathbb{F}$, this implies e.w.n.p. $k = 1$ thus contradiction, so the values $c_{a,i}$ have indeed the claimed form.
Next, by expanding $id_7(X) = 0$ over H, the following holds

$$\begin{aligned} acc_{a,1} &= acc_{a,0} + b_0 \cdot c_{a,0} \\ acc_{a,2} &= acc_{a,1} + b_1 \cdot c_{a,1} \\ &\ldots \\ acc_{a,n-1} &= acc_{a,n-2} + b_{n-2} \cdot c_{a,n-2} \\ acc_{a,0} &= acc_{a,n-1} + b_{n-1} \cdot c_{a,n-1} - \text{sum}. \end{aligned}$$

By summing together the LHS and, respectively, the RHS of the equalities above and reducing the equal terms, we obtain $\text{sum} = \sum_{i=0}^{n-1} b_i \cdot c_{a,i}$.

For the second part of the claim, since $id_5(X) = 0$ holds over H then $b_i = b(\omega^i) \in \mathbb{B}, \forall i \le n-1$. Finally, from verifier's computation and from the first part of the claim we have

$$\sum_{j=0}^{\frac{n}{\text{block}}-1} b'_j r^j = \text{sum} = \sum_{i=0}^{n-1} b_i \cdot c_{a,i} = \sum_{i=0}^{n-1} b_i \cdot 2^{i \mod \text{block}} \cdot r^{i \div \text{block}} =$$
$$= \sum_{j=0}^{\frac{n}{\text{block}}-1} (\sum_{k=0}^{\text{block}-1} 2^k \cdot b_{\text{block}\cdot j+k}) \cdot r^j = \sum_{j=0}^{\frac{n}{\text{block}}-1} b''_j r^j, \tag{2}$$

where $\forall j, b''_j$ are field elements equal to the binary representation that uses contiguous blocks of block components from the bitmask $(b_0, \ldots, b_{n-1})$. Since both the LHS and the RHS of relation (2) represent two ways of computing sum as an inner product of a vector of field elements (on one hand, $(b'_0, \ldots, b'_{\frac{n}{\text{block}}-1})$, on the other hand, $(b''_0, \ldots, b''_{\frac{n}{\text{block}}-1})$) with the vector $(1, r, \ldots, r^{\frac{n}{\text{block}}-1})$, where r has been chosen at random, by the small exponents test [49], we obtain that e.w.n.p. $b''_j = b'_j$,
$\forall\ 0 \le j \le \frac{n}{\text{block}} - 1$. Finally, if we equate the bit representation in $\mathbb{F}$ (i.e., using field elements from $\mathbb{B}$) of field elements b''_j and $b'_j, \forall 0 \le j \le \frac{n}{\text{block}} - 1$ and remember that, by verifier's check or by construction, respectively, each such field element has no more that block binary bits, we can conclude that e.w.n.p. $b_i = bit_i, \forall i < n$. □

Lemma D.2. *$\mathscr{P}_{\text{pa}}$ as described in Section **??** is an H-ranged polynomial protocol for conditional NP relation $\mathcal{R}^{incl}_{\text{pa}}$.*

Proof. If $(\mathbf{b}', \mathbf{pk}, apk, \mathbf{bit}) \in \mathcal{R}_{\mathsf{pa}}^{incl}$, meaning that $\mathbf{pk} \in \mathbb{G}_{1,inn}^{n-1}$ and $\mathbf{bit} \in \mathbb{B}^n$ and $apk = \sum_{i=0}^{n-2}[bit_i] \cdot pk_i$ and $b'_j = \sum_{i=0}^{\mathsf{block}-1} 2^i \cdot bit_{\mathsf{block}\cdot j+i}, \forall j < \frac{n}{\mathsf{block}}$ hold then it is easy to see that the honest prover $\mathcal{P}_{poly}$ in $\mathscr{P}_{\mathsf{pa}}$ will convince the honest verifier $\mathcal{V}_{poly}$ in $\mathscr{P}_{\mathsf{pa}}$ to accept with probability 1 so perfect completeness holds. Regarding knowledge-soundness, if the verifier $\mathcal{V}_{poly}$ in $\mathscr{P}_{\mathsf{pa}}$ accepts, then the extractor $\mathcal{E}$ sets $(bit_0, \ldots, bit_{n-1})$ as the vector of evaluations over H of polynomial $b(X)$ sent by $\mathcal{P}_{poly}$ to $\mathcal{I}$. Next, we prove that if $(pk_0, \ldots, pk_{n-2}) \in \mathbb{G}_{1,inn}^{n-1}$ and the verifier in $\mathscr{P}_{\mathsf{pa}}$ accepts, then

$$((b'_0, \ldots, b'_{\frac{n}{\mathsf{block}}-1}), (pk_0, \ldots, pk_{n-2}), apk, (bit_0, \ldots, bit_{n-1})) \in \mathcal{R}_{\mathsf{pa}}^{incl},$$

which is equivalent to proving that $apk = \sum_{i=0}^{n-2}[bit_i] \cdot pk_i$ and $\mathbf{bit} \in \mathbb{B}^n$ and

$$b'_j = \sum_{i=0}^{\mathsf{block}-1} 2^i \cdot bit_{\mathsf{block}\cdot j+i}, \forall j < \frac{n}{\mathsf{block}}.$$

According to Claim D.1 and corollary 5.2 this indeed holds e.w.n.p. □

E CHOOSING h WHEN $E_{inn} = \mathbb{G}_{1,inn}$

For the polynomial protocols and custom SNARKs we have designed in Section 5, we have chosen $h \in E_{inn} \setminus \mathbb{G}_{1,inn}$. However, we have not covered so far the case when $E_{inn} = \mathbb{G}_{1,inn}$ and how to choose h in such a situation. Our current section will give a guide for that. In fact, if $E_{inn} = \mathbb{G}_{1,inn}$, we provide a method of choosing h that will be suitable not only for our custom SNARKs from Section 5, but also for any other SNARK that proves the correctness of an aggregated public key (i.e., apk for an aggregatable signature scheme), among other modelled constraints.

Let $\mathcal{H}$ be a hash function, $\mathcal{H} : \{0,1\} \to \mathbb{F}$ such that $\mathcal{H}$ is used for the Fiat-Shamir transformation of a succinct argument of knowledge (including a sub-proof of correctness of apk) into its non-interactive version. Let x be the public input corresponding to the above succinct argument of knowledge. Note that in case of the hybrid model SNARKs defined in this work (see Definition B.1), the public input includes the partial input. For a concrete example of a partial input, see one of our custom SNARKs fully rolled out in Section L. Then, the prover and the verifier compute h, for example as

$$h = \mathcal{H}(h, \text{"starting input point for public keys addition"}).$$

Intuitively, in the random oracle model (which is already an assumption we need for a secure Fiat-Shamir transformation that preserves knowledge-soundness), h is thus an elliptic curve point on E_{inn}, uniformly distributed on $E_{inn} = \mathbb{G}_{1,inn}$. Hence, for a large enough elliptic curve group (i.e., an elliptic curve for which the number of elliptic curve points is $O(2^\lambda)$ for security parameter λ), the probability of h plus any elliptic curve point being equal to a fixed elliptic curve is negligible in λ. This, in turn, ensures that condition 3) from Observation **??** is only met with negligible probability. Hence, knowledge soundness for our polynomial protocols in Sections 5.1,**??** still holds with overwhelming probability, so all follow-up results in Section 5 still hold.

F COMPILER FOR HYBRID MODEL SNARKS WITH MIXED INPUTS

F.1 Technical Challenges and Contributions Regarding our Custom SNARKs

In order to define and implement our committee key scheme accountable light client systems and in order to design the custom SNARKs that support our efficiency results, we had to tackle some technical challenges and make additional contributions as summarised below.

Extending PLONK Compiler to Mixed Commitment and Vectors NP Relations. Firstly, our custom SNARKs takes inspiration from PLONK [43] in terms design of the proof system used, and of the circuits and gates. However, our SNARKs also have differences compared to PLONK. PLONK applies to NP relations that use vectors of field elements for public inputs and witnesses. However we need SNARKs whose defining NP relations also have polynomial commitments (in our case, the committee key C) as part of their public inputs. Hence, the original PLONK compiler does not suffice; we extend it with a second step in which we show that under certain conditions which our protocol fulfils, the SNARKs obtained using the original PLONK compiler are also SNARKs for a mixed type of NP relation containing both vectors and polynomial commitments. The full details and proofs can be found in Section F and we believe this compiler extension to be of independent interest.

Conditional NP Relations for Efficiency. Secondly, we also require NP relations that have a well-defined subpredicate which is verified outside the SNARKs. In a blockchain instantiation, any current validator set has to come to a consensus, among other things, on the next validator set, represented by a set of public keys. The validator set computes and signs a pair of polynomial commitments to the next set of validators' public keys. Before including a public key in the set, the validators perform several checks on the proposed public key, such as being in a particular subgroup of the elliptic curve. This check is not performed by the SNARKs' constraint system, but is required for the correctness of the statement the SNARKs prove. This design decision makes our SNARKs more efficient, but it also means we have to extend the usual definition of NP relations to conditional NP relations, where in fact, one of the subpredicates

that define the conditional relation is checked outside the SNARKs or ensured due to a well-defined assumption. We introduce the general notion of conditional NP relation in Section 4.4 and describe our concrete conditional NP relations in Section 5.

Hybrid Model SNARKs. In line with the two above technical challenges and the solutions we came up with, we extend the existing definitions related to SNARKs [31, 43] by introducing an algorithm which we call *PartInput*. For our use case, this allows us to separate the public input for the NP relations that define our custom SNARKs in two: a part that is computed by the current set of validators on the blockchain in question and the rest of the public input plus the corresponding SNARK proof are computed by a (possibly malicious) prover interacting with the light client verifier. Our newly introduced notion of hybrid model SNARK (see Section B) generalises this public input separation concept and its definition is used to prove the security of our custom SNARKs in Section F.

F.2 SNARK compiler

We present a two-steps PLONK-based compilation technique from ranged polynomial protocols for conditional NP relations (formal definition in Section C) to hybrid model SNARKs (Definition B.1) such that the conditional NP relations that define the SNARKs we compile in the second step contain both polynomial commitments and vectors of field elements as public inputs. By using just the first step of our compiler which is equivalent to the original PLONK compiler [43], one would not be able to obtain SNARKs with mixed public inputs consisting of both vectors of field elements and also poly commitments. In turn, this type of NP relations with mixed inputs is crucial for designing accountable light clients via the use of committee key schemes (see Section 5.3).

F.3 Our Compiler: Step 1
(PLONK Compiler - from Polynomial Protocols to SNARKs)

We summarise and exemplify below the PLONK-based compilation technique [43] from ranged polynomial protocols for conditional NP relations (formal definition in Section C) to SNARKs for pure vector-based NP relations. This is also the first of our two-steps compiler. Concretely, our first step applies the PLONK compiler [43] (Lemma 4.7): we compile the information theoretical ranged polynomial protocols $\mathscr{P}_{\mathsf{ba}}$ and $\mathscr{P}_{\mathsf{pa}}$ for relations $\mathcal{R}_{\mathsf{ba}}^{incl}$ and $\mathcal{R}_{\mathsf{pa}}^{incl}$, respectively (see Sections 5.1,**??**) into SNARKs $\mathscr{P}_{\mathsf{ba}}^{*}$, and $\mathscr{P}_{\mathsf{pa}}^{*}$, respectively. We can define this compilation step for any ranged polynomial protocols for relations (as per definition C in Section C.1). In order to do that we need:

- The batched version of KZG polynomial commitments [44] described in Section 3 of PLONK [43].[3]
- A general compilation technique: such a technique has been already defined in Lemma 4.7 of PLONK; combined with Lemma 4.5 from PLONK this technique can be applied with minor adaptations (this includes the corresponding technical measures) to the notion of ranged polynomial protocols.
- So far, both the ranged polynomial protocols for relations and the protocols resulted after the first compilation step have been explicitly defined as interactive protocols. In order to obtain the non-interactive version of the latter (essentially the N in SNARK) one has to apply the Fiat-Shamir transform [52], [53], [54].

Let $\mathcal{R}$ be a (conditional) NP relation, let $\mathscr{P}_{\mathcal{R}}$ be a ranged polynomial protocol for relation $\mathcal{R}$ and let $\mathscr{P}_{\mathcal{R}}^{*}$ be the SNARK compiled from $\mathscr{P}_{\mathcal{R}}$ using the PLONK compiler. The compilation technique requires the SNARK prover of $\mathscr{P}_{\mathcal{R}}^{*}$ to compute polynomial commitments to all polynomials that the prover $\mathcal{P}_{poly}$ in $\mathscr{P}_{\mathcal{R}}$ sent to the ideal party $\mathcal{I}$. Analogously, it requires the SNARK verifier of $\mathscr{P}_{\mathcal{R}}^{*}$ to compute polynomial commitments to all pre-processed polynomials[4] as well polynomial commitments to polynomials the verifier $\mathcal{V}_{poly}$ in $\mathscr{P}_{\mathcal{R}}$ sent to the ideal party $\mathcal{I}$. Then, the SNARK prover sends the SNARK verifier openings to all the polynomial commitments computed by him as well as the polynomial commitments computed by the SNARK verifier. The SNARK prover additionally sends the corresponding batched proofs for polynomial commitment openings. In turn, the SNARK verifier accepts or rejects based on the result of the verification of the batched polynomial commitment scheme.

A more efficient compilation technique exists which reduces the number of polynomial commitments and alleged polynomial commitments openings (i.e., both group elements and field elements) sent by the SNARK prover to the SNARK verifier; this, in turn, reduces the size of the SNARK proof. This technique is called linearisation and is described, at a high level, after Lemma 4.7 in PLONK. The existing description however covers only the SNARK prover side and it does not detail the SNARK verifier side so in the following we cover that.

By functionality, the vectors that are handled by the the verifier $\mathcal{V}_{poly}$ are of two types: pre-processed vectors and public input vectors. These two types of vectors are used by $\mathcal{V}_{poly}$ to obtain, via interpolation over the range on which the respective range polynomial protocol is defined, pre-processed polynomials (as used in the definition C in Section C.1, e.g., polynomial $aux(X)$ used in Section 5.1) and public-inputs-derived polynomials (e.g., polynomials $pkx(X)$ and $pky(X)$ used in Sections 5.1,**??**) and polynomial $b(X)$ used in

[3] In fact, one can replace the use of KZG polynomial commitments with any binding polynomial commitment that has knowledge-soundness, including non-homomorphic polynomial commitments, such as FRI-based polynomial commitments (e.g., RedShift [50]). If the optimisation gained from PLONK linearisation technique is a goal, then, with minimal changes one can use any homomorphic polynomial commitment, e.g., the discrete logarithm based polynomial commitment from Halo [51].
[4] This is a one-time computation that is reused by the SNARK verifier for all SNARK proofs over the same circuit.

Section 5.1). The efficient linearisation technique allows the SNARK verifier to reduce the number of polynomial commitments it has to compute compared to the general PLONK compiler in the following way. Instead of having to compute polynomial commitments to all polynomials $\mathcal{V}_{poly}$ sends to $\mathcal{I}$ (including any corresponding pre-processed polynomials), the SNARK verifier computes polynomial evaluations at one or multiple random points (as per the linearisation step specific requirements) for all the polynomials that are either easy to evaluate (e.g., polynomial $aux(X)$ used in Section **??**) or all the polynomials that are obtained from vectors that do not take up a large amount of memory (e.g., polynomial $b(X)$ used in Section 5.1). For the rest of the polynomials (e.g., $pkx(X)$ and $pky(X)$), the SNARK verifier computes polynomial commitments as before.

We note we can apply all the techniques mentioned above, including the combined prover-and-verifier-side linearisation to compile our ranged polynomial protocols $\mathscr{P}_{\mathsf{ba}}$ and $\mathscr{P}_{\mathsf{pa}}$ into the corresponding SNARKs $\mathscr{P}^*_{\mathsf{ba}}$ and $\mathscr{P}^*_{\mathsf{pa}}$, respectively. To conclude this step, we formally state in Section C.1, Lemma C.2 under which condition and how efficiently one can compile ranged polynomial protocols for conditional NP relations (where the public inputs are interpreted as vector of field elements) into hybrid model SNARKs by using only the original PLONK compiler.

F.4 Our Compiler: Step 2
(Mixed Vector and Commitments based NP Relations and Associated SNARKs)

The type of NP relations we have worked with so far as well as the more general PLONK NP relation ([43], Section 8.2) have vector of field elements as public inputs. Next we show that SNARKs compiled using Step 1 can become, under certain assumption, SNARKs for a new type of NP relation that specifically contains polynomial commitments as part of the input. Interpreting our already compiled SNARKs as SNARKs for this new type of NP relation is essential for designing accountable light client systems via committee key schemes (see instantiation 5.4 in Section 5.3).
Let conditional NP relation $\mathcal{R}^c_{vec}$ be:

$$\begin{aligned}\mathcal{R}^c_{vec} = \{&(\mathbf{input_1} \in \mathcal{D}_1, \mathbf{input_2} \in \mathcal{D}_2; \mathbf{witness_1}) : \\ &p_1(\mathbf{input_1}, \mathbf{input_2}, \mathbf{witness_1}) = 1 \mid c(\mathbf{input_1}) = 1 \wedge \\ &\wedge\ p_2(\mathbf{input_1}, \mathbf{input_2}, \mathbf{witness_1}) = 1\},\end{aligned}$$

where $\mathbf{input_1}$, $\mathbf{input_2}$ are two sets of public input vectors belonging domains $\mathcal{D}_1$, $\mathcal{D}_2$. $\mathbf{witness_1}$ is a set of witness vectors and c, p_1, p_2 are predicates. Let $\mathscr{P}_{vec}$ be a ranged polynomial protocol for relation $\mathcal{R}^c_{vec}$. Note that since c applies only to a part of the public input for relation $\mathcal{R}^c_{vec}$ (i.e., $\mathbf{input_1}$), we can apply Lemma C.2 of Section C and Step 1 of our compiler to polynomial protocol $\mathscr{P}_{vec}$.

We make the following hybrid model assumptions:

- (HMA.1.) $\mathcal{V}_{poly}$ in $\mathscr{P}_{vec}$ computes $Q_{1,\mathbf{input_1}}(X), \ldots, Q_{m,\mathbf{input_1}}(X)$ which depend deterministically on $\mathbf{input_1}$ and sends them to $\mathcal{I}$.
- (HMA.2.) $\mathcal{V}_{poly}$ in $\mathscr{P}_{vec}$ does not use $\mathbf{input_1}$ in any further computation of any other polynomials or values its sends to $\mathcal{I}$.
- (HMA.3.) By evaluating $Q_{1,\mathbf{input_1}}(X), \ldots, Q_{m,\mathbf{input_1}}(X)$ over the range on which $\mathscr{P}_{vec}$ is defined one obtains (using some efficiently computable and deterministic transformations) the set of vectors $\mathbf{input_1}$.

We denote by $\mathscr{P}^*_{vec}$ the hybrid model SNARK obtained after compiling $\mathscr{P}_{vec}$ using compilation Step 1. Due to (HMA.1.) and according to Step 1, the SNARK verifier in $\mathscr{P}^*_{vec}$ computes

$$Com_1 = Com(Q_{1,\mathbf{input_1}}), \ldots, Com_m = Com(Q_{m,\mathbf{input_1}})$$

which are KZG poly commitments to $Q_{1,\mathbf{input_1}}(X), \ldots, Q_{m,\mathbf{input_1}}(X)$. We denote vector $(Com_1, \ldots, Com_m)$ by $\mathbf{Com}(\mathbf{input_1})$ and we denote by C the set of all *KZG* poly commitments or vectors of such poly commitments. We also define the relation:

$$\begin{aligned}\mathcal{R}^c_{vec,com} = \{&\mathbf{C} \in C, \mathbf{input_2} \in \mathcal{D}_2; \mathbf{witness_1}, \mathbf{witness_2}) : \\ &p_1(\mathbf{witness_2}, \mathbf{input_2}, \mathbf{witness_1}) = 1 \mid c(\mathbf{witness_2}) = 1 \wedge \\ &\wedge\ p_2(\mathbf{witness_2}, \mathbf{input_2}, \mathbf{witness_1}) = 1 \wedge \\ &\wedge\ \mathbf{C} = \mathbf{Com}(\mathbf{witness_2})\}\end{aligned}$$

Finally, for $\mathbf{input_1}$ part of $state_1$, we define $SNARK.PartInput$:

$SNARK.PartInput(srs, state_1, \mathcal{R}^c_{vec,com})$
If $c(\mathbf{input_1}) = 0$ *then Return*
Else
  Compute via interpolation on $\mathscr{P}_{vec}$ *range* $Q_{1,\mathbf{input}_1}(X), \ldots, Q_{m,\mathbf{input}_1}(X)$.
  $\mathbf{C} = (Com(Q_{1,\mathbf{input}_1}(X)), \ldots, Com(Q_{m,\mathbf{input}_1}(X)))$
  $state_2 = state_1 \cup \{\mathbf{C}\}$ *then Return*$(state_2, \mathbf{C})$

With the above notation, **our compiler's Step 2 is:**
The alleged hybrid model SNARK $\mathscr{P}^h_{vec}$ for relation $\mathcal{R}^c_{vec,com}$ is:

- $SNARK.Setup$ and $SNARK.KeyGen$ are as for relation $\mathcal{R}^c_{vec}$.
- $SNARK.PartInput$ for relation $\mathcal{R}^c_{vec}$ (see Lemma C.2 in Section C) is replaced with $SNARK.PartInput$ for relation $\mathcal{R}^c_{vec,com}$.
- $SNARK.Prover$ for relation $\mathcal{R}^c_{vec,com}$ is identical with $SNARK.Prover$ for relation $\mathcal{R}^c_{vec}$ (as compiled using Step 1) with the appropriate re-interpretation of the public inputs and witness.
- $SNARK.Verifier$ for relation $\mathcal{R}^c_{vec,com}$ is identical with $SNARK.Verifier$ for relation $\mathcal{R}^c_{vec}$ (as compiled using Step 1) with the appropriate re-interpretation of the public inputs and such that $SNARK.Verifier$ for $\mathcal{R}^c_{vec,com}$ does not compute the polynomial commitments to the polynomials defined by assumption (HMA.1.).

Lemma F.1. *Let* $\mathscr{P}_{vec}$ *be a ranged polynomial protocol for relation* $\mathcal{R}^c_{vec}$ *defined above and let* $\mathscr{P}^*_{vec}$ *be the hybrid model SNARK for relation* $\mathcal{R}^c_{vec}$ *secure in the AGM obtained by compiling* $\mathscr{P}_{vec}$ *using our compiler's Step 1. If the hybrid model assumptions (HMA.1.) - (HMA.3.) hold w.r.t. protocol* $\mathscr{P}_{vec}$ *and* $State_{\mathcal{R}_{vec,com}} \neq \emptyset$ *then* $\mathscr{P}^h_{vec}$ *as compiled using our compiler's Step 2 is a hybrid model SNARK for relation* $\mathcal{R}^c_{vec,com}$ *secure also in the AGM.*

Proof. Let $\mathcal{E}_{KZG}$ and $\mathcal{E}$ be the extractors from the knowledge-soundness definitions for the *KZG* batch polynomial commitment scheme (as in definition 3.1, Section 3 in [43]) and the hybrid model SNARK $\mathscr{P}^*_{\mathcal{R}}$ for relation $\mathcal{R}^c_{vec}$ (as per definition B.1), respectively. Let $\mathcal{A}$ be an adversary against knowledge soundness in the hybrid model w.r.t. $\mathscr{P}^h_{vec}$ and relation $\mathcal{R}^c_{vec,com}$ and let $aux_{SNARK} \in \mathcal{D}$ and let $state_1 \in State_{\mathcal{R}_{vec,com}}$; let $(\mathbf{C}, state_2) = SNARK.PartInput(srs, state_1, \mathcal{R}^c_{vec,com})$. By the definition of $SNARK.PartInput$ for $\mathscr{P}^h_{vec}$, there exists $\mathbf{input_1}$ such that $\mathbf{C} = \mathbf{Com}(\mathbf{input_1})$ and $c(\mathbf{input_1}) = 1$. We denote by $(\mathbf{input_2}, \pi)$ the output of $\mathcal{A}(srs, state_2, \mathcal{R}^c_{vec,com})$ and let $\mathcal{A}_1$ be the part of $\mathcal{A}$ that sends openings and batched proofs for the polynomial commitments in $\mathbf{C}$.

On the one hand, if $SNARK.Verifier(srs_{vk}, (\mathbf{C}, \mathbf{input_2}), \pi, \mathcal{R}^c_{vec,com})$ in $\mathscr{P}^h_{vec}$ accepts, then the *KZG* verifier corresponding to $\mathcal{A}_1$ also accepts. When such an event takes place, then, e.w.n.p. $\mathcal{E}_{KZG}$ extracts polynomials $Q'_1(X), \ldots, Q'_m(X)$ that represent witnesses for the vector $\mathbf{C}$ of commitments and the alleged openings of $\mathcal{A}_1$. Because the *KZG* polynomial commitment scheme is binding and by the definition of $SNARK.PartInput$ for $\mathscr{P}^h_{vec}$, we obtain that $Q'_1(X) = Q_1(X), \ldots, Q'_m(X) = Q_m(X)$. Since per (HMA.3.), the set $\{Q_1(X), \ldots, Q_m(X)\}$ evaluates to $\mathbf{input_1}$ over the range over which $\mathscr{P}_{vec}$ was defined, e.w.n.p. the witness polynomials extracted by $\mathcal{E}_{KZG}$ evaluate to $\mathbf{input_1}$.

On the other hand, if $SNARK.Verifier(srs_{vk}, (\mathbf{C}, \mathbf{input_2}), \pi, \mathcal{R}^c_{vec,com})$ in $\mathscr{P}^h_{vec}$ accepts, then
$SNARK.Verifier(srs_{vk}, (\mathbf{input_1}, \mathbf{input_2}), \pi, \mathcal{R}^c_{vec})$ in $\mathscr{P}^*_{vec}$ also accepts. In turn, this acceptance together with the fact that $\mathscr{P}^*_{vec}$ has knowledge-soundness as per definition B.1, it implies $\mathcal{E}$ e.w.n.p. extracts $\mathbf{witness_1}$ such that $(\mathbf{input_1}, \mathbf{input_2}, \mathbf{witness_1}) \in \mathcal{R}^c_{vec}$ (#).
By the definition of $SNARK.PartInput$ for $\mathscr{P}^h_{vec}$ and the way $\mathbf{input_1}$ was defined, it holds that $c(\mathbf{input_1}) = 1$. Due to (#) and by the definition of relation $\mathcal{R}^c_{vec}$, the predicates: $p_1(\mathbf{input_1}, \mathbf{input_2}, \mathbf{witness_1}) = 1$ and $p_2(\mathbf{input_1}, \mathbf{input_2}, \mathbf{witness_1}) = 1$ hold. If we let $\mathbf{witness_2} = \mathbf{input_1}$, then $(\mathbf{C} = \mathbf{Com}(\mathbf{input_1}), \mathbf{input_2}, \mathbf{witness_1}, \mathbf{input_1}) \in \mathcal{R}^c_{vec,com}$, so using $\mathcal{E}_{KZG}$ and $\mathcal{E}$ we can build an extractor for any knowledge-soundness adversary $\mathcal{A}$ for alleged hybrid model SNARK $\mathscr{P}^h_{vec}$ for relation $\mathcal{R}^c_{vec,com}$, which concludes the proof. □

It is straightforward to apply the technique described above to our SNARKs $\mathscr{P}^h_{\text{ba}}$ and $\mathscr{P}^h_{\text{pa}}$ compiled in Step 2 and obtain relations $\mathcal{R}^{incl}_{\text{ba},com}$ and $\mathcal{R}^{incl}_{\text{pa},com}$ as described below such that they fulfil Lemma F.1.[5]

$$\mathcal{R}^{incl}_{\text{ba},com} = \{(\mathbf{C} \in C, \mathbf{bit} \in \mathbb{B}^n, apk \in \mathbb{F}^2; \mathbf{pk}) : apk = \sum_{i=0}^{n-2} [bit_i] \cdot pk_i \mid$$
$$\mid \mathbf{pk} \in \mathbb{G}^{n-1}_{I,inn} \ \wedge \ \mathbf{C} = \mathbf{Com}(\mathbf{pk})\}$$

[5]Due to our specific application to proof-of-stake blockchain context in which we make use of our custom SNARKs, the assumption/requirement that $State_{\mathcal{R}_{vec,com}} \neq \emptyset$ for $\mathcal{R}_{vec,com} \in \{\mathcal{R}^{incl}_{\text{ba},com}, \mathcal{R}^{incl}_{\text{pa},com}\}$ is fulfilled.

$$\mathcal{R}^{incl}_{\mathsf{pa},com} = \{(\mathbf{C} \in C, \mathbf{b'} \in \mathbb{F}^{\frac{n}{|\mathsf{block}|}}, apk \in \mathbb{F}^2; \mathbf{pk}, \mathbf{bit}) : apk = \sum_{i=0}^{n-2} [bit_i] \cdot pk_i \mid$$

$$\mid \mathbf{pk} \in \mathbb{G}^{n-1}_{1,inn} \ \wedge \ \mathbf{bit} \in \mathbb{B}^n \wedge b'_j = \sum_{i=0}^{\mathsf{block}-1} 2^i \cdot bit_{\mathsf{block}\cdot j+i}, \forall j < \frac{n}{\mathsf{block}} \wedge \ \mathbf{C} = \mathbf{Com}(\mathbf{pk})\}$$

For completeness, we also include the full rolled out SNARK $\mathscr{P}^h_{\mathsf{pa}}$ for relation $\mathcal{R}^{incl}_{\mathsf{pa},com}$ in Section L and we provide a comparison between PLONK universal SNARK and our custom SNARKs in Section G.

G COMPARISON BETWEEN PLONK AND OUR SNARKS

In the following, we briefly look at the differences between PLONK universal SNARK and the SNARKs designed in this work. We observe that while the NP relation that defines PLONK is more general, the relations that define our SNARKs are bespoke as we are only interested in efficiently proving public key aggregation. Because our relations are so bespoke, it turns out we do not require the full functionality that PLONK has to offer, and, in particular, our SNARKs do not require any permutation argument.

A second difference is that while PLONK's circuit is defined by a number of selector polynomials (which are a type of pre-processed polynomials) and a PLONK verifier needs to perform a one-time expensive computation of the polynomial commitments to those selector polynomials, our SNARK verifiers are able to avoid such a pre-processing phase. Indeed, in the case of $\mathscr{P}^h_{\mathsf{a}}$ (which is the only one of our three SNARKs that has a polynomial, namely $aux(X)$, that defines its circuit), our respective SNARK verifier does not need to compute a commitment to its only "selector polynomial" as, due to its structure, $aux(X)$ can be directly and efficiently evaluated by our SNARK verifier itself.

A third difference is that using our two-steps compiler, our SNARKs verifiers are able to efficiently handle input vectors of length $O(n)$, where the degree of the polynomials committed to by our SNARK provers is also $O(n)$. Our SNARKs verifiers achieve efficiency by offloading the expensive polynomial commitment computation involving the public inputs to a trusted third party.

Moreover, while PLONK does not incorporate trusted inputs, one can easily apply the Step 2 of our compiler to PLONK. In particular, one could imagine a situation where a PLONK verifier is relying on a trusted party to compute some or all of the polynomial commitments to the circuit's selector polynomials. This is equivalent to our hybrid model SNARK definition applied to PLONK. The benefit is that by delegating such a computation, the PLONK verifier becomes more efficient.

Finally, looking at our light client system instantiation in Section I.3 due to the inductive structure of the soundness proof (Theorem I.8), the efficiency of using a hybrid model SNARK has an even greater impact for the light client system verifier than that compared to verifying multiple instances of PLONK for the same circuit: while for the latter the PLONK verifier has to compute commitments to selector polynomial only once anyway, in the case of the former, the commitments to public inputs may differ at very step hence a trusted third party relives a higher computation burden from the light client verifier overall.

H POSTPONED SECURITY PROOF FOR COMMITTEE KEY SCHEME

Theorem H.1. *Given the hybrid model SNARK scheme secure for relation $\mathcal{R} \in \{\mathcal{R}^{incl}_{\mathsf{ba},com}, \mathcal{R}^{incl}_{\mathsf{pa},com}\}$ as obtained using our two-step compiler in Section F and the aggregatable signature scheme AS as per Instantiation A.2 (which fulfils Definition 4.1, with the additional specification that $aux_{AS} = v + 1$ and choosing $v = n - 1$, if we assume that an efficient adversary (against soundness of) $CKS_{\mathcal{R}}$ outputs public keys only from the source group $\mathbb{G}_{1,inn}$, then the committee key scheme $CKS_{\mathcal{R}}$ as per Instantiation 5.4 is secure with respect to Definition 4.3.*

Proof. We prove below the statement only for $\mathcal{R}^{incl}_{\mathsf{ba},com}$. The statement can be proven analogously for $\mathcal{R}^{incl}_{\mathsf{pa},com}$.

In order to prove perfect completeness for $CKS_{\mathcal{R}}$ Instantiation 5.4 using a hybrid model SNARK secure for relation $\mathcal{R}^{incl}_{\mathsf{ba},com}$, we note that if $AS.Verify(pp, apk, m, asig) = 1$ holds, then due to the instantiation for $CKS_{\mathcal{R}^{incl}_{\mathsf{ba},com}}.Verify$, we have that

$$CKS_{\mathcal{R}^{incl}_{\mathsf{ba},com}}.Verify(pp, rs_{vk}, ck, m, asig, (\pi_{SNARK}, apk), (bit_i)_{i=1}^{n-1}) = 1$$

iff, in turn,

$$SNARK.Verify(rs_{vk}, (ck, (bit_i)_{i=1}^{n-1}||0, apk), \pi_{SNARK}, \mathcal{R}^{incl}_{\mathsf{ba},com}) = 1 \qquad (1)$$

holds. Using the fact that the keys srs and (rs_{pk}, rs_{vk}) for our hybrid model SNARK were generated correctly using $SNARK.Setup(\upsilon, 3\upsilon)$ and respectively $SNARK.KeyGen(srs, \mathcal{R}^{incl}_{\text{ba},com})$, also since $(pk_i)_{i=1}^{n-1} \in \mathbb{G}^{n-1}_{1,inn}$ as honestly generated by $AS.GenKeypair$, then

$$(x = (ck, (bit_i)_{i=1}^{n-1}||0, apk), w = (pk_i)_{i=1}^{n-1}) \in \mathcal{R}^{incl}_{\text{ba},com}$$

(because $apk = \sum_{i=1}^{n-1} bit_i \cdot pk_i$ due to Instantiation A.2 and ck was honestly generated as $\mathbf{Com}((pk_i)_{i=1}^{n-1})$ as a pair of binding polynomial commitments to the x and y coordinates of the keys in w, respectively) and, finally, adding that the proof π_{SNARK} was generated correctly as

$$\pi_{SNARK} \leftarrow SNARK.Prove(rs_{pk}, (x, w), \mathcal{R}^{incl}_{\text{ba},com}),$$

then, by the perfect completeness property of the hybrid model SNARK for relation $\mathcal{R}^{incl}_{\text{ba},com}$, we can conclude (1).

The proof for the soundness property is described below. Let $\mathcal{A}$ be an efficient adversary that, whenever it outputs a vector of public keys $(pk_i)_{i=1}^{n-1}$, the respective vector belongs to the set $\mathbb{G}^{n-1}_{1,inn}$. Assuming that the following holds

$$CKS_{\mathcal{R}^{incl}_{\text{ba},com}}.Verify(pp, rs_{vk}, ck, m, asig, \pi = (\pi_{SNARK}, apk'), (bit_i)_{i=1}^{n-1}) = 1,$$

then, according to instantiation for $CKS_{\mathcal{R}^{incl}_{\text{ba},com}}$, it implies that both

$$AS.Verify(pp, apk', m, asig) = 1 \qquad (2)$$

and

$$SNARK.Verify(rs_{vk}, (ck, (bit_i)_{i=1}^{n-1}||0, apk'), \pi_{SNARK}, \mathcal{R}^{incl}_{\text{ba},com}) = 1 \qquad (3)$$

hold where apk' was parsed from π. Since ck was generated correctly as the pair of binding polynomial commitments $\mathbf{Com}((pk_i)_{i=1}^{n-1)}$ using the vector $(pk_i)_{i=1}^{n-1}$ output by the adversary $\mathcal{A}$ (which, as per adversary definition, belongs to $\mathbb{G}^{n-1}_{1,inn}$) and due to the knowledge soundness property of the SNARK scheme secure for relation $\mathcal{R}^{incl}_{\text{ba},com}$, the knowledge soundness and the computational binding property of the polynomial commitment scheme (since for our $CKS_{\mathcal{R}}$ instantiation we use the KZG commitment scheme), it implies that, with overwhelming probability

$$(x = (ck, (bit_i)_{i=1}^{n-1}, apk'), w = (pk_i)_{i=1}^{n-1}) \in \mathcal{R}^{incl}_{\text{ba},com}.$$

From this, in turn, by the definition of relation $\mathcal{R}^{incl}_{\text{ba},com}$, we obtain that $apk' = \sum_{i=1}^{n-1} bit_i \cdot pk_i$. Moreover, by the instantiation of aggregatable signature scheme AS, we have that $\sum_{i=1}^{n-1} bit_i \cdot pk_i = AS.AggKeys(pp, (pk_i)_{i:bit_i=1})$ and, as per soundness challenge definition, it holds that
$apk \leftarrow AS.AggKeys(pp, (pk_i)_{i:bit_i=1})$. Hence $apk' = apk$. Finally, due to (2), we conclude that

$$AS.Verify(pp, apk, m, asig) = 1$$

holds with overwhelming probability (q.e.d.).

□

I AN ACCOUNTABLE LIGHT CLIENT SYSTEM

In this section, we give a model for the consensus systems that our light client system can be applied to and we define security properties for light client systems, and, in particular accountable light client systems. Moreover, we present generic pseudocode for light client systems and prove that our implementation fulfils the security properties that we define for this notion.

I.1 Informal Model and Context

First, we informally describe our model, then we formalise it in I.2. There is a consensus system which we assume is a blockchain protocol. We consider consensus systems that make decisions based on signatures from a subset of validators, where the validator set may change periodically. Our model has the following entities:

Full Nodes. - a full node maintains a view of the consensus decisions and stores the current state of the blockchain. A full node obtains both by running the consensus protocol correctly starting from the genesis state of the blockchain.

Validator. - a validator is a full node which the consensus protocol decides it belongs to a validator set. Once elected, validators take part in the consensus protocol and, in turn, their signatures determine what the consensus decides upon.

Light Client Verifier. - a light client verifier is a node that does not keep the full state of the blockchain, but rather obtains (ideally short) proofs of parts of the blockchain state they are interested in; light client verifiers do this by being in communication with e.g., full nodes. In the optimistic scenario, where we have no adversary, the light client verifier can connect to a single full node and the full node should be able to convince the light client verifier of anything that the latter is interested in and the consensus system has agreed upon.

Adversary. The adversary controls a number of full nodes and validators. They are interested in convincing the light client verifier of things that may be in contradiction to what other (honest) nodes see as decided. The adversary, via the parties it controls, can try to double spend on the same blockchain or on another blockchain via a bridge. In the accountable case (which is the one we are interested in), the adversarial parties would like to ensure that if an attack is discovered, the honest validators and not the adversarial ones are to be blamed and punished. In the pessimistic scenario, a light client verifier may only be connected to the adversary. In this scenario, we also assume that all full nodes, including honest validators are only connected to the adversary.

Validator Sets. As briefly mentioned above, the consensus protocol decides which entities are validators; the validators, in turn, agree on the consensus. The consensus protocol designates the next validator set which, in turn, is represented by the set of the corresponding entities' public keys.

1.1.1 Informal Security Properties.

We next informally describe the security properties that our light client system should satisfy.

Completeness: If a full node sees that some fact was decided by the consensus, they can produce a proof that would convince a light client verifier of this fact.

Soundness: If, from some honest full nodes point of view, at least 1/3 of the validators in the validator set at any time are honest, then the light client verifier cannot be convinced of something incompatible with something the honest full node saw as decided.

For short, completeness and soundness mean, respectively, that in the optimistic scenario, a full node can always convince a light client verifier of some fact it sees as decided, and, in the pessimistic scenario, the adversary cannot convince the light client verifier of something that was not decided.

Accountability means that if a light client verifier was convinced of an incorrect statement (in relation to what has been decided on the blockchain so far), then one can detect the misbehaving validators that contributed to that. We can separate this into two properties:

Accountability Completeness: If the light client verifier is convinced via a wrong proof of something which is incompatible with something a full node sees as decided, and then the light client verifier forwards the wrong proof to the full node, that full node can detect that some validators misbehaved.

Accountability Soundness: If a full node is given a light client proof of something that is incompatible with something it sees as decided, then, when the full node detects that some validators misbehaved, indeed none of those validators are honest.

1.1.2 Consensus System Model.

Messages. For a full node to prove to a light client verifier that something has been decided, in the end it will prove that a *message* was signed by a quorum of validators from some *validator set*. Typically this message will not directly include the information the full node wants to convince the light client that it has been decided (during consensus), but the message will be a commitment to that information; hence, the full node can also include an opening of this commitment.

Our formal model will not mention blockchains, but it is useful to remember that in blockchain based consensus systems, often the message is a blockhash, which is a binding commitment to multiple types of data:

(1) the block header
(2) all previous block headers, through parent hashes in block headers
(3) the blocks themselves (whose hash is in the header)

We define the *required data* of a message to be the data that the message is a binding commitment to and which all full nodes should know. We assume that if a full node sees a message as decided, it must have the corresponding required data. The required data of messages can overlap among each other and the full node would not need to store them separately, e.g. two block hashes for blocks in the same chain may have required data that overlap for a prefix of blocks in the chain, which may be many gigabytes of data.

Consensus Decisions, Validator Sets, Epochs and Consensus Views. A message is decided if sufficient signatures corresponding to validators in the current validator set sign it. However the validator set may change.

We define an *epoch* as a period of time in which the validator set cannot change. During each epoch, the consensus determines the validator set for the next epoch.

We assume that the validator set size is bounded by some known constant v. Some threshold t of validators are required to sign a message such that it is considered decided. t may be a function of the size of the validator set of a given epoch, e.g. more than 2/3 of the validators. We assume that the message itself indicates what epoch it belongs to, and only signatures from validators chosen for that epoch count for whether a message has been decided or not.

Each full node maintains a *consensus view*, i.e., its view of the protocol. The consensus view records the view of the validator set for each epoch, the messages that have been decided and the signatures on those messages. It also includes the required data for each decided message.

A well-defined function of the consensus view defines its validity. Full nodes should maintain only a valid consensus view, and must not include in their consensus view messages that would make the respective view invalid.

Incompatible Messages. There are some pairs of decisions that a consensus protocol cannot decide together without breaking validity. If the protocol ensures that honest validators do not sign messages corresponding to both decisions, then we can make signing such pairs of messages punishable.

Unfortunately the messages themselves need not be enough to judge their incompatibility. For example we would not want two block hashes to be decided if one is for a block of height 100 and the other is for a block of height 101, and the block of height 100 was not the parent of the block of height 101. However, if incompatibility is a function of the required data of one or both messages, then, because messages are binding commitments to their required data, this is still unambiguous for a pair of messages.

I.1.3 Network Model. When we need to assume a network model, the one we use is that all parties communicate only to the adversary, who may forward messages from one party to another when the adversary wants or not at all. Both our assumptions and our soundness and accountability soundness security definitions assume this networking model.

The proof of our security properties works in general for *asynchronously safe* protocols. These have a number of safety properties which hold with asynchronous networking. Asynchronous networking means that the adversary decides when a message is delivered but must deliver all messages eventually. For safety properties, those which have a statement that holds always or never, this is equivalent to our network model.

I.2 A Formal Model for Consensus-based Accountable Light Client Design

We need the following fundamental notions:

- some number k of *epochs* with ids $1, \ldots, k$;
- for each epoch id i, $1 \leq i \leq k$, the validators on the blockchain may agree on a subset of the *set of possible consensus messages* M_i;
- associated with each consensus message m there may exist some *required data* $d_m \in D$ for some set D; when such a d_m exists, m is a binding commitment to d_m;
- a secure aggregatable signature scheme AS as defined in Section A.

Building on the above notions, we also define *a valid consensus view*.

Definition I.1. (Consensus View) A consensus view C for a set of epochs with ids i, $\forall i \in [k]$, for some k, contains for each epoch id i:

- a set PK_i of public keys (we may also consider a list of public keys and weights, e.g. proportional to stake, but we focus here on the equal weight case for simplicity).
- a set $\{(m, Signers, \sigma) \mid m \in M_i, Signers \subseteq PK_i\}$ where σ is a signature (or an aggregatable signature) on m and the public key(s) of the signer(s) are $Signers$.
- some *required data* d_m associated with each message m, such that m is a binding commitment to d_m. Note that some required data associated with different messages may overlap.

In addition to the components mentioned above, a consensus view C contains also a *genesis state* genstate; as a concrete example, genstate may contain the set of public keys PK_1 for the first epoch and their proofs of possession. For each of the notions contained in some epoch of C as well as for genstate, we say they belong to C and we simply denote that by "$\in C$".

In the following, we assume that all algorithms processing messages use a common efficient representation that implicitly includes for each of them an epoch id; this epoch id is retrieved using a function $epoch_{id}$.

Definition I.2. (Deciding a Consensus Message) Given a consensus view C, we say a message $m \in M_i$ is *decided in* C if C contains valid signatures from at least some threshold t (e.g., more than 2/3) signers corresponding to public keys in PK_i or, equivalently, a

valid aggregatable signature of t signers over m. Additionally, we denote by $(m, d_m) \in_{decided} C$ the fact that $m \in C, \exists\, d_m \in C \cap D$, d_m is the associated required data of m and m has been decided in C.

Definition I.3. (Valid Consensus View) We assume the following three *functions used for validation* are efficiently computable and they are defined as:

- $VerifyData : \cup_{i=1}^{k} M_i \times D \rightarrow \{1, 0\}$ such that it checks the validity of m given the required data d_m;
- $HistoricVerifyData : \{\text{genstate}\} \times (\cup_{i=1}^{k} M_i \times D)^n \times (\cup_{i=1}^{k} PK_i)^q \rightarrow \{1, 0\}$ such that it checks the validity of genstate, some set of n consensus messages and their required data and some set of q public keys;
- $Incompatible : \cup_{i=1}^{k} (M_i \times M_i) \times D \rightarrow \{0, 1\}$ which given messages m_1, m_2 and potential required data d_{m_1} for m_1 checks the incompatibility.

Let $m_1, \ldots, m_n$ be all the distinct consensus messages contained in C. Let $pk_1, \ldots, pk_q$ be all the public keys, including repetitions, contained in $PK_i, \forall i \in [k]$. We say *the consensus view C is valid* if:

- $\exists\, d_{m_i} \in D \cap C$ such that $VerifyData(m_i, d_{m_i}) = 1, \forall 1 \leq i \leq n$.
- $HistoricVerifyData(\text{genstate}, m_1, d_{m_1}, \ldots, m_n, d_{m_n}, pk_1, \ldots, pk_q) = 1$.
- There exists no pair (i, j), $1 \leq i, j \leq k$, $i \neq j$ such that $Incompatible(m_i, m_j, d_{m_i}) = 1$ or $Incompatible(m_j, m_i, d_{m_j}) = 1$.
- We require that all consensus messages in C are decided according to Definition I.2.

We conclude this subsection by defining what we mean by honest validator.

Definition I.4. (Honest Validator) An honest full node of a blockchain is one that runs the protocol correctly starting from the genesis state of the blockchain. It maintains a valid consensus view of the system. An honest full node is a validator if they produced a public key that is in the set PK_i in some epoch i in some consensus view. An honest validator is an honest full node that is also a validator.

I.2.1 General Light Client Properties.

Next we define a light client system.

Definition I.5. (Light Client System) Let $\mathcal{R}$ be a (conditional) NP relation. A *light client system* involves two parties - *prover* and *light client* (also called *light client verifier*) - and it implements the following algorithms:

- $pp_{LC} \leftarrow LC.Setup(\mathcal{R})$: a setup algorithm that takes the security parameter λ and a (conditional) NP relation $\mathcal{R}$ and outputs public parameters pp_{LC}.
- $\pi \leftarrow LC.GenerateProof(pp_{LC}, C, m, \mathcal{R})$: a proof generation algorithm that takes a valid consensus view C, a message m decided in consensus view C and a (conditional) NP relation $\mathcal{R}$ and generates a proof π.
- $acc/rej \leftarrow LC.VerifyProof(pp_{LC}, LC.seed, \pi, m, \mathcal{R})$: a proof verification algorithm that takes as input a genesis summary $LC.seed$ (whose properties are detailed in definition I.6), a light client proof π and a message m and returns acc if π is a valid proof for m and rej otherwise.

We call the tuple ($LC.Setup$, $LC.GenerateProof$, $LC.VerifyProof$) a light client system if it fulfils perfect completeness and soundness as defined below.

Perfect Completeness A light client system is perfectly complete if a full node sees that any message m has been decided, it can produce a proof that will convince a light client verifier of it. The full node should have a valid consensus view C that decided m which it can use as input in $LC.GenerateProof$ to obtain a proof π. The light client verifier will run $LC.VerifyProof$ with input π and this should always accept. Formally, we say ($LC.Setup$, $LC.GenerateProof$, $LC.VerifyProof$) has perfect completeness if for any valid consensus view C and for any consensus message m decided in C we have that

$$Pr[LC.VerifyProof(pp_{LC}, LC.seed, \pi, m, \mathcal{R}) = acc \mid pp_{LC} \leftarrow LC.Setup(\mathcal{R}),$$
$$\pi \leftarrow LC.GenerateProof(pp_{LC}, C, m, \mathcal{R})] = 1$$

Soundness A light client protocol is sound if, under the assumption that $v - f$ validators in each epoch are honest, the light client cannot be convinced of a message m unless $t - f$ honest validators have signed m. Here $f = v - t'$ is the a bound on the number of adversarial keys. Note that if $t - f$ honest validators sign m and there are f adversarial keys then additional signatures from these adversarial keys are enough to decide m. If the message m belongs to epoch k, then we assume that there is a valid consensus view C in which the validator sets for the first k epochs have t' honest validator's public keys. If this holds and less than $t - f$ honest validators signed m, then an adversary interacting with honest validators should not be able to generate a light client proof π for m that $LC.VerifyProof$ accepts.

We say ($LC.Setup$, $LC.GenerateProof$, $LC.VerifyProof$) has soundness if, for every efficient malicious prover $\mathcal{A}$,

$$\begin{aligned} Pr[&LC.VerifyProof(pp_{LC}, LC.seed, \pi, m, \mathcal{R}) = acc \mid pp_{LC} \leftarrow LC.Setup(\mathcal{R}),\\ &pp \leftarrow Parse(pp_{LC}), (\pi, m, C) \leftarrow \mathcal{A}^{HonestValidator}(pp, \mathcal{R}),\\ &CheckValidConsensus(C) = 1,\\ &NumberHonestSigners(m, OGenerateKeypair) < t + t' - \upsilon\\ &HonestThreshold(t', OGenerateKeypair, C) = 1] = negl(\lambda); \end{aligned}$$

where

- the predicate $CheckValidConsensus(C)$ checks if C is valid w.r.t. Definition I.3 and outputs 1 in that case (and 0 otherwise);
- $NumberHonestSigner(m, OGenerateKeypair)$ returns the number of public keys in Q_{pks} from $OGenerateKeypair$ defined below.
- $\mathcal{A}^{HonestValidator}$ represents the adversary A in communication with the honest validators.
- $HonestThreshold(t', OGenerateKeypair, C)$ checks that at least t' of the public keys in each PK_i of C (for every epoch i in C), are part of Q_{pks} and outputs 1 in that case (and 0 otherwise).

Finally, we assume that $HonestValidator$ (but not the adversary directly) makes oracles calls to $OGenerateKeypair(pp)$ (where pp are the public parameters of aggregated signature scheme AS are part of pp_{LC}) defined as

$$\begin{aligned} &OGenerateKeypair(pp):\\ &((pk, \pi_{PoP}), sk) \leftarrow AS.GenerateKeypair(pp)\\ &Q_{keys} \leftarrow Q_{keys} \cup \{((pk, \pi_{PoP}), sk)\}, Q_{pks} \leftarrow Q_{pks} \cup \{pk, \}\\ &\text{Output } ((pk, \pi_{PoP}), sk). \end{aligned}$$

Finally, we define the genesis summary and its properties with respect to a light client system.

Definition I.6. (Genesis Summary) Light client verifiers have access to a genesis summary $LC.seed$, which is a well defined deterministic function of the genesis state genstate.

I.2.2 Accountable Light Client Properties. In the following, we extend our model above to include accountability. We provide the definition for an accountable light client system which subsumes the light client system definition given above. An accountable light client has the property that if a full node with a consensus view C that decides m is given a light client proof π for a message m' that is incompatible with m, then it should be able to generate a proof that shows that some validators misbehaved. We need to add two more functions to our light client definition, the first one for detecting and generating proofs of misbehaviour, the second one for verifying the proofs of misbehaviour.

Definition I.7. (Accountable Light Client System) Let $\mathcal{R}$ be a (conditional) NP relation. An accountable light client system implements algorithms ($LC.Setup$, $LC.GenerateProof$, $LC.VerifyProof$, $LC.DetectMisbehaviour$, $LC.VerifyMisbehaviour$) where $LC.Setup$, $LC.GenerateProof$ and $LC.VerifyProof$ are defined as in I.5 and

$$(i, S, \mathbf{bit}, \sigma, m'', m') \leftarrow LC.DetectMisbehaviour(pp_{LC}, \pi, m, C, \mathcal{R})$$

is an algorithm such that it takes a proof π for message m, a consensus view C and a (conditional) NP relation $\mathcal{R}$; it outputs an epoch id i, a subset of misbehaving signers $S \subseteq PK_i$ in the same epoch as messages m'' and m', with m' decided in C and m'' signed with signature σ and using bitmask **bit** against the set PK_i and

$$acc/rej \leftarrow LC.VerifyMisbehaviour(pp_{LC}, i, S, \mathbf{bit}, \sigma, m'', m', C, \mathcal{R})$$

is an algorithm which takes the input of $LC.DetectMisbehaviour$ together with a consensus view C and a (conditional) NP relation $\mathcal{R}$ and checks if indeed misbehaviour took place such that completeness, soundness, accountability and accountability soundness hold, where completeness and soundness are identical to Definition I.5 and accountability completeness and accountability soundness are defined below.

Accountability Completeness A light client protocol has accountability completeness if a full node sees a light client proof for a message m and it sees that a message m' has been decided that is incompatible with m, then it can identify and prove that a fraction of validators ($\upsilon + \upsilon' - t$ validators) have misbehaved. The full node is given a proof π of m. It has a consensus view C that decides m', from the same epoch as m with required data $d_{m'}$ that has $Incompatible(m', m, d_{m'}) = 1$. Then it should be able to use $LC.DetectMisbehaviour$ to generate a proof that at least $\upsilon + \upsilon' - t$ validators misbehaved, that $LC.VerifyMisbehaviour$ will always accept.

Formally, we say ($LC.Setup$, $LC.GenerateProof$, $LC.VerifyProof$, $LC.DetectMisbehaviour$, $LC.VerifyMisbehaviour$) achieves accountability completeness if for every efficient adversary $\mathcal{A}$ it holds that:

$$\begin{aligned}
&Pr[LC.VerifyMisbehaviour(pp_{LC}, LC.DetectMisbehaviour(pp_{LC}, \pi, m, C, \mathcal{R}), C, \mathcal{R}) = acc \mid \\
&\quad pp_{LC} \leftarrow LC.Setup(\mathcal{R}), (\pi, m, C) \leftarrow \mathcal{A}(pp_{LC}, \mathcal{R}), \\
&\quad LC.VerifyProof(pp_{LC}, LC.seed, \pi, m, \mathcal{R}) = acc, CheckValidConsensus(C) = 1, \\
&\quad \exists\, (m', d_{m'}) \in_{decided} C, Incompatible(m', m, d_{m'}) = 1, epoch_{id}(m) = epoch_{id}(m')] = 1 - negl(\lambda)
\end{aligned}$$

Accountability Soundness A light client protocol has accountability soundness if an adversary interacting with a single honest validator cannot prove that the honest validator misbehaved. This holds even if all other validators are dishonest and the adversary controls the honest validator's view of the network.

Note that we assume that the adversary interacts with the honest validator, who generates their keys honestly in turn. The adversary can break accountability soundness if it can win the following game except with negligible probability. The adversary wins if they can produce an input $(i, S, \mathbf{bit}, \sigma, m'', m', C)$ to $LC.VerifyMisbehaviour$ such that $LC.VerifyMisbehaviour$ accepts, C is a valid consensus view and S contains a public key the honest validator generated.

Formally, we say ($LC.Setup$, $LC.GenerateProof$, $LC.VerifyProof$, $LC.DetectMisbehaviour$, $LC.VerifyMisbehaviour$) achieves accountability soundness if for every efficient adversary $\mathcal{A}$ it holds that:

$$Pr[Game^{accountability-soundness} = 1] = negl(\lambda)$$

where

$$\begin{aligned}
&Game^{accountability-soundness}(\lambda, \mathcal{R}) : \\
&Q_{keys} := \emptyset \\
&pp_{LC} \leftarrow LC.Setup(\mathcal{R}) \\
&pp \leftarrow Parse(pp_{LC}) \\
&(i, S, \mathbf{bit}, \sigma, m'', m', C) \leftarrow \mathcal{A}^{HonestValidator^{OGenerateKeypair}}(pp, pp_{LC}) \\
&\text{If } LC.VerifyMisbehaviour(pp_{LC}, i, S, \mathbf{bit}, \sigma, m'', m', C, \mathcal{R}) = rej \text{ Return } 0 \\
&\text{If } CheckValidConsensus(C) = 0 \text{ Return } 0 \\
&\text{If } S \cap Q_{pks} = \emptyset \text{ Return } 0 \\
&\text{Return } 1
\end{aligned}$$

I.3 Accountable Light Client System Instantiation

We motivate our light client model from I.2 by detailing below instantiations for a light client system that is accountable light client system. Both are compatible with proof-of-stake based blockchains and, in particular, Polkadot.

I.3.1 Conventions and Assumptions. Before listing our light client systems' algorithms, we make several notational conventions:

- We use boldface font for denoting vectors. Furthermore, whenever necessary to avoid confusion, we denote by $\mathbf{Vec_i}(k)$ the k-th component of vector $\mathbf{Vec_i}$.
- In the following, unless otherwise stated, when we use $\mathcal{R}$, we mean one of the conditional relations from the set $\{\mathcal{R}^{incl}_{\text{ba},com}, \mathcal{R}^{incl}_{\text{pa},com}\}$.
- Given a valid consensus view C over i epochs, we assume there is a well-defined order on the set PK_j of public keys included in C, $\forall j \in [i]$; hence, in the following, we rename this set by $\mathbf{pk_j}$, $\forall j \in [i]$ and interpret it as a vector. Moreover, we instantiate honestly generated keys in $\mathbf{pk_j}$ with keys generated using $AS.GenerateKeypair$ as described in instantiation A.2.
- We remind the reader that by $\mathbf{Com(pk)}$ we denote the set of two computationally binding polynomial commitments to the polynomials obtained by interpolating the x components of $\mathbf{pk}$ and, respectively, the y components of $\mathbf{pk}$ over a range H of size at least $v + 1$, where v is some maximum number of validators that the system allows. In our instantiations for (accountable) light client systems, we use the KZG polynomial commitments, but, as mentioned also in Section F, the general results stated in this section hold for any binding polynomial commitments with a knowledge-soundness property.
- We assume there is a fixed upper bound v on the number of validators in each epoch and we use v in the description of our algorithms. At the same time, for compatibility with the SNARKs that we build for relations $\mathcal{R}^{incl}_{\text{ba},com}$ and $\mathcal{R}^{incl}_{\text{pa},com}$ as defined in F.4, when specifically using our instantiation 5.4 of $CKS_{\mathcal{R}}$ or when proving our results in this section, we let v equal $n - 1$, where n was defined in Section 4.6.1.

- *Parse* and *Transform* denote functions performing the respective operations on the (accountable) light client algorithms' input in order to obtain the necessary components. *Parse* and *Transform* may additionally depend on the (conditional) relation $\mathcal{R}$ under consideration. If that is the case, we explicitly include $\mathcal{R}$. In particular, *Parse* and *Transform* functions which are part of $LC.DetectMisbehaviour$ work only for $\mathcal{R} \in \{\mathcal{R}^{incl}_{\text{ba},com}, \mathcal{R}^{incl}_{\text{pa},com}\}$.
- The accountable light client systems use functions f_x (deriving the public inputs), $f_{threshold}$ (deriving the Hamming weight), *HammingWeight* (deriving the Hamming weight from consensus view elements) and f_{bit} (deriving the bitmask corresponding to public keys that signed a given message). Before providing these functions' definitions, we make the convention that, whenever used as parameters/input to these functions, $\mathbf{bit}$, apk, $\mathbf{b'}$ and s have the meaning and definition provided in Section 5.

$$f_x(\mathbf{Com(pk)}, \mathbf{bit}, s, apk, \mathcal{R}) = \begin{cases} (\mathbf{Com(pk)}, \mathbf{bit}, apk) & \text{if } \mathcal{R} = \mathcal{R}^{incl}_{\text{ba},com} \\ (\mathbf{Com(pk)}, \mathbf{b'}, apk) & \text{if } \mathcal{R} = \mathcal{R}^{incl}_{\text{pa},com} \end{cases}$$

$$HammingWeight^*(\mathbf{vec}) = HammingWeight(\mathbf{vec}_1, \ldots, \mathbf{vec}_{|vec|-1})$$

$$f_{threshold}(\mathbf{x}, \mathcal{R}) = \begin{cases} HammingWeight^*(\mathbf{bit}) & \text{if } \mathcal{R} = \mathcal{R}^{incl}_{\text{ba},com} \\ \sum_{j=1}^{\frac{\upsilon+1}{|block|}-1} HammingWeight(\mathbf{b'}_j) \; + \; HammingWeight^*(\mathbf{b'}_{\frac{\upsilon+1}{|block|}}) & \text{if } \mathcal{R} = \mathcal{R}^{incl}_{\text{pa},com} \end{cases}$$

$$f_{bit}(C, m, \upsilon) = ((\mathbf{bit_i}(k))_{k=1}^{\upsilon} || \; 0, \sigma_{\mathbf{i}}),$$

where $i = epoch_{id}(m)$ and $\forall\ k = 1, \ldots, \upsilon$, if there exists $\sigma \in C \ \wedge \ AS.Verify(pp, \mathbf{pk_i}(k), m, \sigma) = 1$, we set $\mathbf{bit_i}(k) = 1$ and $\sigma_{\mathbf{i}}(k) = \sigma$, otherwise, we set $\mathbf{bit_i}(k) = 0$ and $\sigma_{\mathbf{i}}(k) = \_$.
Note that for each of our relations $\mathcal{R}^{incl}_{\text{ba},com}$ and $\mathcal{R}^{incl}_{\text{pa},com}$, apk and $\mathbf{Com(pk)}$ are public inputs and $\mathbf{pk}$ is a witness. Moreover, for these relations $\mathcal{R}^{incl}_{\text{ba},com}$ and $\mathcal{R}^{incl}_{\text{pa},com}$, we build an accountable light client system.
- We make the following instantiations: genstate is the set of public keys in $\mathbf{pk_1}$ and their alleged proofs of possession; $LC.seed = \mathbf{Com(pk_1)}$.

I.3.2 The Algorithms. The setup algorithm used by the accountable light client system is:

- $LC.Setup(\mathcal{R})$

$$(pp, rs_{pk}, rs_{vk}) \leftarrow CKS_{\mathcal{R}}.Setup(\upsilon)$$
$$Return\ (pp, rs_{pk}, rs_{vk})$$

The four algorithms that are part of the accountable light client system are:

- $LC.GenerateProof(pp, rs_{pk}, C, m, \mathcal{R})$

$\mathbf{\Pi} = (); \ \Sigma = ()$
$i = epoch_{id}(m)$
For $j = 1, \ldots, i$
  If $j < i$
    $m_j = (j, \mathbf{Com(pk_{j+1})})$
  Else
    $m_j = m$
  $(\mathbf{bit_j}, \sigma_{\mathbf{j}}) = f_{bit}(C, m_j, \upsilon)$
  $\Sigma(j) \leftarrow AS.AggregateSignatures(pp, (\sigma_{\mathbf{j}}(k))_{k=1}^{\upsilon})$
  $(\pi_{SNARK,j}, apk_j) \leftarrow CKS_{\mathcal{R}}.Prove(rs_{pk}, \mathbf{Com(pk_j)}, (\mathbf{pk_j}(k))_{k=1}^{\upsilon}, (\mathbf{bit_j}(k))_{k=1}^{\upsilon})$
  $\mathbf{x_j} = f_x(\mathbf{Com(pk_j)}, \mathbf{bit_j}, s, apk_j, \mathcal{R})$
  $\mathbf{\Pi}(j) = (\mathbf{x_j}, \pi_{SNARK,j})$
Return $(\mathbf{\Pi}, \Sigma)$

- $LC.VerifyProof(pp, rs_{vk}, LC.seed, \pi, m, \mathcal{R})$

$i = epoch_{id}(m)$
$(\mathbf{\Pi}, \Sigma) = Parse(\pi)$;
For $j = 1, \ldots, i$
  $(\mathbf{x_j}, \pi_{SNARK,\mathbf{j}}) = \mathbf{\Pi}(j)$; $(com_j, \mathbf{bit_j}, apk_j) = Parse(\mathbf{x_j}, \mathcal{R})$
If $LC.seed \neq com_1$
  Return rej
For $j = 1, \ldots, i$
  If $j < i$
    $m_j = (j, com_{j+1})$
  Else
    $m_j = m$
  $threshold_j = f_{threshold}(\mathbf{x_j}, \mathcal{R})$
  If $(CKS_{\mathcal{R}}.Verify(pp, rs_{vk}, com_j, m_j, \Sigma(j), (\pi_{SNARK,\mathbf{j}}, apk_j), \mathbf{bit_j}) = 0) \vee (threshold_j < t)$
    Return rej
Return acc

- $LC.VerifyMisbehaviour(pp, i, S, \mathbf{bit}, \sigma, m'', m', C)$

$apk = AS.AggregateKeys(pp, (\mathbf{bit}(k) \cdot \mathbf{pk_i}(k))_{k=1}^{\upsilon})$
$(\mathbf{bit}', \_) = f_{bit}(C, m', \upsilon)$
Compute $S_{m''} = \{\mathbf{pk_i}(k) \mid \mathbf{bit}(k) = 1, k \in [\upsilon]\}$
Compute $S_{m'} = \{\mathbf{pk_i}(k) \mid \mathbf{bit}'(k) = 1, k \in [\upsilon]\}$
If $(AS.Verify(pp, apk, m'', \sigma) = 1) \wedge (S_{m''} \cap S_{m'} = S) \wedge (|S_{m'}| \geq t) \wedge (|S_{m''}| \geq t) \wedge$
    $\wedge ((m', d_{m'}) \in_{decided} C) \wedge$
    $\wedge (i = epoch_{id}(m'') = epoch_{id}(m')) \wedge (Incompatible(m'', m', d_{m'}) = 1)$
  Return acc
Else
  Return rej

- $LC.DetectMisbehaviour(pp, rs_{vk}, \pi, m, C, \mathcal{R})$

$(\mathbf{\Pi}, \Sigma) = Parse(\pi)$
$i = epoch_{id}(m)$
$index = i$
$m'' = m$
For $j = 1, \ldots, i$
$(\mathbf{x_j}, \pi_{SNARK,j}) = \mathbf{\Pi}(j)$; $(apk_j, com_j) = Parse(\mathbf{x_j})$
If $(LC.VerifyProof(pp, rs_{vk}, LC.seed, \pi, m, \mathcal{R}) = 1) \ \wedge \ (\exists \ min \ 2 \leq j \leq i, \ com_j \neq \mathbf{Com}(\mathbf{pk_j}))$
$m'' = (j-1, com_j)$; $m' = (j-1, \mathbf{Com}(\mathbf{pk_j}))$; $index = j-1$
ElseIf $(LC.VerifyProof(pp, rs_{vk}, LC.seed, \pi, m, \mathcal{R}) = 1) \ \wedge \ (\forall \ 2 \leq j \leq i, \ com_j = \mathbf{Com}(\mathbf{pk_j})) \wedge$
$\wedge \ (\exists \ (aux, d_{aux}) \in_{decided} C) \ \wedge \ Incompatible(aux, m'', d_{aux}) = 1)$
$m' = aux$
Else Return
$\mathbf{bit} = Transform(Parse(\mathbf{x}_{index}, \mathcal{R}), \mathcal{R})$
Compute $S_{m''} = \{\mathbf{pk_{index}}(k) \mid \mathbf{bit}(k) = 1, k \in [\upsilon]\}$
$(\mathbf{bit}', \_) = f_{bit}(C, m', \upsilon)$
Compute $S_{m'} = \{\mathbf{pk_{index}}(k) \mid \mathbf{bit}'(k) = 1, k \in [\upsilon]\}$
Return $(index, S_{m''} \cap S_{m'}, \mathbf{bit}, \Sigma(index), m'', m')$

I.3.3 Assumptions and Security Proofs. We complete our instantiation by proving the security properties of our light client and accountable light client systems according to definitions introduced in Sections I.2.1 and I.2.2. However, beforehand, we present the assumptions we use, of which there are six classes, i.e., there are assumptions about honest validators' behaviour (B), about consensus (C), about parameters (P), about instantiation of primitives (S), about genesis state (G) and assumptions about light client integration (I).

The assumptions about honest validators' behaviour are:

- (B.1.) An honest validator never signs a message m unless it knows some required data d_m such that $VerifyData(m, d_m) = 1$ holds.
- (B.2.) An honest validator never signs a message m such that $VerifyData(m, d_m) = 1$ holds if they have previously signed m' such that $VerifyData(m', d_{m'}) = 1$ holds and
$Incompatible(m, m', d_m) = 1$ or $Incompatible(m', m, d_{m'}) = 1$ hold.
- (B.3.) An honest validator does not sign any message in M_i unless they have a valid consensus view C (with $M_i \subset C$) for which their public key is in $\mathbf{pk_i}$ with $\mathbf{pk_i} \in C$.

The assumptions about consensus are:

- (C.1.) The adversary interacting with honest validators should not except with negligible probability be able to produce both: (i) a valid consensus view C in which at least t' validators in every epoch are honest that decides some message m with d_m such that $VerifyData(m, d_m) = 1$ and (ii) a valid consensus view C' with the same genesis state as C (in particular, with the same $\mathbf{pk_1} \subset$ genstate) which decides some message m' in the same epoch as m, with $Incompatible(m, d_m, m') = 1$.
- (C.2.) The adversary interacting with honest validators should not except with negligible probability be able to produce both: (i) a valid consensus view C in which at least t' validators in every epoch are honest and (ii) a valid consensus view C' with the same genesis state as C (in particular, with the same $\mathbf{pk_1} \subset$ genstate) in which there is some epoch i that C and C' both reach with $\mathbf{pk_i} \neq \mathbf{pk_i'}$.

The assumptions about parameters are:

- (P.1.) $2t - \upsilon > 0$
- (P.2.) $t + t' > \upsilon$

The assumption about instantiation of primitives is:

- (S.1.) We instantiate the aggregatable signature scheme AS such that the oracle $OSign$ in Definition 4.1 (in particular in the unforgeability property definition), is replaced with $OSpecialSign$. It is easy to see that if AS is an aggregatable signature scheme secure according to Definition 4.1, then AS is also an aggregatable signature with oracle $OSign$ replaced by $OSpecialSign$ in Definition 4.1.

The assumptions about genesis state are:

- (G.1.) In a valid consensus view, *HistoricVerifyData* checks, among others, that a) every $pk \in \mathbf{pk_1}$ is also part of genstate, b) that every $pk \in \mathbf{pk_1}$ is in $\mathbb{G}_{1,inn}$ and c) that the proofs of possession for each of the public keys in $\mathbf{pk_1}$ pass the verification in $AS.VerifyPoP$.
- (G.2.) We assume that all honest full nodes and validators have access to the same genesis state genstate even when the genesis state is generated by a potential adversary.

Before the last class of assumptions, we add two notational conventions in the form of two functions:

- $NextEpochKeys(m, d_m)$ returns $\perp$ or a list of public keys; if $epoch_{id}(m) = i$, these keys are supposed to be the public keys of epoch $i + 1$.
- $IsCommitment(m)$ returns 0 or 1; $IsCommitment(m) = 1$ iff there exists some i such that $m = (i, \mathbf{Com}(\mathbf{pk_{i+1}}))$.

Finally, we make the following light client integration assumptions, i.e., these are assumptions that apply to our specific light client instantiation:

- (I.1.) If m and m' are such that $epoch_{id}(m) = epoch_{id}(m')$ and $NextEpochKeys(m, d_m) \neq \perp$ and $IsCommitment(m') = 1$ and $m' \neq (epoch_{id}(m), \mathbf{Com}(NextEpochKeys(m, d_m)))$ then

$$Incompatible(m, m', d_m) = 1.$$

- (I.2.) If $epoch_{id}(m) = i$ and $NextEpochKeys(m, d_m) = \mathbf{pk_{i+1}}$, then $ValidateData(m, d_m)$ must call $AS.VerifyPoP(pp, pk, \pi_{POP})$ for each $pk \in \mathbf{pk_{i+1}}$ and some data $\pi_{POP} \in d_m$ and also check that $pk \in \mathbb{G}_{1,inn}$; if any of these checks fails, then $ValidateData(m, d_m)$ fails.
- (I.3.) An honest validator with a valid consensus view C, does not sign a message m' with $IsCommitment(m') = 1$ unless there exists a message m decided in C and its required data d_m (i.e., $ValidateData(m, d_m) = 1$) such that

$$m' = (epoch_{id}(m), \mathbf{Com}(NextEpochKeys(m, d_m))).$$

- (I.4.) If *HistoricVerifyData* outputs 1 and there exist a message $m \in C$ that has been decided in epoch i, then for all $1 \leq j < i$, $(j, \mathbf{Com}(\mathbf{pk_{j+1}}))$ was decided in epoch j.
- (I.5.) If *HistoricVerifyData* outputs 1 and a message m' has been decided in C such that $IsCommitment(m') = 1$, then there exist $m, d_m \in C$ with $ValidateData(m, d_m) = 1$, m decided in C and $epoch_{id}(m) = epoch_{id}(m')$ such that

$$\mathbf{pk}_{epoch_{id}(\mathbf{m})+\mathbf{1}} = NextEpochKeys(m, d_m).$$

We are now ready to state and prove the security properties of our (accountable) light client systems.

Theorem I.8. *If AS is the secure aggregatable signature scheme defined in instantiation A.2 and if $CKS_{\mathcal{R}}$ is the secure committee key scheme defined in instantiation 5.4, then, together with the assumptions stated at the beginning of Section I.3.3 and for $\mathcal{R} \in \{\mathcal{R}^{incl}_{\text{ba},com}, \mathcal{R}^{incl}_{\text{pa},com}\}$, the tuple (LC.Setup, LC.GenerateProof , LC.VerifyProof) as instantiated in Section I.3.2 is a light client system.*

Proof. We include the full proof in Section J. □

Theorem I.9. *If AS is the secure aggregatable signature scheme defined in instantiation A.2 and if $CKS_{\mathcal{R}}$ is the secure committee key scheme defined in instantiation 5.4, then, together with the assumptions stated at the beginning of Section I.3.3 and for $\mathcal{R} \in \{\mathcal{R}^{incl}_{\text{ba},com}, \mathcal{R}^{incl}_{\text{pa},com}\}$, the tuple (LC.Setup, LC.GenerateProof , LC.VerifyProof , LC.DetectMisbehaviour, LC.VerifyMisbehaviour) as instantiated in Section I.3.2 is an accountable light client system.*

Proof. Due to theorem I.8, the tuple (*LC.Setup*, *LC.GenerateProof* , *LC.VerifyProof* , *LC.DetectMisbehaviour*, *LC.VerifyMisbehaviour*) as instantiated in Section I.3.2 is already a light client system. It is only left to show that both accountability completeness and accountability soundness also hold and we include the full details of this proof in Section K. □

Corollary I.10. *In an accountable light client system, the number of misbehaving validators output by LC.DetectMisbehaviour is $|S|$ and $|S| > 0$.*

Proof. Due to theorem K.1 and accountability completeness, given a valid consensus view C, a verifying light client proof π for a message m'' and given the existence in C of a message m' incompatible with m'', the number of validators that *LC.DetectMisbehaviour* is able to catch is at least $|S|$. Moreover, due to accountability soundness, any public key output by *LC.DetectMisbehaviour*, e.w.n.p. , belongs to a misbehaving validator. Finally, using again accountability completeness and, in particular, since *LC.VerifyMisbehaviour* accepts with overwhelming probability the output of an honest party running *LC.DetectMisbehaviour* and due to assumption (P.1.) it holds that:

$$|S| = |S_{m'} \cap S_{m''}| = |S_{m'}| + |S_{m''}| - |S_{m'} \cup S_{m''}| \geq t + t - v > 0.$$

□

J POSTPONED SECURITY PROOF FOR LIGHT CLIENT SYSTEMS

In this section we prove the following theorem:

Theorem J.1. *If AS is the secure aggregatable signature scheme defined in instantiation A.2 and if $CKS_{\mathcal{R}}$ is the secure committee key scheme defined in instantiation 5.4, then, together with the assumptions stated at the beginning of Section I.3 and for $\mathcal{R} \in \{\mathcal{R}^{incl}_{\text{ba},com}, \mathcal{R}^{incl}_{\text{pa},com}\}$, the tuple (LC.Setup, LC.GenerateProof , LC.VerifyProof) as instantiated in Section I.3 is a light client as formalised by Definition I.5.*

Proof. *Perfect Completeness:* Let m be a message decided in some epoch i of a valid consensus view C. Since C is a valid consensus view, this implies *HistoricVerifyData* outputs 1. Adding that m has been decided in epoch i and using assumption (I.4.), we have that for each previous epoch $j \in [i-1]$, $(j, \mathbf{Com}(\mathbf{pk_{j+1}}))$ was decided in epoch j; we denote this as property $(*)$. Since

$$IsCommitment(j, \mathbf{Com}(\mathbf{pk_{j+1}})) = 1, \forall j \in [i-1]$$

holds and using assumptions (I.5.), (I.2.) and (G.1.), we conclude the proofs of possession for each of the public keys in $\mathbf{pk_j}$, $j \in [i]$ pass the verification *AS.VerifyPoP* (property $(**)$) and, as a consequence, each of the public keys in $\mathbf{pk_j}$, $j \in [i]$ belong to $\mathbb{G}_{1,inn}$ (property $(***)$). The main fact we have to show (with the notation used in the description of *LC.VerifyProof*), is that the following two predicates hold:

$$AS.Verify(pp, apk_j, m_j, \Sigma(j)) = 1, \forall j \in [i] \quad (1)$$

and

$$threshold_j \geq t, \forall j \in [i] \quad (2).$$

Indeed, (1) holds due to perfect completeness for aggregation for secure signature scheme instantiation *AS* which applies because: (a) for every epoch $j \in [i]$, as computed by *LC.GenerateProof*, each of the individual signatures aggregated into $\Sigma(j)$ passes *AS.Verify*, (b) the aggregation $\Sigma(j)$ is computed correctly as per *LC.GenerateProof*, (c) the proofs of possession have been checked for each of the public keys in $\mathbf{pk_j}$, $\forall j \in [i]$ (see property $(**)$), and, finally, (d) the aggregation of public keys denoted by apk_j, $\forall\, j \in [i]$, has been computed correctly as $(\mathbf{bit_j}(k) \cdot \mathbf{pk_j}(k))_{k=1}^{\upsilon}$ due to property $(***)$ and the perfect completeness of the SNARK scheme for relation $\mathcal{R}$ invoked by the instantiation of $CKS_{\mathcal{R}}.Prove$.
Moreover, due to definition of $f_{threshold}$ and the fact that $m_j = (j, \mathbf{Com}(\mathbf{pk_{j+1}}))$, $\forall j \in [i-1]$ and, respectively, $m_i = m$ have been decided in their respective epochs as per $(*)$, we have that (2) holds.

Finally, using (1), letting $ck_j = com_j = \mathbf{Com}(\mathbf{pk_{j+1}})$, $\forall\, j \in [i]$ and since $\forall\, j \in [i]$, $\pi_{SNARK,j}$, apk_j and ck_j are honestly computed as described by *LC.GenerateProof* and invoking the perfect completeness property of the $CKS_{\mathcal{R}}$ committee key scheme, we obtain that

$$CKS_{\mathcal{R}}.Verify(pp, rs_{vk}, \mathbf{Com}(\mathbf{pk_{j+1}}), m_j, \Sigma(j), (\pi_{SNARK,j}, apk_j), \mathbf{bit_j}) = 1, \forall j \in [i] \quad (3).$$

In turn, the fact that (2) and (3) hold with probability 1 immediately implies

$$LC.VerifyProof(pp_{LC}, LC.seed, LC.GenerateProof(pp_{LC}, C, m, \mathcal{R}), m, \mathcal{R}) = acc$$

with probability 1 (q.e.d.).

Soundness: In order to prove soundness, we first state and prove the following:

Proposition J.2. *Given an efficient adversary $\mathcal{A}$ as defined in the soundness game (Definition I.5) and let (π, m, C) be its corresponding output.*

Let $i = epoch_{id}(m)$. Assuming that

$$LC.VerifyProof(pp_{LC}, LC.seed, m, \mathcal{R}) = acc$$

and $CheckValidConsensus(C) = 1$ and $HonestThreshold(t', OGenerateKeypair, C) = 1$ (i.e., the light client proof π is accepted, C is a valid consensus view as per Definition I.3 and for each epoch k in C, PK_k contains at least t' honest validators), then:

- *Statement $A(j)$: For $j < i$, assuming further that $com_j = \mathbf{Com}(\mathbf{pk_j})$, then there exists some honest validator whose key is in $\mathbf{pk_j}$ such that it signed $m_j = (j, \mathbf{Com}(\mathbf{pk_{j+1}}))$, except with negligible probability.*
- *Statement $B(j)$: For $j < i$, if an honest validator whose key is in $\mathbf{pk_j}$ signed m_j with $epoch_{id}(m_j) = j$ and $IsCommitment(m_j) = 1$ then $m_j = (j, \mathbf{Com}(\mathbf{pk_{j+1}}))$.*

Proof. (Proposition) We prove the proposition above by induction. Moreover, we prove the proposition only for $\mathcal{R} = \mathcal{R}^{incl}_{\text{ba},com}$. The proposition can be proven analogously for $\mathcal{R} = \mathcal{R}^{incl}_{\text{pa},com}$. Proving the base case, namely that $A(1)$ holds under the assumption G.1. and proving that $A(j)$ holds if $B(j-1)$ holds follows a very similar proof structure hence we give complete details only for the latter and add only the differences for the former. We complete the induction step by proving that if $A(j)$ holds then $B(j)$ holds.

First proof of the induction step: Assume that statement $B(j-1)$ holds. We have to prove that $A(j)$ holds. Due to the assumption that the light client proof π is accepted and due to the definition of step j in algorithm *LC.VerifyProof*, we have that properties (1) and (2) as described below hold, except with negligible probability, where

$$(CKS_{\mathcal{R}}.Verify(pp, rs_{vk}, com_j, m_j, \Sigma(j), (\pi_{SNARK,\mathbf{j}}, apk_j), \mathbf{bit_j}) = 1) \quad (1)$$

and

$$(threshold_j \geq t) \quad (2)$$

Due to instantiation 5.4, (1), in turn, is equivalent to properties (3) and (4) holding, except with negligible probability, where:

$$AS.Verify(pp, apk_j, m_j, \Sigma(j)) = 1 \quad (3)$$

and

$$SNARK.Verify(rs_{vk}, (com_j, \mathbf{bit_j}||0, apk), \pi_{SNARK}, \mathcal{R}) = 1 \quad (4)$$

By the knowledge soundness property of the hybrid model SNARK for relation $\mathcal{R}$ and algorithm *SNARK.PartInputs* defined in Section F (where $c(\mathbf{pk_j}) = incl(\mathbf{pk_j}) = 1$ iff $\mathbf{pk_j} \in \mathbb{G}_{1,inn}^{n-1}$ holds) and since (4) holds and since $\mathbf{pk_j} \in \mathbb{G}_{1,inn}^{n-1}$ holds as a consequence of the fact that the proofs of possession for each of the public keys in $\mathbf{pk_j}$ pass the verification in *AS.VerifyPoP* (which, in turn, holds since $B(j-1)$ holds plus due to integration assumptions I.1.- I.3. and the definition of *IsCommitment*), it means that, extractor $\mathcal{E}$ (as described in Definition B.1 can extract w such that

$$(\mathbf{x_j} = (com_j, \mathbf{bit_j}||0, apk_j), w = \mathbf{pk}') \in \mathcal{R},$$

except with negligible probability. In particular, this means $apk_j = \sum_{k=1}^{n-1} \mathbf{bit_j}(k) \cdot \mathbf{pk}'(k)$ and $\mathbf{Com}(\mathbf{pk}') = com_j$. By the computational binding of the KZG commitment used in defining com_j and by the fact that $com_j = \mathbf{Com}(\mathbf{pk_j})$ by assumption (i), we obtain that $\mathbf{pk}' = \mathbf{pk_j}$, hence

$$apk_j = \sum_{k=1}^{n-1} \mathbf{bit_j}(k) \cdot \mathbf{pk_j}(k) \quad (5)$$

which, in turn, by the definition of aggregatable signature scheme *AS* in instantiation A.0.1 is equivalent to:

$$apk_j = AS.AggKeys(pp, (\mathbf{pk_j}(k))_{k=1}^{n-1}) \quad (6)$$

Next, we look at (2) which is equivalent to $HammingWeight^*(\mathbf{bit_j}) \geq t$ (7); (7) together with the fact that there are at least t' honest validators in $\mathbf{pk}$ (since $HonestThreshold(t', OGenerateKeypair, C) = 1$) and the assumption P.2. that $t + t' \geq v = n - 1$, we obtain that there exists at least an honest validator in $\mathbf{pk_j}$ whose public key is aggregated into apk_j. We denote this as property (8).

Finally, it is clear that due to (3), (6), (8) and since the proofs of possession for each of the public keys in $\mathbf{pk_j}$ pass the verification in AS.VerifyPoP (in turn, since $B(j-1)$ holds and due to integration assumptions I.1.- I.3. and the definition of *IsCommitment*), the statement $A(j)$ becomes equivalent to showing that the advantage $Adv_{\mathcal{A}_{sound}}^{multiforge}(\lambda)$ in the following game is negligible (9), where, in general,

$$Adv_{\mathcal{A}}^{multiforge}(\lambda) = Pr[Game_{\mathcal{A}}^{multiforge}(\lambda) = 1]$$

and

$Game_{\mathcal{A}}^{multiforge}(\lambda)$:
$pp \leftarrow AS.Setup(aux_{AS})$
$((pk_k^*, \pi_{k,PoP}^*), sk_k^*)_{k=1}^{t'} \leftarrow AS.GenKeypair(pp)$
$Q \leftarrow \emptyset$
$((pk_k, \pi_{k,PoP})_{k=1}^{u}, m, asig) \leftarrow \mathcal{A}^{OMSign}(pp, (pk_k^*, \pi_{k,PoP}^*)_{k=1}^{t'})$
If $\exists\ k \in [t'], pk_k^* \notin \{pk_i\}_{i=1}^{u} \vee (m, pk_k^*) \in Q$, *then return* 0
For $i \in [u]$
  If $AS.VerifyPoP(pp, pk_i, \pi_{PoP,i}) = 0$ *return* 0
$apk \leftarrow AS.AggKeys(pp, (pk_i)_{i=1}^{u})$
Return $AS.Verify(pp, apk, m, asig)$

and

$$
\begin{aligned}
&OMSign(m_k, pk^*):\\
&If\ pk^* \in Q_{keys|pk}\\
&\quad \sigma_j \leftarrow AS.Sign(pp, sk^*, m_k)\\
&\quad Q \leftarrow Q \cup \{(m_k, pk^*)\}\\
&\quad return\ \ \sigma_k\\
&Else\\
&\quad return
\end{aligned}
$$

and $\mathcal{A}_{sound}$ is defined such that $asig = \Sigma(\mathbf{j})$, $m = m_j$, $apk = apk_j$ and the public keys output by $\mathcal{A}_{sound}$ are the non-zero public keys from the vector $(\mathbf{bit_j}(k) \cdot \mathbf{pk_j}(k))_{k=1}^{n-1}$.

We prove statement (9) by contradiction: if we assume the advantage $Adv^{multiforge}_{\mathcal{A}_{sound}}(\lambda)$ is non-negligible, then, using a standard hybrid argument and reducing the game $Game^{multiforge}_{\mathcal{A}}(\lambda)$ to the game $Game^{forge}_{\mathcal{A}}(\lambda)$ as per Definition 4.1, the advantage $Adv^{forge}_{\mathcal{A}_{sound}}(\lambda)$ is also non-negligible; however, this, in turn, contradicts the fact that the instantiation AS is an unforgeable aggregatable signature scheme, hence our proof for $A(j)$ is complete.

Observation: In the case of the proof for $A(1)$, the only difference is that the proofs of possession for each of the public keys in $\mathbf{pk_1}$ pass the verification in $AS.VerifyPoP$ by assumption G.1. By the definition of aggregatable signature scheme AS, as the consequence, $\mathbf{pk_1} \in \mathbb{G}^{n-1}_{1,inn}$.

Second proof of the induction step: Assume that statement $A(j)$ holds. Assume by contradiction that $B(j)$ does not hold, i.e., an honest validator $HVal$ whose key is in $\mathbf{pk_j}$ signed m_j such that $IsCommitment(m_j) = 1$ and $m_j \neq (j, \mathbf{Com}(\mathbf{pk_{j+1}}))$ (we call this property (10)). Due to assumption I.3, $HVal$ does not sign m_j unless $HVal$ has a valid consensus view C' deciding a message m' with required data $d_{m'}$ and $m_j = (j, \mathbf{Com}(NextEpochKeys(m', d_{m'}))$ (we call this property (11)). By (10) and (11) and the fact that the commitment scheme used to compute $\mathbf{Com}(\cdot)$ is binding, we obtain:

$$NextEpochKeys(m', r_{m'}) \neq \mathbf{pk_{j+1}} \quad (12).$$

By assumptions I.3. and I.4, there exists in epoch j of valid consensus view C some decided message m'_j with $epoch_{id}(m'_j) = j$ and $m'_j = \mathbf{Com}(\mathbf{pk_{j+1}})$. Then, by assumption I.1, m'_j and m' are incompatible. This, in turn, contradicts assumption C.1. combined with assumption G.2. since C and C' decided in epoch j messages m'_j and m', respectively. Hence our initial assumption is false and $B(j)$ is proven to hold. And our proposition proof is complete. □

We are now able to prove the soundness property. Given an efficient adversary $\mathcal{A}$ as defined in the soundness game (Definition I.5) and let (π, m, C) be its corresponding output. Let $i = epoch_{id}(m)$. Assuming that

$$LC.VerifyProof(pp_{LC}, LC.seed, m, \mathcal{R}) = acc$$

and $CheckValidConsensus(C) = 1$ and $HonestThreshold(t', OGenerateKeypair, C) = 1$, then, using the proposition above, we obtain that statement $B(i-1)$ holds. Then, letting $m_i = m$ and with an analogous reasoning used for proving the induction step, namely that $A(j)$ holds when $B(j)$ holds (please see proof above) we are able to conclude that m was signed by an honest validator only with negligible probability (q.e.d). □

K POSTPONED SECURITY PROOF FOR ACCOUNTABLE LIGHT CLIENT SYSTEMS

In this section we prove the following theorem:

Theorem K.1. *If AS is the secure aggregatable signature scheme defined in instantiation A.2 and if $CKS_{\mathcal{R}}$ is the secure committee key scheme defined in instantiation 5.4, then, together with the assumptions stated at the beginning of Section I.3.3 and for $\mathcal{R} \in \{\mathcal{R}^{incl}_{\text{ba},com}, \mathcal{R}^{incl}_{\text{pa},com}\}$, the tuple (LC.Setup, LC.GenerateProof, LC.VerifyProof, LC.DetectMisbehaviour, LC.VerifyMisbehaviour) as instantiated in Section I.3.2 is an accountable light client system.*

Proof. Due to theorem I.8, the tuple ($LC.Setup$, $LC.GenerateProof$, $LC.VerifyProof$, $LC.DetectMisbehaviour$, $LC.VerifyMisbehaviour$) as instantiated in Section I.3.2 is already a light client system. It is only left to show that both accountability completeness and accountability soundness also hold. Indeed:

Accountability Completeness: Let $\mathcal{A}$ be an efficient adversary that on input pp_{LC} and $\mathcal{R}$ outputs π, m and C. It is easy to see that if the descriptions of *LC.DetectMisbehaviour* and *LC.VerifyMisbehaviour* are followed honestly, then the predicate $S_{m''} \cap S_{m'} = S$ checked in the end of *LC.VerifyMisbehaviour* is fulfilled. Moreover, due to the satisfied predicate

$$LC.VerifyProof(pp, rs_{vk}, LC.seed, \pi, m, \mathcal{R}) = 1 \quad (1)$$

it holds that all $m_j, j \in [i]$ (as defined in *LC.VerifyProof*) are decided in C. Due to the way m' and m'' are computed by *LC.DetectMisbehaviour* from the messages $(m_j)_{j=1}^{i}$, this implies $|S_{m'}| \geq t$, $|S_{m''}| \geq t$ and $(m', d_{m'}) \in_{decided} C$ and

$$Incompatible(m'', m', d_{m'}) = 1.$$

We are only left to show that

$$AS.Verify(pp, apk, m'', \sigma) = 1 \quad (*)$$

holds with overwhelming probability. Indeed, since (1) holds then, for every epoch $j \in [i]$ it holds that

$$CKS_{\mathcal{R}}.Verify(pp, rs_{vk}, com_j, m_j, \Sigma(j), (\pi_{\mathbf{j}}, apk_j), \mathbf{bit_j}) = 1 \quad (2).$$

In particular, (2) holds for $j = index$. Due to soundness property of the committee key scheme $CKS_{\mathcal{R}}$, since $com_{index} = \mathbf{Com}(\mathbf{pk_{index}})$ by the definition of *index* and *LC.DetectMisbehaviour*, since $apk_{index} = AS.AggKeys(pp, (\mathbf{bit_{index}}(k) \cdot \mathbf{pk_{index}}(k))_{k=1}^{\upsilon})$ as computed by *LC.DetectMisbehaviour*, since also $m'' = m_{index}$ (with $m_j, \forall j \in [i]$ defined in *LC.VerifyProof* and *index* defined in *LC.DetectMisbehaviour*) and, finally, since $\Sigma(index) = \sigma$, as defined in *LC.DetectMisbehaviour*, it follows that $(*)$ holds with overwhelming probability (q.e.d.).

Accountability Soundness: Let $\mathcal{A}$ be an efficient adversary who interacts with an honest validator. If $LC.VerifyMisbehaviour(pp, i, S, \mathbf{bit}, \sigma, m'', m', C)$ outputs *acc* $(*)$, its checks together with completeness for aggregation imply

$$AS.Verify(pp, \sigma', m', apk_S) = 1 \ (**),$$

where

$$\sigma_{\mathbf{i}}(j) = \begin{cases} sig & \text{if } \exists\ sig \in C, AS.Verify(pp, sig, m', \mathbf{pk_i}(j)) \\ \_ & \text{otherwise} \end{cases}$$

$$\mathbf{b_S}(j) = \begin{cases} 1 & \text{if } \mathbf{pk_i}(j) \in S \\ 0 & \text{otherwise} \end{cases}$$

$$\sigma' \leftarrow AS.AggSigs(pp, (\mathbf{b_S}(j) \cdot \sigma_{\mathbf{i}}(j))_{j=1}^{\upsilon}),$$

$$apk_S \leftarrow AS.AggKeys(pp, (\mathbf{b_S}(j) \cdot \mathbf{pk_i}(j))_{j=1}^{\upsilon}),$$

Additionally, since $(*)$ holds and for *apk* as defined in *LC.VerifyMisbehaviour*, we obtain

$$AS.Verify(pp, \sigma, m'', apk) = 1 \ (**').$$

Since $CheckValidConsensus(C) = 1$ holds and m' has been decided in epoch i of C and $d_{m'}$ is the required data associated with m', due to assumptions (I.5.), (I.4.) and (I.2.) we have that $d_{m'}$ contains correct proofs of possession for all keys in $\mathbf{pk_i}$ $(***)$.
We assume by contradiction that $S \cap Q_{pks}$ is non-empty with more than negligible probability. Since the following check (which is part of *LC.VerifyMisbehaviour*) passes:

$$S_{m''} \cap S_{m'} = S,$$

any $pk \in S$ is aggregated into *apk* and also into apk_S; this includes pk^*. Since the aggregate signature instantiation *AS* is unforgeable (see Definition 4.1 plus the assumption (S.1.)), due to $(**)$, $(**')$, $(***)$ and $(****)$ we have that, with more than negligible probability, both m' and m'' have been signed by the honest validator. However, this contradicts that $Incompatible(m', m'', d_{m'}) = 1$ which is ensured by assumption (B.2.). Hence, our assumption is false and $S \cap Q_{pks} = \emptyset$, so the probability defined in the accountability soundness property is indeed negligible.

□

L ROLLED OUT PROTOCOL $\mathscr{P}^h_{\mathsf{pa}}$ FOR RELATION $\mathcal{R}^{incl}_{\mathsf{pa},com}$

We give below the full rolled-out hybrid model protocol $\mathscr{P}^h_{\mathsf{pa}}$ for conditional NP relation $\mathcal{R}^{incl}_{\mathsf{pa},com}$. This is obtained by applying our two-steps compiler from Section F to polynomial protocol $\mathscr{P}_{\mathsf{pa}}$. In order to obtain the non-interactive version (i.e., the N from SNARK) we have additionally applied the Fiat-Shamir transform. In the following, by **transcript** at a certain point in time we denote the concatenation of the global constant, verification key, trusted public input, other public input and the proof elements created by the prover up to that point in time. $\mathcal{H}$ is a hash function, $\mathcal{H} : \{0,1\} \rightarrow \mathbb{F}$ and it emulates the random oracle. In the following, $\oplus$ is the addition operation on E_{inn} in affine coordinates. Note that in our implementation we instantiate E_{inn} with BLS12-377 [40] and E_{out} with BW6-761 [41], while we choose block to be 256 as this is the largest power of 2 smaller than the size of a field element in $\mathbb{F}$ (i.e., the base field for BLS12-377 which is the same as the scalar field of BW6-761). Finally, n has been defined as per Section 4.6.1, i.e., n is a

large enough power of 2; moreover, we let $v = n - 1$ and we let $N = n$. This, in turn, ensures that N has been chosen according to the properties stated in instantiation A.0.1, in particular when defining *AS.Setup*.

Public Parameters:
$(\mathbb{G}_{1,inn}, g_{1,inn}, \mathbb{G}_{2,inn}, g_{2,inn}, \mathbb{G}_{T,inn}, e_{inn}, H_{inn}, H_{PoP})$ from pp where $pp \leftarrow AS.Setup(aux_{AS} = n)$

Global constant: $h \in E_{inn} \setminus \mathbb{G}_{1,inn}$

Trusted Setup: $srs \leftarrow SNARK.Setup(aux_{SNARK} = (n, 3n-3))$, where $srs = ([1]_{1,out}, [\tau]_{1,out}, [\tau^2]_{1,out}, \ldots, [\tau^{3n-3}]_{1,out}, [1]_{2,out}, [\tau]_{2,out})$

Proving and Verifying Key Generation: $(srs_{pk}, srs_{vk}) \leftarrow SNARK.KeyGen(srs, \mathcal{R}^{incl}_{\text{pa},com})$,
where $(srs_{pk}, srs_{vk}) = (([1]_{1,out}, [\tau]_{1,out}, [\tau^2]_{1,out}, \ldots, [\tau^{3n-3}]_{1,out}), ([1]_{1,out}, [1]_{2,out}, [\tau]_{2,out}))$

Partial Input: $(x_1, state_2) \leftarrow SNARK.PartInput(srs, state_1, \mathcal{R}^{incl}_{\text{pa},com})$, where $(pk_0, \ldots, pk_{n-2})$ is part of $state_1$;
if $(pk_0, \ldots, pk_{n-2}) \notin \mathbb{G}^{n-1}_{1,inn}$, $SNARK.PartInput(srs, state_1, \mathcal{R}^{incl}_{\text{pa},com})$ outputs the empty string, otherwise *SNARK.PartInput* outputs $x_1 = ([pkx]_{1,out}, [pky]_{1,out})$ and $state_2 = state_1 \cup \{x_1\}$, where $\forall i \in \{0, \ldots, n-2\}$, pk_i as an element of the curve E_{inn} has the affine representation (pkx_i, pky_i). The polynomials $pkx(X)$ and $pky(X)$ are computed as $pkx(X) = \sum_{i=0}^{n-2} pkx_i \cdot L_i(X)$ and $pky(X) = \sum_{i=0}^{n-2} pky_i \cdot L_i(X)$ and finally, the polynomial commitments are computed as $[pkx]_{1,out} = pkx(\tau) \cdot [1]_{1,out}$ and $[pky]_{1,out} = pky(\tau) \cdot [1]_{1,out}$.

Public input: $x_1 = ([pkx]_{1,out}, [pky]_{1,out})$, $x_2 = ((b'_0, \ldots, b'_{\frac{n}{\text{block}}-1}), apk)$

Witness: $w = ((pk_0, \ldots, pk_{n-2}), (bit_0, \ldots, bit_{n-1}))$

Prover's Algorithm: $\pi \leftarrow SNARK.Prove(srs_{pk}, ((x_1, x_2), w), \mathcal{R}^{incl}_{\text{pa},com})$, where

Step 1:
Compute the affine representation $h = (h_x, h_y)$ and $apk \oplus h = ((apk \oplus h)_x, (apk \oplus h)_y)$.

Compute $\mathbf{pkx} = (pkx_0, \ldots, pkx_{n-2})$ and $\mathbf{pky} = (pky_0, \ldots, pky_{n-2})$ s. t. $\forall i \in \{0, \ldots, n-2\}$, pk_i as an element of the curve E_{inn} has the affine representation (pkx_i, pky_i).

Let $(kaccx_0, kaccy_0) = (h_x, h_y)$ and compute $(kaccx_{i+1}, kaccy_{i+1}) = (kaccx_i, kaccy_i) \oplus bit_i(pkx_i, pky_i)$, $\forall i < n-1$.

Compute polynomials

$$b(X) = \sum_{i=0}^{n-1} bit_i \cdot L_i(X),$$

$$kaccx(X) = \sum_{i=0}^{n-1} kaccx_i \cdot L_i(X),$$

$$kaccy(X) = \sum_{i=0}^{n-1} kaccy_i \cdot L_i(X),$$

$$pkx(X) = \sum_{i=0}^{n-2} pkx_i \cdot L_i(X),$$

$$pky(X) = \sum_{i=0}^{n-2} pky_i \cdot L_i(X).$$

Compute $[b]_{1,out} = b(\tau) \cdot [1]_{1,out}$, $[kaccx]_{1,out} = kaccx(\tau) \cdot [1]_{1,out}$, $[kaccy]_{1,out} = kaccy(\tau) \cdot [1]_{1,out}$.

The first output of the prover is $([b]_{1,out}, [kaccx]_{1,out}, [kaccy]_{1,out})$.

Step 2:
Compute the sum challenge $r = \mathcal{H}(\textbf{transcript})$.

Compute $sum = \sum_{j=0}^{\frac{n}{\text{block}}-1} b'_j r^j$.

Compute: $\frac{r}{2^{\text{block}-1}}, r^{\frac{n}{\text{block}}}$.

Compute polynomials

$$c(X) = \sum_{i=0}^{n-1} c_i \cdot L_i(X),$$

where $c_i = 2^{i \mod \text{block}} \cdot r^{i \div \text{block}}, 0 \leq i \leq n-1$.

$$acc(X) = \sum_{i=0}^{n-1} acc_i \cdot L_i(X),$$

where $acc_0 = 0$ and $acc_i = \sum_{j=0}^{i-1} bit_j \cdot c_j, 0 < i \leq n-1$.

$$aux(X) = \sum_{i=0}^{n-1} aux_i \cdot L_i(X),$$

where $aux_i = 1$ if i is divisible with block and $aux_i = 0$ otherwise, $\forall i < n$

Compute $[c]_{1,out} = c(\tau) \cdot [1]_{1,out}$, $[acc]_{1,out} = acc(\tau) \cdot [1]_{1,out}$.

The second output of the prover is $([c]_{1,out}, [acc]_{1,out})$.

Step 3:
Compute the quotient challenge $\alpha = \mathcal{H}(\textbf{transcript})$.

Compute the polynomial $t(X)$ of degree at most $3 \cdot n - 3$ where

$$\begin{aligned}
&t(X)(X^n - 1) = \\
&(X - \omega^{n-1}) \cdot [b(X) \cdot ((kaccx(X) - pkx(X))^2 \cdot (kaccx(X) + pkx(X) + kaccx(\omega \cdot X)) - (pky(X) - kaccy(X))^2) + \\
&+ (1 - b(X)) \cdot (kaccy(\omega \cdot X) - kaccy(X))] + \\
&+ \alpha(X - \omega^{n-1}) \cdot [b(X) \cdot ((kaccx(X) - pkx(X)) \cdot (kaccy(\omega \cdot X) + kaccy(X)) - (pky(X) - kaccy(X)) \cdot \\
&\cdot (kaccx(\omega \cdot X) - kaccx(X))) + (1 - b(X)) \cdot (kaccx(\omega \cdot X) - kaccx(X))] + \\
&+ \alpha^2 \cdot [b(X) \cdot (1 - b(X))] + \\
&+ \alpha^3 \cdot [c(\omega \cdot X) - c(X) \cdot (2 + (\frac{r}{2^{\text{block}-1}} - 2) \cdot aux(\omega \cdot X)) - (1 - r^{\frac{n}{\text{block}}}) \cdot L_{n-1}(X)] + \\
&+ \alpha^4 \cdot [(kaccx(X) - h_x) \cdot L_0(X) + (kaccx(X) - (h + apk)_x) \cdot L_{n-1}(X)] + \\
&+ \alpha^5 \cdot [(kaccy(X) - h_y) \cdot L_0(X) + (kaccy(X) - (h + apk)_y) \cdot L_{n-1}(X)] + \\
&+ \alpha^6 \cdot [acc(\omega \cdot X) - acc(X) - b(X) \cdot c(X) + sum \cdot L_{n-1}(X)] \ .
\end{aligned}$$

Compute $[t]_{1,out} = t(\tau) \cdot [1]_{1,out}$.

The third output of the prover is $[t]_{1,out}$.

Step 4:
Compute evaluation challenge $\zeta = \mathcal{H}(\textbf{transcript})$.

Compute evaluations: $\overline{pkx} = pkx(\zeta)$, $\overline{pky} = pky(\zeta)$, $\overline{b} = b(\zeta)$, $\overline{kaccx} = kaccx(\zeta)$, $\overline{kaccy} = kaccy(\zeta)$, $\overline{c} = c(\zeta)$, $\overline{acc} = acc(\zeta)$, $\overline{t} = t(\zeta)$.

Compute linearisation polynomial:

$$\begin{aligned} r(X) = &(\zeta - \omega^{n-1}) \cdot [\bar{b} \cdot (\overline{kaccx} - \overline{pkx})^2 \cdot kaccx(X) + (1 - \bar{b}) \cdot kaccy(X)] + \\ &+ \alpha \cdot (\zeta - \omega^{n-1}) \cdot [\bar{b} \cdot ((\overline{kaccx} - \overline{pkx}) \cdot kaccy(X) - (\overline{pky} - \overline{kaccy}) \cdot kaccx(X)) + (1 - \bar{b}) \cdot kaccx(X)] + \\ &+ \alpha^3 \cdot c(X) + \\ &+ \alpha^6 \cdot acc(X). \end{aligned}$$

Compute evaluation of linearisation polynomial $\overline{r_\omega} = r(\omega \cdot \zeta)$.

The fourth output of the prover is $(\overline{pkx}, \overline{pky}, \bar{b}, \overline{kaccx}, \overline{kaccy}, \bar{c}, \overline{acc}, \overline{r_\omega})$.

Step 5:
Compute opening challenge $\nu = \mathcal{H}(\textbf{transcript})$.

Compute first opening proof polynomial

$$\begin{aligned} W_\zeta(X) = \frac{1}{X - \zeta}(&t(X) - \bar{t} + \\ &+ \nu(pkx(X) - \overline{pkx}) + \\ &+ \nu^2(\cdot pky(X) - \overline{pky}) + \\ &+ \nu^3(b(X) - \bar{b}) + \\ &+ \nu^4(kaccx(X) - \overline{kaccx}) + \\ &+ \nu^5(kaccy(X) - \overline{kaccy}) + \\ &+ \nu^6(c(X) - \bar{c}) + \\ &+ \nu^7(acc(X) - \overline{acc})) \end{aligned}$$

and second opening proof polynomial

$$W_{\zeta \cdot \omega}(X) = \frac{1}{X - \zeta \cdot \omega}(r(X) - \overline{r_\omega}).$$

Compute $[W_\zeta]_{1,out} = W_\zeta(\tau) \cdot [1]_{1,out}$ and $[W_{\zeta \cdot \omega}]_{1,out} = W_{\zeta \cdot \omega}(\tau) \cdot [1]_{1,out}$.

The fifth output of the prover is $([W_\zeta]_{1,out}, [W_{\zeta \cdot \omega}]_{1,out})$.

Compute the multipoint evaluation challenge $u = \mathcal{H}(transcript)$.

Return $\pi = ([b]_{1,out}, [kaccx]_{1,out}, [kaccy]_{1,out}, [c]_{1,out}, [acc]_{1,out}, [t]_{1,out}, [W_\zeta]_{1,out}, [W_{\zeta \cdot \omega}]_{1,out}, \overline{pkx}, \overline{pky}, \bar{b}, \overline{kaccx}, \overline{kaccy}, \bar{c}, \overline{acc}, \bar{r}_\omega)$

Verifier's Algorithm: $0/1 \leftarrow SNARK.Verify(srs_{vk}, (x_1, x_2), \pi, \mathcal{R}^{incl}_{\text{pa},com})$, where

Step 1:
Compute the affine representation $h = (h_x, h_y)$ and $apk \oplus h = ((apk \oplus h)_x, (apk \oplus h)_y)$.

Step 2:
Validate proof elements $([b]_{1,out}, [kaccx]_{1,out}, [kaccy]_{1,out}, [c]_{1,out}, [acc]_{1,out}, [t]_{1,out}, [W_\zeta]_{1,out}, [W_{\zeta \cdot \omega}]_{1,out}) \in \mathbb{G}^8_{1,out}$.

Step 3:
Validate proof elements $(\overline{pkx}, \overline{pky}, \bar{b}, \overline{kaccx}, \overline{kaccy}, \bar{c}, \overline{acc}, \bar{r}_\omega) \in \mathbb{F}^8$.

Step 4:
Compute challenges $(r, \alpha, \zeta, \nu, u)$ as in the prover $P^{SNARK}_{pa,com}$ description from the common input, trusted public input, public input and respective necessary parts of the **transcript** using elements of π_{pa}.

Step 5:
Compute: $sum = \sum_{j=0}^{\frac{n}{\text{block}}-1} b'_j r^j$.

Compute: $\frac{r}{2^{\text{block}-1}}, r^{\frac{n}{\text{block}}}$.

Step 6:
Compute polynomial evaluations $\zeta^n - 1$ and $\overline{aux}_\omega = aux(\omega \cdot \zeta)$[6] and Lagrange basis polynomials $L_0(\zeta) = \frac{\zeta^n - 1}{n \cdot (\zeta - 1)}$ and $L_{n-1}(\zeta) = \frac{(\zeta^n - 1) \cdot \omega^{n-1}}{n \cdot (\zeta - \omega^{n-1})}$.

Step 7[7]:
Compute quotient polynomial evaluation

$$\bar{t} = \frac{\overline{r_\omega} + [\bar{b}((\overline{kaccx} - \overline{pkx})^2 \cdot (\overline{kaccx} + \overline{pkx}) - (\overline{pky} - \overline{kaccy})^2) - (1 - \bar{b}) \cdot \overline{kaccy}] \cdot (\zeta - \omega^{n-1})}{\zeta^n - 1} +$$

$$+ \frac{\alpha \cdot [\bar{b} \cdot ((\overline{kaccx} - \overline{pkx}) \cdot \overline{kaccy} + (\overline{pky} - \overline{kaccy}) \cdot \overline{kaccx}) - (1 - \bar{b}) \cdot \overline{kaccx}] \cdot (\zeta - \omega^{n-1})}{\zeta^n - 1} +$$

$$+ \frac{\alpha^2 \cdot \bar{b} \cdot (1 - \bar{b})}{\zeta^n - 1} +$$

$$- \frac{\alpha^3 \cdot [(1 - r^{\frac{n}{\text{block}}}) \cdot L_{n-1}(\zeta)]}{\zeta^n - 1} - \alpha^3 \cdot \bar{c} \cdot (2 + (\frac{r}{2^{\text{block}-1}} - 2)) \cdot \overline{aux}_\omega +$$

$$+ \frac{\alpha^4 \cdot [(\overline{kaccx} - h_x) \cdot L_0(\zeta) + (\overline{kaccx} - (h + apk)_x) \cdot L_{n-1}(\zeta)]}{\zeta^n - 1} +$$

$$+ \frac{\alpha^5 \cdot [(\overline{kaccy} - h_y) \cdot L_0(\zeta) + (\overline{kaccy} - (h + apk)_y) \cdot L_{n-1}(\zeta)]}{\zeta^n - 1} +$$

$$+ \frac{\alpha^6 \cdot [-\overline{acc} - \bar{b} \cdot \bar{c} + sum \cdot L_{n-1}(\zeta)]}{\zeta^n - 1}.$$

Step 8:
Compute full batched polynomial commitment $[F]_{1,out}$.

$$\begin{aligned}
[F]_{1,out} =& [t]_{1,out} + v \cdot [pkx]_{1,out} + v^2 \cdot [pky]_{1,out} + v^3 \cdot [b]_{1,out} + \\
&+ (u \cdot (\zeta - \omega^{n-1}) \cdot (\bar{b} \cdot ((\overline{kaccx} - \overline{pkx})^2 + \alpha \cdot (\overline{pky} - \overline{kaccy})) + \alpha \cdot (1 - \bar{b})) + v^4) \cdot [kaccx]_{1,out} + \\
&+ (u \cdot (\zeta - \omega^{n-1})(\alpha \cdot \bar{b}(\overline{kaccx} - \overline{pkx}) + (1 - \bar{b})) + v^5) \cdot [kaccy]_{1,out} + \\
&+ (u \cdot \alpha^3 + v^6) \cdot [c]_{1,out} + \\
&+ (u \cdot \alpha^6 + v^7) \cdot [acc]_{1,out}.
\end{aligned}$$

Step 9:
Compute group-encoded batch evaluation $[E]_{1,out}$

$$[E]_{1,out} = (\bar{t} + v \cdot \overline{pkx} + v^2 \cdot \overline{pky} + v^3 \cdot \bar{b} + v^4 \cdot \overline{kaccx} + v^5 \cdot \overline{kaccy} + v^6 \cdot \bar{c} + v^7 \cdot \overline{acc} + u \cdot \overline{r_w}) \cdot [1]_{1,out}$$

Step 10:
Batch validate all evaluations by checking that the following holds

$$e_{out}([W_\zeta]_{1,out} + u \cdot [W_{\zeta \cdot \omega}]_{1,out}, [\tau]_{2,out}) = e_{out}(\zeta \cdot [W_\zeta]_{1,out} + u \cdot \zeta \cdot \omega \cdot [W_{\zeta \cdot \omega}]_{1,out} + [F]_{1,out} - [E]_{1,out}, [1]_{2,out}).$$

[6] We have $aux(\omega \cdot \zeta) = 1$ if $(\omega \cdot \zeta)^{\frac{n}{\text{block}}} = 1$ and $aux(\omega \cdot \zeta) = \frac{1}{\text{block}} \cdot \frac{\zeta^n - 1}{(\omega \cdot \zeta)^{\frac{n}{\text{block}}} - 1}$ otherwise.

[7] This step can be optimised in obvious ways in order to reduce the number of field operations necessary to compute $\bar{t}$. We choose to include the non-compact formula in this write-up such that the reader is able to follow the linearisation process to a larger extent than via a more compact formula.

M IS OUR PROTOCOL APPLICABLE TO ETHEREUM?

We believe that our protocol could not feasibly be directly applied to Ethereum as it is without a hard fork, but it would be easy to apply it with changes that might feasibly be implemented, even with changes not specifically designed with our protocol in mind. Ethereum already uses BLS signatures on the BLS12-381 curve in consensus. To work with BLS12-381, Our protocol would use KZG commitments on the BW6-767 curve[55]. Because the base field of BL12-381 is not very 2-adic, a prover would need a more complicated FFT algorithm but this is feasible [55]. We also need an appropriately sized subgroup of the multiplicative field to use for our Lagrange basis commitments. An easy calculation gives the small prime factors $2, 3^2, 11, 23, 47, 10177$ and 859267 for the order of the multiplicative group and any product of these larger than the number of validators gives a usable subgroup.

Next we consider who constructs the the KZG commitment to the validators public keys. For the shortest light client proofs validators would construct and sign this commitment, which would require a change to the consensus logic. As an alternative, a smart contract could compute the commitments on chain. This requires the EVM to have access to the active validator's public keys and would also require a precompile for BW6-767 operations to be feasible. We note that there have been many suggestions for adding elliptic curve operations for different curves to Ethereum (e.g. EIPs 2537,2538,3026[[56]) but few have been implemented so far. We would expect this to be the bottleneck for implementing our protocol on Ethereum.

Finally we compare running our scheme on the full validator set to Ethereum's current light client design[57]. That uses s subset of 1024 public keys that changes every epoch (i.e. 64 blocks or 12.8 minutes). It is not accountable because it would take less than 1024 validators misbehaving to deceive a light client. We remark that 1024 384-bit public keys is comparable in size to 1 bit for all of Ethereum's over 500000 validators, and as a result our light client scheme can be used for an accountable light client with a similar overhead to Ethereum's existing unaccountable scheme.

With epochless Casper FFG [58], 64 aggregated signatures are required to represent a single Casper FFG vote, 2 of which are required for a proof. The form of accountable safety satisfied by Casper FFG [18] suffices for us: if two forks are finalised, then then 2/3 of the validator set voted on two messages such that signing both is punishable. One complication of a Casper FFG light client is that valid votes include two blockhashes, one of which is required to be the ancestor of the other. There needs to be an efficient "proof of ancestry" such as introducing a more efficient commitment to previous block hashes, e.g. Merkle Mountain Range of blockhashes as suggested in [59].