Shielded CSV: Private and Efficient Client-Side Validation

Jonas Nick[1], Liam Eagen[2], and Robin Linus[3]

[1]Blockstream
[2]Alpen Labs
[3]ZeroSync

September 20, 2024

Abstract

Cryptocurrencies allow mutually distrusting users to transact monetary value over the internet without relying on a trusted third party.

Bitcoin, the first cryptocurrency, achieved this through a novel protocol used to establish consensus about an ordered transaction history. This requires every transaction to be broadcasted and verified by the network, incurring communication and computational costs. Furthermore, transactions are visible to all nodes of the network, eroding privacy, and are recorded permanently, contributing to increasing storage requirements over time. To limit resource usage of the network, Bitcoin currently supports an average of 11 transactions per second.

Most cryptocurrencies today still operate in a substantially similar manner. Private cryptocurrencies like Zcash and Monero address the privacy issue by replacing transactions with proofs of transaction validity. However, this enhanced privacy comes at the cost of increased communication, storage, and computational requirements.

Client-Side Validation (CSV) is a paradigm that addresses these issues by removing transaction validation from the blockchain consensus rules. This approach allows sending the coin along with a validity proof directly to its recipient, reducing communication, computation and storage cost. CSV protocols deployed on Bitcoin today [1, 2] do not fully leverage the paradigm's potential, as they still necessitate the overhead of publishing ordinary Bitcoin transactions. Moreover, the size of their coin proofs is proportional to the coin's transaction history, and provide limited privacy. A recent improvement is the Intmax2 [3] CSV protocol, which writes significantly less data to the blockchain compared to a blockchain transaction and has succinct coin proofs.

In this work, we introduce Shielded CSV, which improves upon state-of-the-art CSV protocols by providing the first construction that offers truly private transactions. It addresses the issues of traditional private cryptocurrency designs by requiring only 64 bytes of data per transaction, called a *nullifier*, to be written to the blockchain. Moreover, for each nullifier in the blockchain, Shielded CSV users only need to perform a single Schnorr signature verification, while non-users can simply ignore this data. The size and verification cost of coin proofs for Shielded CSV receivers is independent of the transaction history. Thus, one application of Shielded CSV is adding privacy to Bitcoin at a rate of 100 transactions per second, provided there is an adequate bridging mechanism to the blockchain.

We specify Shielded CSV using the Proof Carrying Data (PCD) abstraction. We then discuss two implementation strategies that we believe to be practical, based on Folding Schemes and Recursive STARKs, respectively. Finally, we propose future extensions, demonstrating the power of the PCD abstraction and the extensibility of Shielded CSV. This highlights the significant potential for further improvements to the Shielded CSV framework and protocols built upon it.

Contents

1 Introduction

Cryptocurrencies and Blockchains Bitcoin, the first practical cryptocurrency, was created by Satoshi Nakamoto as a means of enabling censorship-resistant online payments that eliminate the need for a trusted intermediary. In Bitcoin, coins are typically associated with a public key, and knowledge of the corresponding private key confers ownership of the coins. Using this private key, the owner can cryptographically sign a transaction to transfer the coins to another participant. After preparing the transaction, the owner broadcasts the signed transaction to the Bitcoin network. Nodes in the Bitcoin network verify the transaction, which includes confirming the existence of the coins being spent and the validity of the signature. In particular, nodes ensure that there is only one transaction spending the same coins; otherwise, it is considered a "double-spend" attempt by the owner. If the transaction is accepted and includes a sufficient fee, it is included in the blockchain, and the recipient can spend the received coins. However, Bitcoin transactions can potentially reveal information about the parties involved, which compromises the privacy of its users.

Many other blockchains and associated cryptocurrencies have emerged since Bitcoin. While certain aspects of the protocols vary dramatically, the high-level structure described here has largely remained the same. Some protocols, such as Ethereum, enrich the notion of ownership to support arbitrary computation inside "smart contracts," while other protocols, like Zcash [4], publish zero-knowledge proofs of transactions instead of transactions themselves. Transactions in these protocols are often larger and more expensive to verify compared to Bitcoin's transactions.

Despite these variations, the fundamental principle remains unchanged: transferring ownership requires signing a transaction and broadcasting it to the peer-to-peer network. Valid transactions with sufficient fees are included in blocks, which are then propagated to nodes across the network. Before extending their local copy of the blockchain, all nodes must verify the validity of new blocks. This includes checking every transaction by verifying signatures, zero-knowledge proofs, or smart contract executions. Blocks containing invalid transactions are rejected.

To ensure that participants with restricted computational resources can verify the protocol rules, a limit on transaction throughput is necessary. Bitcoin, for instance, can process an average of 11 transactions per second, which poses a limitation for widespread adoption.

Client-Side Validation Cryptocurrencies do not necessarily require broadcasting transactions nor the validation of all transactions, an insight first published by Todd [5]. This approach, known as *client-side validation* (CSV), fundamentally separates transaction validation from block validation.

In contrast to traditional cryptocurrencies, where blocks contain transactions that are validated by all nodes, the CSV model allows blocks to contain arbitrary data. Individual nodes then interpret this data "client-side" (for example, as transactions) and do not reject blocks containing data they consider invalid. The blockchain serves solely to establish consensus on the order of published data to prevent double spending.

Consider, for example, that the data contained in a block is interpreted client-side as a list of transactions. To make a payment, the sender publishes the transaction on the blockchain. If the receiver deems the transaction invalid, they do not reject the block containing the transaction; instead, they simply do not accept the payment. One can see that, while the blockchain can support multiple client-side validated protocols, it is essential that the sender and receiver use the same protocol.

Because in the CSV model transaction validity rules are irrelevant for establishing consensus on the blockchain, it is unnecessary to post full transactions to the blockchain. The blockchain's primary function is to prevent double-spending, for which the transaction details are unnecessary. Therefore, senders of CSV payments write a piece of data derived from the transaction to the blockchain, which we refer to as a *nullifier*.

As a result, senders only need to broadcast nullifiers, rather than entire transactions. To make a payment, the sender provides the coin directly to the receiver, along with any proof that the receiver requires to verify the validity of the coin. The receiver verifies the coin and ensures that the correct nullifier is present in the blockchain to prevent double-spending.

The specific implementation of nullifiers varies across CSV protocols. Contemporary CSV protocols on Bitcoin make use of ordinary Bitcoin transactions and the existing protocol rules for spending coins [1, 2]. In these cases, the nullifier is effectively a Bitcoin transaction. While client-side payments in those protocols require making a Bitcoin transaction, they allow for the transfer of non-Bitcoin assets and the implementation of more expressive spending policies than what is possible in native Bitcoin. In contrast, recent CSV protocols [3, 6] and Shielded CSV employ a different approach. Here, the nullifier is smaller than a blockchain transaction and is not interpreted by existing blockchain consensus rules.

In order to check the validity of a coin, receivers typically need to check more than just a single transaction before accepting a payment. Each coin has a transaction history comprising all ancestor transactions that led to its creation. A received coin is invalid if there exists a transaction in the coin's history that is invalid, for example, because it produces more coins than it consumes or because a coin has not been nullified correctly. To allow the receiver to verify the validity of a coin, every coin in a client-side validation protocol has an associated *coin proof*.

In its simplest form, a coin proof consists of the coin's entire transaction history. Given that transactions commonly have multiple parent transactions, the history of a single coin is likely to encompass a substantial portion of all transactions within the system. Consequently, this approach offers limited advantage over broadcast-based systems in terms of communication efficiency.

Privacy Traditional client-side validation protocols, where the coin proof reveals the transaction history of the coin, offer certain privacy advantages over transparent blockchain transactions. However, while blockchain observers do not obtain the history of a CSV coin, coin recipients receiving the coin proof still do. As demonstrated for transparent blockchains, the transaction history is rich source of information that allows linking transactions and de-anonymizing users [7].

We consider a CSV protocol private if it discloses no information to the recipient beyond the coin's validity and, in particular, does not reveal the transaction history.

Publishing Nullifiers Depending on how nullifiers are published on the blockchain, we can distinguish between permissioned and permissionless CSV protocols. In permissioned CSV protocols, users rely on the honesty of an entity operating the protocol in order to transfer coins. That entity is responsible for publishing nullifiers on the blockchain and may maintain additional state of the protocol.

Conversely, permissionless CSV protocols rely on the blockchain as a censorship-resistant bulletin-board that makes sure the data necessary to create transactions is available. To minimize the cost of posting nullifiers to the blockchain, permissionless CSV can utilize specialized participants called *publishers*, a role open to any protocol user. Publishers collect nullifiers from users, aggregate them when possible, and post them to the blockchain. Since publishers pay transaction fees on the blockchain, they typically charge users for their service. Without publishers, users would have to incur the overhead of making dedicated transactions on the blockchain to post a nullifier. Not only does this waste blockchain space, but it also requires users to possess the blockchain's native currency, rather than allowing payment to publishers using the CSV protocol.

Bridging Using the native currency of the blockchain in a CSV protocol necessitates a bridging protocol that allows the transfer between the two systems. Such a bridge can be instantiated by a group of entities entrusted to honestly follow the bridging protocol.

A trustless bridging protocol, which does not rely on the honesty of individual entities, requires that the CSV protocol's rules be verifiable on the blockchain. In principle, sending a CSV transaction to the blockchain to withdraw from the CSV protocol is not significantly different from sending it to any other CSV recipient. To verify the CSV protocol, the blockchain requires either a sufficiently expressive blockchain programming language or a dedicated change to the blockchain's protocol rules. Nevertheless, the substantial coin proofs required by traditional CSV protocols, which would have to be published to the blockchain and verified by all protocol nodes, erase the advantages of client-side validation. For bridging to be viable, CSV protocols must have a way to compress coin proofs significantly.

In the case of Bitcoin, the blockchain programming language currently lacks the expressiveness to verify the CSV protocol, and proposing changes to Bitcoin's protocol rules is a lengthy process with a very uncertain outcome. Nevertheless, there is significant potential in building a bridge using the BitVM [8] paradigm, which would not necessitate changes to the protocol rules. BitVM enables the verification of any computation on Bitcoin, assuming that fraudulent computation can be challenged on the blockchain.

Apart from full-fledged bridging, there are innovative methods to derive an asset on the CSV protocol from the blockchain's native currency. One such approach is the "one-way peg", which involves burning coins on the blockchain and subsequently minting equivalent tokens on the CSV protocol.

1.1 Contributions

We present Shielded CSV, the first client-side validation protocol that is private.

Privacy Coin proofs reveal no information other than the validity of the coin and its creation time.

Short Coin Proofs The size of a coin proof is independent of its transaction history.

Blockchain Efficiency Each transaction requires only 64 bytes of data to be written to the blockchain, plus a small constant overhead, regardless of the transaction size.

Permissionless Shielded CSV does not rely on any trusted party for transaction execution. All necessary data is directly written to, and retrieved from, the blockchain.

Efficient Instantiability Shielded CSV can be instantiated with existing cryptographic primitives. Coin proofs are designed to be proven and verified efficiently in zero-knowledge.

Trustless Publishing Publishers are guaranteed to receive a transaction fee when they are the first to post a nullifier to the blockchain. Anyone has the opportunity to take on the role of a publisher.

Prunable Wallet State Shielded CSV wallets are not required to retain information about all received coins and past transactions. This reduces the wallet's storage requirements and prevents potential adversaries from learning about previously received coins and historical transaction data in the event of a wallet compromise.

Advanced Spending Policies Shielded CSV natively supports "t-of-n" shared ownership, which requires signatures from t out of n account owners to spend a coin. With the addition of time-locked transactions, Shielded CSV supports atomic swaps with Bitcoin.

Shielded CSV offers substantial advantages over private cryptocurrencies. They require the validation of all transactions, which contain relatively large and computationally expensive zero-knowledge proofs. In contrast, Shielded CSV only requires the receiver of a coin to download and verify the coin proof, resulting in substantial reductions in computational and bandwidth costs. Regarding security, Shielded CSV derives its resistance to double-spending from the parent blockchain, eliminating the need for its own consensus mechanism. Furthermore, Shielded CSV conceals the transaction history as effectively as private cryptocurrencies, and in some cases, provides enhanced privacy. For instance, unlike systems such as Zcash, Shielded CSV hides the number of transaction outputs from the receiver.

Shielded CSV is a promising candidate for significantly enhancing the privacy of existing cryptocurrencies, such as Bitcoin. It provides efficient coin proofs and data availability, allowing the development of bridges that transfer assets between Shielded CSV and the blockchain, for example, using the BitVM design. Given the 64-byte space requirement on the blockchain, the current Bitcoin block size limit supports approximately 100 Shielded CSV transactions per second (irrespective of the number of coins spent and created in each transaction). For Bitcoin nodes not using Shielded CSV, nullifiers appear as raw blockchain data, requiring no signature verifications and having no impact on the set of unspent transaction outputs.

1.2 Related Work

Client-side validation was first proposed by Todd in 2013 [5]. One of the suggested applications was to enable the expression of more advanced spending policies ("smart contracts") than those possible with Bitcoin's native language, Bitcoin Script [9], while simultaneously minimizing the computational cost for protocol participants.

CSV achieves this by requiring only minimal data to be published on the blockchain to prevent double spending and transmitting additional proofs necessary for the payment directly to the recipients. Although implementing CSV to extend Bitcoin's functionality requires either bridging protocols or significant changes to Bitcoin's protocol rules, it can be used to issue custom assets on the Bitcoin blockchain without such additional requirements. Indeed, the ability to issue custom assets has emerged as a primary application of CSV protocols inspired by Todd's ideas.

In subsequent years, Orlovsky, Todd, Zucco, Tenga, and Ukolova proposed the RGB protocol [1], with implementation beginning in 2019. RGB builds upon the original CSV proposals and incorporates confidential transactions [10], which hides the amounts and types of assets being transferred, although it does not conceal the transaction history.

In 2022, Lightning Labs published Taproot Assets [2], a CSV protocol that, while not supporting confidential transactions, focuses on integration with the Lightning Network [11] to enable cheap and instant transactions.

Table 1: Comparison of Shielded CSV with other client-side validation protocols. In the Privacy column, "No" indicates that the recipient obtains the plain transaction history. Protocols are "LN-compatible" when CSV assets can be transferred through Lightning Network channels, obscuring the transaction history. The column "Coin Proof Size" shows the size of a coin proof. $|\mathsf{tx\_history}|$ refers to the number of ancestor transactions that led to the creation of the coin. The column "Blockchain Space" indicates the space required per CSV protocol transaction. $\mathsf{BitcoinTx}(\mathsf{num\_ins}, \mathsf{num\_outs})$ refers to the size of a Bitcoin transaction with $\mathsf{num\_ins}$ inputs spending Taproot outputs and $\mathsf{num\_outs}$ Taproot outputs, which is $12 + 107 \cdot \mathsf{num\_ins} + 43 \cdot \mathsf{num\_outs}$ bytes (for $\mathsf{num\_ins}, \mathsf{num\_outs} \leq 252$). The space required on the Bitcoin blockchain for a RGB or Taproot Asset CSV transaction with $\mathsf{num\_ins}$ inputs and $\mathsf{num\_outs}$ outputs is $\mathsf{BitcoinTx}(\mathsf{num\_ins}, \mathsf{num\_outs})$. Intmax2 requires a 1.5-round interactive protocol with the publisher.

	Privacy	Coin Proof Size	Blockchain Space
RGB	Encrypted amounts & assets, LN-compatible	$O(\lvert\mathsf{tx\_history}\rvert)$	$\mathsf{BitcoinTx}(\mathsf{num\_ins}, \mathsf{num\_outs})$
Taproot Assets	No, LN-compatible	$O(\lvert\mathsf{tx\_history}\rvert)$	$\mathsf{BitcoinTx}(\mathsf{num\_ins}, \mathsf{num\_outs})$
Intmax2	No	$O(1)$	Asymptotic to 4 to 5 bytes (interactive)
Shielded CSV	Yes	$O(1)$	Asymptotic to 64 bytes (non-interactive)

Both RGB and Taproot Assets share a common issue: the coin proof comprises the entire transaction history of the coin. This quickly becomes unwieldy, as each transaction typically has multiple ancestors, causing proof sizes to grow rapidly over time. However, recent advancements in succinct non-interactive arguments of knowledge (SNARKs) allow compression of coin proofs to a modest size, fundamentally improving the practicality of CSV. Todd had mentioned this approach in 2015 [12], but no practical SNARK existed at that time.

In early 2023, Linus reintroduced the idea with zkCoins [6], the first proposal combining CSV with a zero-knowledge SNARK (zk-SNARK), improving both scalability and privacy of previous protocols. Moreover, zkCoins proposes an alternative to using Bitcoin transactions as nullifiers, diverging from the approach used in RGB and Taproot assets. Instead, zkCoins nullifiers only consist of "about 64 bytes" that are written to the blockchain and are neither verified by regular blockchain users nor affect the set of unspent transaction outputs. In contrast to RGB, which only encrypts amounts and assets, zkCoins also hides the transaction history. While the ideas suggested in zkCoins represent significant progress, the proposal does not describe the complete protocol and lacks some important details.

In late 2023, Rybakken, Hioki, and Yaksetig published Intmax2 [3], a ZK-rollup that shifts data and computation to the client and requires posting nullifiers to the blockchain. As a result, it can also be characterized as a permissionless CSV protocol. Intmax2 utilizes SNARKs to obtain succinct coin proofs, but does not provide privacy. However, it offers extremely efficient nullifier sizes, requiring only 4 to 5 bytes per transaction. This efficiency is achieved by identifying user accounts with short integers and using an interactive protocol between publisher and sender.

See Table 1 for a comparison of Shielded CSV to other CSV protocols.

2 Technical Overview

This section provides an overview of Shielded CSV by building it up step-by-step from a simpler protocol. The starting point is a toy client-side validation protocol with transactions similar to Bitcoin transactions, i.e., they have outputs and inputs that spend outputs of previous transactions.

We refer to transaction outputs as *coins* to clearly distinguish them from other types of outputs we encounter in Shielded CSV. In contrast to Bitcoin's transaction outputs, for now coins only consist of an amount and do not include a public key, implying that anyone can spend any coin in the toy protocol. This issue will be addressed later.

A coin proof in our basic protocol is simply the transaction history of the coin, i.e., the entire directed acyclic graph of transactions that connects the coin to some issuance transactions. We do not discuss issuance transactions in detail as their specifics depend on the scenario in which Shielded CSV is used; we just assume that there is an existing supply of coins. To verify the coin proof, the receiver checks that all transactions in the graph are valid, e.g., that non-issuance transactions do not create more coins than they spend.

Double Spending Thus far, our protocol lacks a mechanism to prevent double spending. To address this issue, we require that all coins spent in a transaction are "nullified" by publishing a corresponding nullifier on the blockchain. We observe that each coin can be uniquely identified by the hash of the transaction that created it and its index in the transaction's output list. Therefore, before giving the coin and coin proof to the receiver, senders post a nullifier consisting of the coin identifier and the hash of the transaction to the blockchain.

$$\texttt{Nullifier} := (\texttt{Coin Identifier}, \texttt{Transaction Hash})$$

Shielded CSV users maintain a data structure we call *nullifier accumulator* that stores all nullifiers that have been published so far. When users receive a new block, they scan for nullifiers and insert every encountered nullifier into the accumulator if the nullifier's coin identifier has not been inserted previously. During a blockchain reorganization, nullifiers in blocks that are absent in the new chain are removed from the accumulator. For non-users of Shielded CSV, nullifiers are just arbitrary data in the blockchain.

To actually prevent double spending, we add the following rule to coin proof validation: for every coin spent in the transaction graph, its corresponding coin identifier must be present in the accumulator and the stored transaction hash must match the transaction that spends the coin. Should a sender attempt to double spend by publishing a nullifier with the same coin identifier and a different transaction hash, the receiver's nullifier accumulator would only contain the original transaction hash and therefore, validation of the coin proof would fail.

Accounts One undesirable property of the design so far is that the sender has to publish a nullifier for every coin spent in a transaction. To improve upon this, we introduce the concept of accounts, each uniquely identified by an account ID. Users generate a new account by generating a Schnorr key pair, where the public key serves as the account ID. Accounts allow that senders only nullify an account state instead of all spent coins. Each account has a single associated account state that is not nullified.

An account state consists of the account ID, balance, nullifier public key, and spent accumulator value. The nullifier public key is a Schnorr signature public key that is used to nullify this account state. Therefore, we change our definition of nullifiers: instead of coin identifiers, senders post nullifier public keys to the blockchain instead. The spent accumulator contains the coins this account has already spent and allows to enforce that an account can only spend a coin once. The spent accumulator *value*, which is contained in the account state, is a commitment to the spent accumulator.

$$\texttt{Nullifier} := (\texttt{Nullifier Public Key}, \texttt{Transaction Hash})$$

Account states are updated in transactions. Thus, we require that every transaction has a special account state output and "spends" an account state output, except when the transaction creates a new account.

$$\texttt{Transaction} := ((\texttt{Previous State}, \texttt{Previous Coins}), (\texttt{New State}, \texttt{New Coins}))$$

In order to create a new transaction, the sender first collects coins to spend and to create. They then create a new account state, with the same account ID and updated balance. They also generate a fresh elliptic curve key pair, whose public part becomes the nullifier public key of the new account state.

To obtain the new spent accumulator value, senders insert the coins they spend into the account's spent accumulator and compute the commitment to the new accumulator. Verifiers check every transaction in the coin proof follows the account state update rules: the account ID stays the same, and the balance and spent accumulator were updated correctly. Additionally, they consult their nullifier accumulator to check that the previous account state was correctly nullified with the nullifier public key and the transaction hash.

Coins and Addresses At this point of the protocol description, coins only consist of an amount and can be spent by anyone. To address that, we mandate that coins contain a hiding commitment to the account ID of the receiver. Furthermore, coin proofs must include the opening of these commitments for all spent coins to prove that their account ID matches the account ID of the state update.

Consequently, to create a coin that only the receiver's account can spend, the sender obtains a hiding commitment to the receiver's account ID. These commitments therefore effectively serve as user addresses in Shielded CSV.

Signatures The protocol developed up to this point has a major problem: should someone obtain an account's nullifier public key, they can update the account state and make transactions on their behalf. To prevent this issue, we add a Schnorr signature for the nullifier public key over the transaction hash to the nullifier. Users only insert a nullifier into their nullifier accumulator if the Schnorr signature is valid for the nullifier public key. As such, creating a valid transaction requires knowledge of the nullifier secret key.

It is possible to save blockchain space by removing the transaction hash from the nullifier. Instead, we commit to the transaction hash with the signature, using a technique known as *sign-to-contract*:

Recall that for a Schnorr signature (R, s) we have $sG = R + \mathsf{H}_{\mathsf{Sig}}(R, P, \mathsf{m})P$ where G is the generator of a group, P is the public key, and $\mathsf{H}_{\mathsf{Sig}}$ is a hash function. To commit to transaction hash h in the signature (R, s), the sender draws random scalar k, computes $R' = kG$, $R = R' + \mathsf{H}_{\mathsf{SigComm}}(R', h)G$, and s such that $sG = R + \mathsf{H}_{\mathsf{Sig}}(R, P, \mathsf{m})G$ where m is some fixed string, for example, "Shielded CSV: state update". As a result, users no longer insert transaction hashes into the accumulator, only commitments R to the transaction hash. Coin proofs must contain an opening of every transaction's commitment to prove that the commitments in the accumulator commit to the correct transaction hash.

$$\texttt{Nullifier} := (\texttt{Nullifier Public Key}, \texttt{Signature})$$

Publishers To post a nullifier to the blockchain, users can create a dedicated blockchain transaction. However, a dedicated transaction for a single nullifier creates significant blockchain space overhead. Therefore, it is more efficient to send nullifiers to an entity who collects nullifiers and posts them all at once to the blockchain. We call this entity *publisher*, a role that is open to any protocol participant. The publisher will typically demand a fee payment as compensation for the transaction cost on the blockchain. The fee mechanism will be elaborated later in this section.

Signature Aggregation To minimize the size of the nullifiers, we half-aggregate their signatures. Schnorr signature half-aggregation [13] allows to non-interactively aggregate n Schnorr signatures into a signature $(R_1, \ldots, R_n, s)$ that is about half the size of the individual signatures. To make use of half-aggregation, we stop posting individual nullifiers to the blockchain and post *aggregate nullifiers* instead. Aggregate nullifiers consist of a list of nullifier public keys and a Schnorr half-aggregate signature.

$$\texttt{Aggregate Nullifier} := (\texttt{Nullifier Public Keys}, \texttt{Aggregate Signature})$$

Half-aggregation is compatible with sign-to-contract: After aggregation, R_i remains the transaction commitment corresponding to the i-th nullifier public key. Thus, when encountering an aggregate nullifier in a block, users iterate through every nullifier public key and if it is not present in the accumulator already, they insert it along with the corresponding transaction commitment R_i.

Publishers typically collect nullifiers and create an aggregate nullifier by half-aggregating the nullifiers' Schnorr signatures. To clarify, aggregate nullifiers are allowed to consist of only a single nullifier public key and the half-aggregate of a single signature.

As a result, the blockchain space necessary to nullify n accounts is n Schnorr public keys, n group elements for the signature, and a constant overhead, comprising the s part of the signature and the remainder of the blockchain transaction. For a 256-bit elliptic curve, the space used per Shielded CSV transaction approaches 64 bytes, independent of the transaction size.

Interim Summary See Fig. 1 for a diagram of the protocol developed so far.

Succinct & Private Coin Proofs The coin proofs in the protocol presented thus far, which include all ancestor transactions involved in creating the coin, have a significant drawback. As time progresses, the number of transactions in the proof grows rapidly, eventually making verification computationally expensive.

To remedy this, we base the protocol on a *proof-carrying data* (PCD) scheme. This approach allows creating a coin proof with verification time and size that is independent of the number of transactions. Moreover, a PCD scheme that is zero-knowledge provides near perfect privacy: the coin proof hides transactions, balances, accounts, and more, revealing only the nullifier that generated the coin.

PCD was originally introduced in the context of distributed computing [14]. In the original setting, there is a network of distrusting nodes that output the result of certain computations and use each other's output data as input to subsequent computations. Proof-Carrying Data requires that every output is

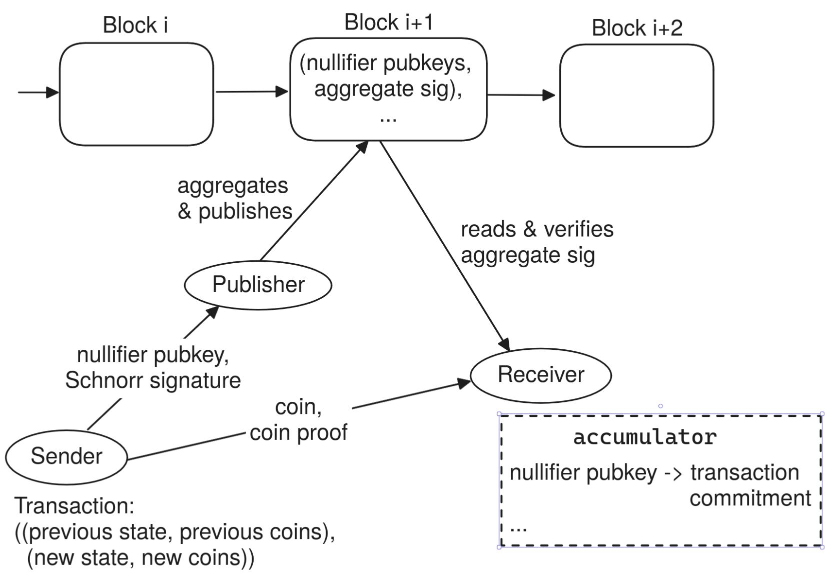
Block i
Block i+1
(nullifier pubkeys, aggregate sig), ...
Block i+2
aggregates & publishes
reads & verifies aggregate sig
Publisher
nullifier pubkey, Schnorr signature
Receiver
coin, coin proof
Sender
accumulator
nullifier pubkey -> transaction commitment
...
Transaction: ((previous state, previous coins), (new state, new coins))

Figure 1: Schematic depiction of a sender sending a payment to a receiver in a simplified variant of Shielded CSV.

accompanied by a proof that all computations that led to this output were done correctly. Fig. 2 shows an example of a PCD communication graph that connects nodes of computation.

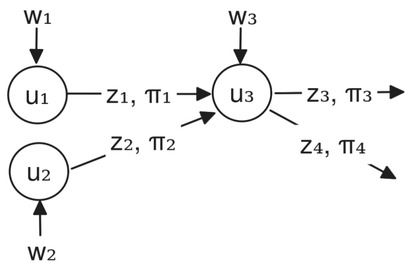

Figure 2: Exemplary Proof-Carrying Data communication graph. Nodes u_1 and u_2 receive local inputs w_1, w_2 and output z_1, z_2, respectively. They also output proofs π_1, π_2 that z_1, z_2 were correctly computed from w_1, w_2. Node u_3 receives (z_1, π_1) and (z_2, π_2) and outputs z_3 and z_4, each with a proof that the outputs were correctly computed from inputs z_1, z_2 and w_3, and that inputs z_1, z_2 themselves were correctly computed.

Intuitively, the proof is generated as follows: a node receives the output of other nodes, along with proofs of correctness. The node checks the proofs, outputs the computations result and proves correctness of the computation and the proofs of the input. This results in a recursive proof of correctness of all preceding computation that contributed to the output.

In order to make use of PCD, we need to describe Shielded CSV in terms of PCD: nodes in the graph correspond to protocol transactions and outputs of the computation are either account states or coins. Local inputs to the computation include, for instance, account ID and openings of the commitments. The core of the protocol description in Section 4 is the definition of the rules that constitute a correct computation.

Shielded CSV does not rely a specific PCD scheme, but we require it to be efficient (proving time, verification time and proof size do not depend on the size of the graph) and zero-knowledge (the proof does not leak any information about the inputs or local input). Section 6.4 discusses PCD schemes with these properties.

Fees One remaining issue with the protocol as described so far is the lack of built-in support for fee payments to publishers. One possible but flawed approach would be to add a coin to the transaction that effectively serves as a fee payment to a specific publisher. The idea is that once the publisher posts the corresponding nullifier as part of an aggregate nullifier to the blockchain, the transaction becomes valid, permitting the publisher to claim the coin. However, before the nullifier is included in the blockchain, the sender is unable to create a coin proof. Thus, the publisher would have to trust the sender to provide a valid coin proof after the nullifier has been published.

We aim to develop a protocol that does not require trust between publisher and sender: the first party to publish a nullifier should be guaranteed to receive the corresponding fee payment. Additionally, the protocol must not compromise on privacy, introduce interaction between the two parties, or force senders to commit to a particular publisher.

To achieve this, we first modify the definition of aggregate nullifiers to allow publishers to append their address for collecting fees:

$$\texttt{Aggregate Nullifier} := (\texttt{Nullifier Public Keys}, \texttt{Aggregate Signature}, \texttt{Publisher Address})$$

When scanning for aggregate nullifiers and storing a nullifier in the accumulator, users append the publisher's address to the stored transaction commitment.

The other modification to the protocol effectively transforms payments into a two-step process (which is implemented by representing payments with multiple PCD vertices instead of a single one). The process now works as follows:

1. The sender creates a transaction and a corresponding nullifier, then generates an intermediate proof. This proof can be created before the nullifier is included in the blockchain.

2. The publisher receives nullifier and proof, adds their address to the nullifer and publishes it. Once the nullifier is included, the publisher can create a regular coin for the fee payment and a corresponding coin proof. The address of the coin must match the address in the previously published aggregate nullifier. The sender, on the other hand, notices the nullifier in the nullifier accumulator and can subsequently create coin proofs for the transaction.

As a consequence of not requiring senders to commit to a specific publisher in the first step, it is possible to build a gossip network of Shielded CSV users and publishers. In this network, nullifiers are broadcast similarly to transactions in cryptocurrencies. Publishers learn about nullifiers from the network, aggregate them, post them to the blockchain, and claim a fee in Shielded CSV.

Dealing with Blockchain Reorganizations Blockchain nodes keep track of the current "best" chain, for example, the chain with the most proof-of-work. In some blockchains, such as those based on proof-of-work, it is possible for a node to experience a *blockchain reorganization*. This happens when the node receives a better blockchain that lacks at least one block from the previous best chain. A blockchain reorganization could replace a transaction with a conflicting transaction that sends the spent coins to someone other than the original receiver. Hence, receivers of transactions are recommended to wait for blockchain *confirmations*, i.e., a number of blocks chained on top of the block containing the transaction, which reduces the chance of a blockchain reorganization. If a blockchain reorganization occurs nonetheless, and the newly accepted best chain contains a conflicting transaction, the receiver cannot spend the coins created by the replaced transaction and all transactions depending on the replaced transaction are invalid and not part of the best best chain.

In the account-based private client-side validation setting, the effect of a blockchain reorganization is different because dependent transactions may still appear on the new best blockchain. Consider a scenario where a transaction recipient waits for some confirmations before publishing a nullifier to the blockchain that commits to a transaction. If the blockchain then reorganizes such that this user's account update nullifier remains in the chain, but the spent coins no longer exist, the user not only loses the received coins but also the entire account balance is effectively burned. This happens because the user published an account update on-chain spending non-existing coins, which prevents them from creating a coin proof.

Shielded CSV solves this problem by allowing users to add a *conditional nullifier accumulator value* (conditional NAV) to the transaction, which commits to the nullifier accumulator that contains the nullifiers of all the dependencies of the transaction, i.e., the previous state and the previous coins:

$$\texttt{Transaction} := (\texttt{Conditional NAV}, (\texttt{Previous State}, \texttt{Previous Coins}), (\texttt{New State}, \texttt{New Coins}))$$

Once the nullifier of the transaction is published on the blockchain, there are two different scenarios. Either the current best chain includes all nullifiers present in the conditional nullifier accumulator, or a reorganization has occurred, and it does not. In the former case, all dependencies of the transaction are in the current best chain and the user can create a coin proofs as usual. In the latter case, the user is permitted to create a proof for a specific transaction that differs from the one originally committed. This specific transaction has no effect: it neither spends nor creates new coins and the new account state's balance equals that of the previous account state. Therefore, despite the blockchain reorganization, the user has a proof for an account state as if the transaction was never committed and can continue to make new transactions with the account.

3 Preliminaries

3.1 Notation

The security parameter is denoted λ.

System-wide parameters pp are generated by a setup algorithm Setup taking as input the security parameter. For notational simplicity, we assume that pp is given as implicit input to all algorithms described below. We write $x \leftarrow y$ to denote the assignment of the value of y to x. Similarly, for a randomized algorithm A, we write $y \leftarrow\!\!\$\ \mathsf{A}(x)$ to denote that y is distributed according to the output of $\mathsf{A}(x)$ (over uniformly sampled random coins).

We denote the set of *polynomially-bounded* functions in the security parameter λ by $poly(\lambda) = \{f : \exists a \in \mathbb{N},\ f(\lambda) \in O(\lambda^a)\}$, the set of *negligible* functions in the security parameter λ by $negl(\lambda) = \{f : f(\lambda)^{-1} \notin poly(\lambda)\}$.

A probabilistic interactive Turing machine is *probabilistic polynomial-time (PPT)* if its runtime is in $poly(\lambda)$; it is *probabilistic expected polynomial-time (expected-PPT)* if its expected runtime is in $poly(\lambda)$.

We use additive notation for the group operation.

3.2 Tuple

A tuple $\boldsymbol{y}$ is a *prefix* of $\boldsymbol{y'}$ if $|\boldsymbol{y}| \leq |\boldsymbol{y'}|$ and $y_i = y'_i$ for all $i = 1..|\boldsymbol{y}|$. We define the predicate $\mathsf{IsPrefix}(\boldsymbol{y}, \boldsymbol{y'})$ to output 1 if $\boldsymbol{y}$ is a prefix of $\boldsymbol{y'}$ and 0 otherwise.

We also define the predicate $\mathsf{DistinctElement}(\boldsymbol{y}, \boldsymbol{y'})$ to output 1 if there exists $i < \min(|\boldsymbol{y}|, |\boldsymbol{y'}|)$ such that $y_i \neq y'_i$ and 0 otherwise.

3.3 Commitment

A commitment scheme consists of algorithm $\mathsf{Com}_{\mathsf{Commit}}$:

- $\mathsf{Com}_{\mathsf{Commit}}(m; r) \rightarrow C$ outputs a commitment C to message m with randomness r.

Security A commitment scheme is *binding* if no PPT adversary produces two distinct messages m_0, m_1 and randomness r_0, r_1 such that $\mathsf{Com}(m_0; r_0) = \mathsf{Com}(m_1; r_1)$ (except with probability negligible in λ). A commitment scheme is *hiding* if no PPT adversary obtains any information about the message from the commitment.

3.4 Non-interactive Signature Half-Aggregation with Commitments

Starting from the notion of a non-interactive signature half-aggregation scheme [13], we add the ability to commit to messages in the signature. The resulting scheme consists of the following algorithms:

- $\mathsf{Sig}_{\mathsf{KeyGen}}() \rightarrow (\mathsf{sk}, \mathsf{pk})$ outputs a signing key sk and a public key pk.
- $\mathsf{Sig}_{\mathsf{Sign}}(\mathsf{sk}, \mathsf{m}, \mathsf{m}_{\mathsf{SC}}) \rightarrow (\sigma, \mathsf{r}_{\mathsf{SC}})$ outputs a signature σ on message $\mathsf{m} \in \mathcal{M}$ that commits to message $\mathsf{m}_{\mathsf{SC}} \in \mathcal{MSC}$ using randomness r_{SC}.
- $\mathsf{Sig}_{\mathsf{Verify}}(\mathsf{m}, \mathsf{pk}, \sigma) \rightarrow b$ outputs $b = 1$ if σ is a valid signature on message m for public key pk and $b = 0$ otherwise.
- $\mathsf{Sig}_{\mathsf{AggregateSig}}((\mathsf{m}_1, \mathsf{pk}_1, \sigma_1), \ldots (\mathsf{m}_n, \mathsf{pk}_n, \sigma_n)) \rightarrow \sigma_{\mathsf{Aggr}}$ outputs an aggregated signature σ_{Aggr} given triples of message, public key and signature.
- $\mathsf{Sig}_{\mathsf{AggregateVerify}}(\sigma_{\mathsf{Aggr}}, (\mathsf{m}_1, \mathsf{pk}_1), \ldots (\mathsf{m}_n, \mathsf{pk}_n)) \rightarrow b$ outputs $b = 1$ if the aggregate signature is valid for the tuples of message and public key and $b = 0$ otherwise.

- $\mathsf{Sig}_{\mathsf{CommRetrieve}}(\sigma_{\mathsf{Aggr}}, i) \rightarrow C$ outputs the commitment made in the i-th signature from aggregate signature σ_{Aggr}.
- $\mathsf{Sig}_{\mathsf{CommVerify}}(C, \mathsf{m}_{\mathsf{SC}}, \mathsf{r}_{\mathsf{SC}}) \rightarrow b$ outputs 1 if C is a commitment to message m_{SC} with randomness r_{SC} and 0 otherwise.

The sets $\mathcal{M}$ and $\mathcal{MSC}$ are specified by parameters pp.

A non-interactive signature half-aggregation scheme with commitments must satisfy the following completeness properties:

1. For all $(\mathsf{sk}, \mathsf{pk}) \leftarrow\!\$\; \mathsf{Sig}_{\mathsf{KeyGen}}()$, and for all $\mathsf{m} \in \mathcal{M}, \mathsf{m}_{\mathsf{SC}} \in \mathcal{MSC}$,

$$\begin{aligned}(\sigma, \mathsf{r}_{\mathsf{SC}}) &\leftarrow\!\$\; \mathsf{Sig}_{\mathsf{Sign}}(\mathsf{sk}, \mathsf{m}, \mathsf{m}_{\mathsf{SC}}) \\ \mathsf{Sig}_{\mathsf{Verify}}(\mathsf{m}, \mathsf{pk}, \sigma) &= 1.\end{aligned}$$

2. For all $n \in \mathbb{N}$, for all $(\mathsf{sk}_i, \mathsf{pk}_i) \leftarrow\!\$\; \mathsf{Sig}_{\mathsf{KeyGen}}()$, and all $\mathsf{m}_i \in \mathcal{M}, \mathsf{m}_{\mathsf{SC}i} \in \mathcal{MSC}$ for $= 1..n$,

$$\begin{aligned}(\sigma_i, \mathsf{r}_{\mathsf{msg}i}) &\leftarrow\!\$\; \mathsf{Sig}_{\mathsf{Sign}}(\mathsf{sk}_i, \mathsf{m}_i, \mathsf{m}_{\mathsf{SC}i}) \text{ for } i = 1..n \\ \sigma_{\mathsf{Aggr}} &\leftarrow \mathsf{Sig}_{\mathsf{AggregateSig}}((\mathsf{m}_1, \mathsf{pk}_1, \sigma_1), \ldots (\mathsf{m}_n, \mathsf{pk}_n, \sigma_n)) \\ C_i &\leftarrow \mathsf{Sig}_{\mathsf{CommRetrieve}}(\sigma_{\mathsf{Aggr}}, i) \text{ for } i = 1..n \\ 1 &= \mathsf{Sig}_{\mathsf{AggregateVerify}}(\sigma_{\mathsf{Aggr}}, (\mathsf{m}_1, \mathsf{pk}_1), \ldots (\mathsf{m}_n, \mathsf{pk}_n)) \\ \wedge\ 1 &= \mathsf{Sig}_{\mathsf{CommVerify}}(C_i, \mathsf{m}_{\mathsf{SC}i}, \mathsf{r}_{\mathsf{msg}i}) \text{ for } i = 1..n.\end{aligned}$$

Security The algorithms $\mathsf{Sig}_{\mathsf{KeyGen}}, \mathsf{Sig}_{\mathsf{Sign}}, \mathsf{Sig}_{\mathsf{Verify}}$ define a signature scheme. A signature scheme is *existentially unforgeable under chosen message attacks* (EUF-CMA) if no PPT adversary (except with probability negligible in λ) with access to the public key pk and a signing oracle that produces signatures for pk and messages chosen by the adversary can produce a signature for a message the adversary has not previously given to the signing oracle.

An non-interactive signature half aggregation signature scheme is *chosen-key aggregate existential forgery under chosen-message attacks* (CK-AEUF-CMA) secure if a PPT adversary that is given public key pk and access to a signing oracle for pk cannot produce an aggregate signature that passes AggregateVerify with a list of tuples that contains pk and a new message (except with probability negligible in λ).

For a non-interactive signature half-aggregation with commitments we define a new security notion *chosen-key aggregate existential forgery under chosen-commitments and chosen-message attacks* (CK-AEUF-CC-CMA). In this game the adversary can request signatures with commitments to messages chosen by the adversary. The adversary wins the game if they break the binding property of the commitment, create a signature forgery, if the signature commits to a new message, or if the commitment cannot be opened. More specifically, a non-interactive signature half-aggregation with commitments scheme is CK-AEUF-CC-CMA secure if no PPT adversary (except with probability negligible in λ) wins the following game:

```
$\mathsf{CK\text{-}AEUF\text{-}CC\text{-}CMA}^{\mathcal{A}}(\lambda)$
$pp \leftarrow\!\$\ \mathsf{Setup}(1^\lambda)$
$(\mathsf{sk}^*, \mathsf{pk}^*) \leftarrow\!\$\ \mathsf{Sig}_{\mathsf{KeyGen}}()$
$L \leftarrow \emptyset$
$(\sigma_{\mathsf{Aggr}}, \mathsf{m}_{\mathsf{SC}}, \mathsf{r}_{\mathsf{SC}}, \mathsf{m}_{\mathsf{SC2}}, \mathsf{r}_{\mathsf{SC2}},$
  $(\mathsf{m}_1, \mathsf{pk}_1), \ldots, (\mathsf{m}_n, \mathsf{pk}_n)) \leftarrow\!\$\ \mathcal{A}^{\mathsf{OSign}}(pp, \mathsf{pk}^*)$
if $\mathsf{Sig}_{\mathsf{AggregateVerify}}(\sigma_{\mathsf{Aggr}}, (\mathsf{m}_1, \mathsf{pk}_1), \ldots, (\mathsf{m}_n, \mathsf{pk}_n)) = 0$
  return $0$
for $i = 1..n$
  $C \leftarrow \mathsf{Sig}_{\mathsf{CommRetrieve}}(\sigma_{\mathsf{Aggr}}, i)$
  if $\mathsf{Sig}_{\mathsf{CommVerify}}(C, \mathsf{m}_{\mathsf{SC}}, \mathsf{r}_{\mathsf{SC}}) = 1 \wedge \mathsf{Sig}_{\mathsf{CommVerify}}(C, \mathsf{m}_{\mathsf{SC2}}, \mathsf{r}_{\mathsf{SC2}}) = 1$
    // Commitment is not binding
    return $1$
  if $\mathsf{pk}_i = \mathsf{pk}^*$
    if $\forall(\mathsf{m}', \ldots) \in L : \mathsf{m}' \neq \mathsf{m}_i$
      // Signature Forgery
      return $1$
    if $\exists(\mathsf{m}', \mathsf{m}'_{\mathsf{SC}}, \ldots) \in L : \mathsf{m}' = \mathsf{m}_i \wedge \mathsf{m}'_{\mathsf{SC}} \neq \mathsf{m}_{\mathsf{SC}}$
      $\wedge\ \mathsf{Sig}_{\mathsf{CommVerify}}(C, \mathsf{m}_{\mathsf{SC}}, \mathsf{r}_{\mathsf{SC}}) = 1$
      // Signature commits to new message
      return $1$
    if $\forall(\mathsf{m}', \mathsf{m}'_{\mathsf{SC}}, \mathsf{r}'_{\mathsf{SC}}) \in L : \mathsf{m}' \neq \mathsf{m}_i \vee \mathsf{Sig}_{\mathsf{CommVerify}}(C, \mathsf{m}'_{\mathsf{SC}}, \mathsf{r}'_{\mathsf{SC}}) = 0$
      // Commitment cannot be opened
      return $1$
return $0$
```

```
$\mathsf{OSign}(\mathsf{m}, \mathsf{m}_{\mathsf{SC}})$
$(\sigma, \mathsf{r}_{\mathsf{SC}}) \leftarrow\!\$\ \mathsf{Sig}_{\mathsf{Sign}}(\mathsf{sk}, \mathsf{m}, \mathsf{m}_{\mathsf{SC}})$
$L \leftarrow L \cup (\mathsf{m}, \mathsf{m}_{\mathsf{SC}}, \mathsf{r}_{\mathsf{SC}})$
return $\sigma$
```

3.5 Accumulator

An accumulator is an efficient representation of a set [15]. The type of accumulator used in Shielded CSV supports insertion of elements, verification of the insertion operation, and non-membership proofs. There are two types of parties involved in operating an accumulator: the accumulator manager and the accumulator verifier.

The accumulator manager has access to the following algorithms:

- $\mathsf{AccM}_{\mathsf{New}}() \to \mathsf{state}$ outputs the initial state state of the accumulator.
- $\mathsf{AccM}_{\mathsf{Value}}(\mathsf{state}) \to v$ outputs the accumulator value v given accumulator state state.
- $\mathsf{AccM}_{\mathsf{ProveNonMembershipAndInsert}}(\mathsf{state}, x) \to (\mathsf{state}', \pi)$ outputs a new state state' after inserting the element $x \in \mathcal{X}_{\mathsf{Acc}}$ into the accumulator with state state and outputs a proof π that attests to the correctness of the insertion and to the non-membership of x in the accumulator with state state.

The accumulator verifier has access to the following algorithms:

- $\mathsf{AccV}_{\mathsf{New}}() \to v$ outputs the value v of an empty accumulator.
- $\mathsf{AccV}_{\mathsf{VerifyNonMembershipAndInsert}}(v, v', x, \pi) \to b$ outputs $b = 1$ if the proof π proves that $x \in \mathcal{X}_{\mathsf{Acc}}$ is not a member of the set represented by accumulator value v and v' is the value of the accumulator after inserting x into the set represented by v. Otherwise, it outputs $b = 0$.

The set $\mathcal{X}_{\mathsf{Acc}}$ is specified by parameters pp.

The accumulator state may be dependent on the order of insertion. We denote the set of states after inserting all elements in some set $\mathcal{X} \subseteq \mathcal{X}_{\mathsf{Acc}}$ by $\mathcal{S}_{\mathcal{X}}$.

The accumulator scheme must satisfy the following completeness property: For all $\mathcal{X} \subseteq \mathcal{X}_{\mathsf{Acc}}$, all states $\mathsf{state} \in \mathcal{S}_{\mathcal{X}}$ and all elements $x \notin \mathcal{X}$, we have

$$\begin{aligned} v &\leftarrow \mathsf{AccM}_{\mathsf{Value}}(\mathsf{state}) \\ (\mathsf{state}', \pi) &\leftarrow\!\!\$\ \mathsf{AccM}_{\mathsf{ProveNonMembershipAndInsert}}(\mathsf{state}, x) \\ v' &\leftarrow \mathsf{AccM}_{\mathsf{Value}}(\mathsf{state}') \\ 1 &= \mathsf{AccV}_{\mathsf{VerifyNonMembershipAndInsert}}(v, v', x, \pi). \end{aligned}$$

Security An accumulator scheme is A-SEC ("accumulator secure") if no PPT adversary (except with probability negligible in λ) given pp outputs tuples $\boldsymbol{v}, \boldsymbol{x}, \boldsymbol{\pi}$ such that

$$\begin{aligned} \mathsf{AccV}_{\mathsf{VerifyNonMembershipAndInsert}}(v_i, v_{i+1}, x_i, \pi_i) = 1 \text{ for } i = 1..|\boldsymbol{x}| \\ \wedge \exists i, j : (i \neq j \wedge x_i = x_j). \end{aligned}$$

3.6 ToS-Accumulator

A ToS-accumulator represents a tuple of sets. The ToS-accumulator allows appending a set to the tuple and membership proofs for the union of sets in the tuple. Moreover, the ToS-accumulator allows proving that the tuple represented by one ToS-accumulator is a prefix of the tuple represented by another ToS-accumulator or that the represented tuples have distinct elements.

Similar to regular accumulators, there are two types of parties involved in a ToS-accumulator: the manager and the verifier.

The ToS-accumulator manager has access to the following algorithms:

- $\mathsf{ToSAccM}_{\mathsf{New}}() \rightarrow \mathsf{state}$ outputs the initial state state of the accumulator.
- $\mathsf{ToSAccM}_{\mathsf{Value}}(\mathsf{state}) \rightarrow v$ outputs the accumulator value v given accumulator state state.
- $\mathsf{ToSAccM}_{\mathsf{AppendSet}}(\mathsf{state}, \mathcal{X}) \rightarrow \mathsf{state}'$ outputs a new state state' after appending the set $\mathcal{X} \subseteq \mathcal{X}_{\mathsf{ToSAcc}}$ to the tuple of sets represented by the accumulator with state state.
- $\mathsf{ToSAccM}_{\mathsf{RemoveSet}}(\mathsf{state}) \rightarrow \mathsf{state}'$ outputs a new state state' after removing the last appended set from the accumulator with state state.
- $\mathsf{ToSAccM}_{\mathsf{ProveUnionMembership}}(\mathsf{state}, x) \rightarrow \pi$ outputs proof π that $x \in \mathcal{X}_{\mathsf{ToSAcc}}$ is a member of the union of sets represented by the accumulator with state state.
- $\mathsf{ToSAccM}_{\mathsf{ProveIsPrefix}}(\mathsf{state}, \mathsf{state}') \rightarrow \pi$ outputs a proof π that the tuple represented by the accumulator with state state is a prefix of the tuple represented by accumulator with state state'.
- $\mathsf{ToSAccM}_{\mathsf{ProveDistinctElement}}(\mathsf{state}, \mathsf{state}') \rightarrow \pi$ outputs a proof π that $\mathsf{DistinctElement}(\boldsymbol{\mathcal{X}}, \boldsymbol{\mathcal{X}}')$ for tuple $\boldsymbol{\mathcal{X}}$ represented by the accumulator with state state and tuple $\boldsymbol{\mathcal{X}}'$ represented by the accumulator with state state'.

The ToS-accumulator verifier has access to the following algoritms:

- $\mathsf{ToSAccV}_{\mathsf{New}}() \rightarrow v$ outputs the value v of an empty accumulator.
- $\mathsf{ToSAccV}_{\mathsf{VerifyUnionMembership}}(v, x, \pi) \rightarrow b$ outputs $b = 1$ if the proof π proves that $x \in \mathcal{X}_{\mathsf{ToSAcc}}$ is a member of the union of sets represented by accumulator with value v. Otherwise, it outputs $b = 0$.
- $\mathsf{ToSAccV}_{\mathsf{VerifyIsPrefix}}(v, v', \pi) \rightarrow b$ outputs $b = 1$ if the proof π proves that the tuple represented by accumulator value v is a prefix of the tuple represented by v'. Otherwise, it outputs $b = 0$.
- $\mathsf{ToSAccV}_{\mathsf{VerifyDistinctElement}}(v, v', \pi) \rightarrow b$ outputs $b = 1$ if proof π proves that $\mathsf{DistinctElement}(\boldsymbol{\mathcal{X}}, \boldsymbol{\mathcal{X}}') = 1$ for tuple $\boldsymbol{\mathcal{X}}$ represented by accumulator value v and tuple $\boldsymbol{\mathcal{X}}'$ represented by accumulator value v'. Otherwise, it outputs $b = 0$.

The set $\mathcal{X}_{\mathsf{ToSAcc}}$ is specified by parameters pp.

To simplify the definition of ToS-accumulator completeness and security we define the algorithm $\mathsf{ToSAccM}_{\mathsf{AppendSetMulti}}(\boldsymbol{\mathcal{X}})$ for tuple $\boldsymbol{\mathcal{X}}$ whose elements are in $\mathcal{X}_{\mathsf{ToSAcc}}$:

```
$\mathsf{ToSAccM}_{\mathsf{AppendSetMulti}}(\boldsymbol{\mathcal{X}})$
1 : $\mathsf{state} \leftarrow \mathsf{ToSAccM}_{\mathsf{New}}()$
2 : **for** $i = 1..|\boldsymbol{\mathcal{X}}|$
3 :     $\mathsf{state} \leftarrow \mathsf{ToSAccM}_{\mathsf{AppendSet}}(\mathsf{state}, \mathcal{X}_i)$
4 : **return** $\mathsf{state}$
```

The ToS-accumulator scheme must satisfy the following completeness properties for all $n \in \mathbb{N}$, all tuples $\boldsymbol{\mathcal{X}} \in \mathcal{P}(\mathcal{X}_{\mathsf{ToSAcc}})^n$:

1. For all $x \in \bigcup \mathcal{X}_i$, we have

$$\begin{aligned}\mathsf{state} &\leftarrow \mathsf{ToSAccM}_{\mathsf{AppendSetMulti}}(\boldsymbol{\mathcal{X}})\\ \pi &\leftarrow\!\!\$\ \mathsf{ToSAccM}_{\mathsf{ProveUnionMembership}}(\mathsf{state}, x)\\ v &\leftarrow \mathsf{ToSAccM}_{\mathsf{Value}}(\mathsf{state})\\ 1 &= \mathsf{ToSAccV}_{\mathsf{VerifyUnionMembership}}(v, x, \pi).\end{aligned}$$

2. For all $n' \in \mathbb{N}$ and all tuples $\boldsymbol{\mathcal{X}}' \in \mathcal{P}(\mathcal{X}_{\mathsf{ToSAcc}})^{n'}$, let

$$\mathsf{state} \leftarrow \mathsf{ToSAccM}_{\mathsf{AppendSetMulti}}(\boldsymbol{\mathcal{X}}), v \leftarrow \mathsf{ToSAccM}_{\mathsf{Value}}(\mathsf{state})$$
$$\mathsf{state}' \leftarrow \mathsf{ToSAccM}_{\mathsf{AppendSetMulti}}(\boldsymbol{\mathcal{X}}'), v' \leftarrow \mathsf{ToSAccM}_{\mathsf{Value}}(\mathsf{state}').$$

   Then, if $\mathsf{IsPrefix}(\boldsymbol{\mathcal{X}}, \boldsymbol{\mathcal{X}}') = 1$, we have

$$\begin{aligned}\pi &\leftarrow\!\!\$\ \mathsf{ToSAccM}_{\mathsf{ProveIsPrefix}}(\mathsf{state}, \mathsf{state}')\\ 1 &= \mathsf{ToSAccV}_{\mathsf{VerifyIsPrefix}}(v, v', \pi)\end{aligned}$$

   and if $\mathsf{DistinctElement}(\boldsymbol{\mathcal{X}}, \boldsymbol{\mathcal{X}}') = 1$, we have

$$\begin{aligned}\pi &\leftarrow\!\!\$\ \mathsf{ToSAccM}_{\mathsf{ProveDistinctElement}}(\mathsf{state}, \mathsf{state}')\\ 1 &= \mathsf{ToSAccV}_{\mathsf{VerifyDistinctElement}}(v, v', \pi).\end{aligned}$$

Security To characterize the security of ToS-accumulators, we define the game ToSA-SEC. In this game, the adversary outputs a tuple of sets for $\mathsf{ToSAccM}_{\mathsf{AppendSet}}$ state transitions. They win if they either provide a set membership proof for an element not in the union of tuple elements, a IsPrefix proof for a tuple that is not a prefix, or a DistinctElement proof for tuples that do not have distinct elements. More specifically, a ToS-accumulator is ToSA-SEC ("ToS-accumulator secure")if no PPT adversary $\mathcal{A}$ wins the following game (except with probability negligible in λ):

```
$\mathsf{ToSA\text{-}SEC}^{\mathcal{A}}(\lambda)$
1 : $pp \leftarrow\!\!\$\ \mathsf{Setup}(1^\lambda)$
2 : $(\boldsymbol{\mathcal{X}}, n, x, \boldsymbol{\mathcal{X}}', \pi) \leftarrow\!\!\$\ \mathcal{A}(pp)$
3 : $\mathsf{state} \leftarrow \mathsf{ToSAccM}_{\mathsf{AppendSetMulti}}(\boldsymbol{\mathcal{X}})$
4 : **for** $i = 1..|\boldsymbol{\mathcal{X}}| - n$
5 :     $\mathsf{state} \leftarrow \mathsf{ToSAccM}_{\mathsf{RemoveSet}}(\mathsf{state})$
6 : $\boldsymbol{\mathcal{X}} \leftarrow (\mathcal{X}_1, \ldots, \mathcal{X}_n)$
7 : $v \leftarrow \mathsf{ToSAccM}_{\mathsf{Value}}(\mathsf{state})$
8 : **if** $x \notin \bigcup_{i=1}^{n} \mathcal{X}_i \wedge \mathsf{ToSAccV}_{\mathsf{VerifyUnionMembership}}(v, x, \pi) = 1$
9 :     **return** 1
10 : $v' \leftarrow \mathsf{ToSAccM}_{\mathsf{Value}}(\mathsf{ToSAccM}_{\mathsf{AppendSetMulti}}(\boldsymbol{\mathcal{X}}'))$
11 : **if** $\mathsf{IsPrefix}(\boldsymbol{\mathcal{X}}, \boldsymbol{\mathcal{X}}') = 0 \wedge \mathsf{ToSAccV}_{\mathsf{VerifyIsPrefix}}(v, v', \pi) = 1$
12 :     **return** 1
13 : **if** $\mathsf{DistinctElement}(\boldsymbol{\mathcal{X}}, \boldsymbol{\mathcal{X}}') = 0 \wedge \mathsf{ToSAccV}_{\mathsf{VerifyDistinctElement}}(v, v', \pi) = 1$
14 :     **return** 1
15 : **return** 0
```

3.7 Proof-Carrying Data

We define a proof-carrying data (PCD) scheme for a class of compliance predicates Φ as follows (similar to [16]).

A PCD *transcript* T is a directed acyclic graph where each vertex u is labeled by local data $\mathsf{w}_{\mathsf{loc}}^{(u)}$ and each edge e is labeled by a message $z^{(e)} \neq \bot$. The output of a transcript $\mathsf{out}(T)$ is the message $z^{(e)}$ where $e = (u, v)$ is the lexicographically-first edge such that v is a sink.

A vertex u is *ϕ-compliant* for $\phi \in \Phi$ if for all outgoing edges $e = (u, v)$:

- (base case) if u has no incoming edges, $\phi(z^{(e)}, \mathsf{w}_{\mathsf{loc}}^{(u)}, \bot, \ldots, \bot) = 1$.
- (recursive case) if u has incoming edges $e_1, \ldots, e_m$, $\phi(z^{(e)}, \mathsf{w}_{\mathsf{loc}}^{(u)}, z^{(e_1)}, \ldots, z^{(e_m)}) = 1$.

We say that PCD transcript T is ϕ-compliant if all of its vertices are ϕ-compliant.

A PCD scheme consists of the following algorithms:

- $\mathsf{PCD}_{\mathsf{KeyGen}}(\phi) \to (\mathsf{prk}, \mathsf{vk})$ outputs a proving key prk and a verification key vk.
- $\mathsf{PCD}_{\mathsf{Prove}}(\mathsf{prk}, z, \mathsf{w}_{\mathsf{loc}}, [z_i, \pi_i]_{i=1}^m) \to \pi$ outputs a proof π that $\phi(z, \mathsf{w}_{\mathsf{loc}}, z_1, \ldots, z_m) = 1$ and that all proofs π_i for messages z_i are valid.
- $\mathsf{PCD}_{\mathsf{Verify}}(\mathsf{vk}, z, \pi) \to b$ outputs 1 if the proof π is correct for verification key vk and message z, and outputs 0 otherwise.

The PCD scheme must satisfy the following completeness property: For every adversary $\mathcal{A}$, we have

$$\begin{aligned} & pp \leftarrow\$ \mathsf{Setup}(1^\lambda) \\ & (\phi, z, \mathsf{w}_{\mathsf{loc}}, [z_i, \pi_i]_{i=1}^m) \leftarrow\$ \mathcal{A}(pp) \\ & (\mathsf{prk}, \mathsf{vk}) \leftarrow\$ \mathsf{PCD}_{\mathsf{KeyGen}}(\phi) \\ & \pi \leftarrow\$ \mathsf{PCD}_{\mathsf{Prove}}(\mathsf{prk}, z, \mathsf{w}_{\mathsf{loc}}, [z_i, \pi_i]_{i=1}^m) \\ & (\phi \in \Phi \wedge \phi(z, \mathsf{w}_{\mathsf{loc}}, z_1, \ldots, z_m) = 1 \\ & \wedge (\forall i = 1..m : z_i = \bot \vee \mathsf{PCD}_{\mathsf{Verify}}(\mathsf{vk}, z_i, \pi_i) = 1)) \implies \mathsf{PCD}_{\mathsf{Verify}}(\mathsf{vk}, z, \pi) = 1. \end{aligned}$$

For a PCD scheme to be considered *efficient*, both the prover and verifier must run in polynomial time. Additionally, the verification time, proving time, and proof size must be independent of the transcript's length.

Security A PCD scheme has knowledge soundness if for every expected polynomial time adversary $\mathcal{A}$ there exists an expected polynomial time extractor $\mathsf{Ext}_{\mathcal{A}}$, such that for compliance predicate $\phi \in \Phi$:

$$\Pr\left[\begin{array}{c|r} \mathsf{PCD}_{\mathsf{Verify}}(\mathsf{vk}, z, \pi) = 1 & pp \leftarrow\$ \mathsf{Setup}(1^\lambda) \\ (\mathsf{out}(T) \neq z \vee T \text{ not } \phi\text{-compliant}) & (\mathsf{prk}, \mathsf{vk}) \leftarrow\$ \mathsf{PCD}_{\mathsf{KeyGen}}(\phi) \\ & (z, \pi) \leftarrow\$ \mathcal{A}(\mathsf{prk}) \\ & T \leftarrow\$ \mathsf{Ext}_{\mathcal{A}}(\mathsf{prk}) \end{array}\right] \leq negl(\lambda)$$

A PCD scheme has *statistical zero knowledge* if there exists a probabilistic polynomial-time simulator Sim such that for every honest prover $\mathcal{P}$ the distributions

$$\Pr\left[\begin{array}{c|r} (pp, \phi, z, \pi) & pp \leftarrow\$ \mathsf{Setup}(1^\lambda) \\ & (\phi, z, \mathsf{w}_{\mathsf{loc}}, [z_i, \pi_i]_{i=1}^m) \leftarrow\$ \mathcal{P}(pp) \\ & (\mathsf{prk}, \mathsf{vk}) \leftarrow\$ \mathsf{PCD}_{\mathsf{KeyGen}}(\phi) \\ & \pi \leftarrow\$ \mathsf{PCD}_{\mathsf{Prove}}(\mathsf{prk}, z, \mathsf{w}_{\mathsf{loc}}, [z_i, \pi_i]_{i=1}^m) \end{array}\right]$$

and

$$\Pr\left[\begin{array}{c|r} (pp, \phi, z, \pi) & (pp, \tau) \leftarrow\$ \mathsf{Sim}(1^\lambda) \\ & (\phi, z, \mathsf{w}_{\mathsf{loc}}, [z_i, \pi_i]_{i=1}^m) \leftarrow\$ \mathcal{P}(pp) \\ & \pi \leftarrow\$ \mathsf{Sim}(\tau, \phi, z)] \end{array}\right]$$

are statistically close.

4 Protocol

In this section, we define the protocol using the primitives introduced in Section 3. While most primitives remain abstract, we make certain primitives concrete to specify the exact number of bytes that need to be written to the blockchain for each payment. The data written to the blockchain, referred to as *aggregate nullifiers*, consists of a commitment to the fee receiver's account, a half-aggregate signature, and corresponding public keys.

For the commitment scheme, we employ Pedersen commitments which consist of a single group element. The details of the aggregation scheme used in the protocol will be described in the following section.

4.1 Non-interactive *Schnorr* Signature Half-Aggregation with Commitments

We construct our scheme, Non-interactive *Schnorr* Signature Half-Aggregation with Commitments (NISSHAC), by extending non-interactive Schnorr signature half-aggregation [13] with a simple method for adding commitments. Non-interactive Schnorr signature half-aggregation allows aggregating ordinary Schnorr signatures into a single signature that is about half as large as the individual signatures.

NISSHAC modifies the $\mathsf{Sig}_{\mathsf{Sign}}$ algorithm of the original half-aggregation scheme and introducing two new algorithms: $\mathsf{Sig}_{\mathsf{CommRetrieve}}$ and $\mathsf{Sig}_{\mathsf{CommVerify}}$. The technique for incorporating commitments, known as "sign-to-contract" in Bitcoin folklore, bears resemblance to pay-to-contract commitments [17]. In NISSHAC, a public key is an element of group a $\mathbb{G}$ (as in ordinary Schnorr signatures) and an aggregate of n signatures is a tuple consisting of a scalar and n group elements.

Let G be the generator of group $\mathbb{G}$ of order p, where p is λ-bit prime, and let $\mathsf{H}_{\mathsf{Sig}}$ and $\mathsf{H}_{\mathsf{SigComm}}$ be hash functions. We modify the $\mathsf{Sig}_{\mathsf{Sign}}$ algorithm of Schnorr signatures to enable commitments to message m_{SC}, with the changed line highlighted:

```
$\mathsf{Sig}_{\mathsf{Sign}}(\mathsf{sk}, \mathsf{m}, \mathsf{m}_{\mathsf{SC}})$
1 : $k \leftarrow\!\$\ \mathbb{F}_p, R' \leftarrow kG$
2 : $R = R' + \mathsf{H}_{\mathsf{SigComm}}(R', \mathsf{m}_{\mathsf{SC}})G$
3 : $x \leftarrow \mathsf{sk}, P \leftarrow xG$
4 : $s = k + \mathsf{H}_{\mathsf{Sig}}(R, P, \mathsf{m})$
5 : $\sigma \leftarrow (R, s)$
6 : return $(\sigma, R')$
```

Furthermore, we add algorithms $\mathsf{Sig}_{\mathsf{CommRetrieve}}$ and $\mathsf{Sig}_{\mathsf{CommVerify}}$:

```
$\mathsf{Sig}_{\mathsf{CommRetrieve}}(\sigma_{\mathsf{Aggr}}, i)$
1 : $(R_1, \ldots, R_n, s_{\mathsf{Aggr}}) \leftarrow \sigma_{\mathsf{Aggr}}$
2 : return $R_i$
```

```
$\mathsf{Sig}_{\mathsf{CommVerify}}(C, \mathsf{m}_{\mathsf{SC}}, \mathsf{r}_{\mathsf{SC}})$
1 : $R' \leftarrow \mathsf{r}_{\mathsf{SC}}$
2 : if $C = R' + \mathsf{H}_{\mathsf{SigComm}}(R', \mathsf{m}_{\mathsf{SC}})G$
3 :     return 1
4 : return 0
```

Claim 1. NISSHAC *is* CK-AEUF-CC-CMA *secure in the random oracle model under the discrete logarithm assumption.*

Claim 2. *Commitments R_i in a* NISSHAC *signature $(R_1, \ldots, R_n, s)$, are (statistically) hiding in the random oracle model.*

4.2 Protocol Description

We provide pseudocode of the protocol in the form of a program written in the Rust Programming Language, available at `https://github.com/ShieldedCSV/ShieldedCSV/tree/paper`. The remainder of this section is a high-level description of the key concepts and data structures used in the protocol.

Account ID An account ID is an element of group $\mathbb{G}$ that is generated by running $\mathsf{Sig}_{\mathsf{KeyGen}}$ at the time of account creation.

Address An address is a hiding commitment to an account ID.

Accumulators There are two types of accumulators in Shielded CSV:

- *spent accumulator*: an accumulator for every account, managed by the owner of the account.
- *nullifier accumulator*: a ToS-accumulator managed by each user of the protocol.

Coin In Shielded CSV, a payment involves the sender providing the receiver with a `Coin` and a *coin proof.* A `Coin` contains contextual information dependent on both the transaction that created it and the current blockchain state. To accommodate situations where Shielded CSV needs only essential coin information without full context, we define a `CoinEssence` object. This object contains:

- The address of the owner
- The *amount*, which is the number of units of the asset represented by this `Coin`
- The index of the `Coin` in the transaction that created it

A complete `Coin` consists of:

- The `CoinEssence`
- The hash of the transaction that created the `Coin`
- The location of the nullifier in the blockchain that created this `Coin`
- The value of a nullifier accumulator containing the nullifier that created this `Coin`

There are two ways to identify a `Coin` in Shielded CSV:

1. The `CoinID` is the hash of the transaction that created the coin and the index of the `Coin` in the transaction.
2. The `CoinIDOnChain` is the location of the nullifier in the blockchain that created this `Coin` and the index of the `Coin` in the transaction.

The coin proof is a PCD proof that, along with the `Coin`, is passed to $\mathsf{PCD}_{\mathsf{Verify}}$ before the receiver accepts the payment.

AcctState An `AcctState` represents the state of an account and, similarly to `Coin`, is dependent on blockchain state. Thus, we define `AcctStateEssence` that contains only the essential information of an account state:

- The account ID
- The account balance
- The Schnorr public key that will nullify the account state

A complete `AcctState` consists of:

- The `AcctStateEssence`
- The value of a spent accumulator committing to the coins already spent by the account
- The value of a nullifier accumulator containing the nullifier that created this `AcctState`

The spent accumulator uses the `CoinIDOnChain` to identify coins. [1]

[1] The purpose of the on-chain coin ID is to reduce the state required to maintain the spent accumulator. In contrast to `CoinID`s, a lexicographically sorted list of `CoinIDOnChain` is ordered by the time of creation. Thus, if the spent accumulator is implemented as a sorted Merkle tree, the order of `CoinIDOnChain` elements allows the manager of the accumulator to forget old subtrees without losing the ability to run $\mathsf{AccM}_{\mathsf{ProveNonMembershipAndInsert}}$. Forgetting old subtrees of the spent accumulator reduces the data that a wallet needs to store and the data that an adversary can obtain when a wallet is compromised.

Nullifiers and AggregateNullifier Nullifiers "nullify" an account state and commit to a Shielded CSV transaction that creates a new account state. They are published on the blockchain, allowing the protocol to ensure that an account state is nullified only once. In Shielded CSV, nullifiers appear only in the form of `AggregateNullifiers`.

`AggregateNullifier` is a data structure that contains a list of Schnorr public keys, a $\mathsf{NISSHAC}$ signature, and the address of the entity that added the aggregate nullifier to the blockchain and collects the fee in Shielded CSV. The aggregate nullifier is valid if the signature verifies for the public keys and the static message "Shielded CSV: state update". Conceptually, each public key nullifies an account state and the corresponding commitment in the $\mathsf{NISSHAC}$ signature commits to a Shielded CSV transaction that creates a new account state.

process_block The `Node::process_block` algorithm is run by every user of Shielded CSV when receiving a new block. It scans the block for aggregate nullifiers and verifies their signatures. Shielded CSV nodes maintain a key-value store that keeps track of all nullifiers encountered in the blockchain so far. For each public key in the aggregate nullifier, `process_block` checks if the public key already exists in the key-value store. If it does, the public key is ignored. Otherwise, the following information is inserted into the key-value store: the public key, the commitment to the transaction retrieved from the $\mathsf{NISSHAC}$ signature, the location of the nullifier in the blockchain, and the fee receiver's address

Ignoring duplicate public keys is crucial because if a duplicate public key were to replace an existing key-value entry in the store, an account state could be nullified multiple times while committing to different transactions. This would be analogous to a double-spend in traditional blockchain designs.

In addition to the key-value store of nullifiers, the user also manages a nullifier accumulator (a ToS-accumulator). After scanning the block, the user appends the set of new nullifiers to the accumulator. As a result, the nullifier accumulator value commits to the sequence of nullifiers added for every block to the accumulator.

Furthermore, the user keeps a record of all historical values of the nullifier accumulator. After updating the nullifier accumulator, its value is appended to the list of historic nullifier accumulator values. This list plays an essential role in the `Node::accept_payment` algorithm.

If a blockchain reorganization occurs that disconnects n blocks, the user executes $\mathsf{ToSAccM}_{\mathsf{RemoveSet}}$ n times on their nullifier accumulator state, trims the list of historic nullifier accumulator values and removes the affected nullifiers from the key-value store. After disconnecting the stale blocks, the user runs `process_block` on each of the newly received blocks.

accept_payment The `Node::accept_payment` algorithm takes a `Coin` and a coin proof as arguments. It verifies the coin with $\mathsf{PCD}_{\mathsf{Verify}}$ and the provided coin proof. However, this only proves that the coin is valid with respect to the nullifier accumulator value contained within the `Coin`, and not necessarily that it is valid with respect to the nullifiers the user obtained from the blockchain through the `Node::process_block` algorithm. This is where the list of historic nullifier accumulator values comes into play: `accept_payment` checks that the nullifier accumulator value of the coin is contained in the list, ensuring that the coin is valid with respect to some set of nullifiers the user computed from the blockchain in the past.

Transaction A `Transaction` consists of

- The *conditional* nullifier accumulator value
- The `AcctStateEssence` to nullify
- The list of `CoinIDs` to spend
- The `AcctStateEssence` to create
- The list of `CoinEssences` to create

Note that the transaction outputs are essences because the corresponding `AcctState` and `Coins` can only be created *after* the transaction has been included in the blockchain. Using essences for the transaction *inputs*, rather than full `AcctState` and `Coin` objects, allows the transaction to be assembled (and signed) before the existence of these inputs is confirmed in the blockchain. The conditional nullifier accumulator value depends on the state of the blockchain: it is typically set to the value of the nullifier accumulator that contains the nullifiers that create all the inputs of the transaction. As we will explore in more detail below when examining the `compliance_predicate`, the purpose of the conditional nullifier accumulator is to protect the user in case a blockchain reorganization erases a transaction input.

Payment To initiate a payment, we assume the sender has an account state that hasn't been nullified and either enough balance to pay the receiver or coins available that pass the `Node::accept_payment` algorithm. The sender obtains the receiver's address and creates a new `CoinEssence` with the specified amount. To prepare for the new account state, the sender generates a fresh Schnorr signature keypair using $\mathsf{Sig}_{\mathsf{KeyGen}}$ and inserts the coins to be spent into the spent accumulator managed for this account. The sender then creates a new `AcctStateEssence` containing the account ID, updated balance, the newly generated public key as the nullifier public key, and the value of the updated spent accumulator.

At this point, the sender has all components to assemble a transaction. So, the sender uses the signing key corresponding to the nullifier public key of the current account state to run NISSHAC's $\mathsf{Sig}_{\mathsf{sign}}$ algorithm with the message "Shielded CSV: state update" while committing to the transaction. The public key and signature are then aggregated into a `AggregateNullifier` and posted to the blockchain (which is typically done by a publisher) to nullify the account state and finalize the transaction. Once the nullifier is included in the blockchain, all the data is available to construct `AcctState` and `Coin`s objects from the `AcctStateEssence` and `CoinEssence`s created in the transaction.

The final step is to create a coin proof. The sender creates a PCD transcript that takes the previous `AcctState` and `Coin`s as inputs, outputs the new `AcctState` and `Coin`s, and provides the correct local inputs $\mathsf{w}_{\mathsf{loc}}$ for the transcript to comply with the predicate. Finally, the sender runs $\mathsf{PCD}_{\mathsf{Prove}}$ using the transcript to generate a coin proof for the receiver.

compliance_predicate The `compliance_predicate` used in the PCD scheme plays a crucial role as it encodes many of the protocol's key rules.

We begin by discussing a simplified variant of the compliance predicate that doesn't handle fee payments. This variant is less complex than the one actually used in Shielded CSV. It consists of two predicates that are chosen based on the number of incoming edges: the issuance predicate when there are no incoming edges, and the payment predicate otherwise. We do not specify the issuance predicate as it heavily depends on the actual asset being issued. Edges are labeled either by an `AcctState` or by a `Coin`. In addition to the incoming edges, Shielded CSV vertices also receive local witnesses `w_loc` as input. These include, in particular, the current blockchain's nullifier accumulator value `w_loc.nullifier_accum` and the commitment to the transaction `w_loc.nullifier_tx_comm` as it appears in the `AggregateNullifier`.

The simplified payment predicate verifies certain conditions for a vertex in the PCD transcript. These conditions include:

- Either there is a single incoming edge labeled by an `AcctState` and the rest of the edges are labeled by a `Coin`, or there is no incoming `AcctState` edge and there is at least one incoming `Coin` edge. The latter scenario occurs when creating a new account.
- One of its outgoing edges is labeled by an `AcctState`. Any other outgoing edges, if present, must carry a `Coin` label.
- The sum of the account balance and coin amounts of the incoming edges equals the sum of the balance and amounts of the outgoing edges.
- None of the incoming coins were spent previously by the account, and these coins are correctly inserted into the new account state's spent accumulator. This is verified using the algorithm $\mathsf{AccV}_{\mathsf{VerifyNonMembershipAndInsert}}$.
- The commitment `w_loc.nullifier_tx_comm` commits to the transaction as verified via $\mathsf{Sig}_{\mathsf{CommVerify}}$.
- Exactly one of the following two conditions must be met:
  - The tuple represented by the conditional nullifier accumulator value contains an element distinct from the element at the same position in the tuple represented by `w_loc.nullifier_accum`. This is verified using the $\mathsf{ToSAccV}_{\mathsf{VerifyDistinctElement}}$ algorithm. In this case, the committed transaction is largely ignored, and the predicate allows only one outgoing edge: an `AcctState` with the same balance and spent accumulator as the incoming `AcctState`.
  - All tuples represented by the conditional nullifier accumulator value and the nullifier accumulator values of the incoming `AcctState` and `Coin`s are a prefix of the tuple represented by `w_loc.nullifier_accum`. This is verified using the $\mathsf{ToSAccV}_{\mathsf{VerifyIsPrefix}}$ algorithm. In this case, the outgoing `AcctState` and `Coin` edges allowed by the predicate must correspond to the transaction.

- The nullifier accumulator values of the outgoing `AcctState` and `Coin` edges are equal to value `w_loc.nullifier_accum`.
- The nullifier public key of the incoming account state and the commitment to the transaction `w_loc.nullifier_tx_comm` are committed in `w_loc.nullifier_accum`. This is verified using the $\mathsf{ToSAccV}_{\mathsf{VerifyUnionMembership}}$ algorithm.

As a result, the compliance predicate ensures: (i) that the coin was not created out of thin air, and (ii) that the recipient of the coin can spend it if it passes the `Node::accept_payment` algorithm and no subsequent blockchain reorganization occurs. If a blockchain reorganization takes place after the recipient has spent the coin, and the transaction's conditional nullifier accumulator contains the nullifier that created the coin (which is typical for conditional nullifier accumulators), two possible scenarios arise, depending on which blocks are affected by the reorganization:

- The blockchain reorganization only affects blocks in the chain after the block containing the nullifier that created the coin. In this case, for tuple $\boldsymbol{\mathcal{X}}$ represented by the conditional nullifier accumulator value and tuple $\boldsymbol{\mathcal{X}}'$ represented by the current blockchain's nullifier accumulator value (which is `w_loc.nullifier_accum` in the compliance predicate) it holds that $\mathsf{IsPrefix}(\boldsymbol{\mathcal{X}}, \boldsymbol{\mathcal{X}}') = 1$, as if no blockchain reorganization occurred. Consequently, the user can generate proofs for outgoing edges corresponding to the transaction.
- The blockchain reorganization affects the block containing the nullifier that created the coin or any of the preceding blocks in the chain. In this scenario, for tuple $\boldsymbol{\mathcal{X}}$ represented by the conditional nullifier accumulator value and tuple $\boldsymbol{\mathcal{X}}'$ represented by the current blockchain's nullifier accumulator value, it holds that $\mathsf{DistinctElement}(\boldsymbol{\mathcal{X}}, \boldsymbol{\mathcal{X}}') = 1$. As a result, the user can generate a proof only for a single outgoing `AcctState` that is identical to the incoming account state, except for an updated nullifier public key. If the reorganization retains all the transactions the coin depends on and the user receives an updated coin proof, they can attempt to spend the coin again in a subsequent transaction.

To enable fee payments to the entity posting the nullifier to the blockchain, the compliance predicate used in Shielded CSV is more complex. The payment predicate is actually composed of three predicates that are selected based on the label of the incoming edges: `payment_init`, `payment_finalize`, and `payment_finalize_fee`. The sender creates the transaction as before but reduces the output account balance by the fee amount. Before the nullifier is posted to the blockchain, the sender collects the inputs necessary to satisfy the `payment_init` predicate. This predicate is identical to the payment predicate in the simplified compliance predicate – except that the nullifier accumulator, where the nullifier will be included, is not yet known. Therefore, the last two steps of the simplified payment predicate are omitted.

The `payment_init` vertex results in two outgoing edges: one labelled `PaymentInitOutput` and one labelled `PaymentInitOutputFee`. The sender produces `PaymentInitOutputFee` and a corresponding PCD proof. Both are sent to the publisher, in addition to the nullifier public key and the signature. The publisher is guaranteed that they will be able to satisfy the `payment_finalize_fee` predicate to generate a `Coin` of the fee amount, given they are the first to publish the nullifier on the blockchain. Similarly, once the nullifier is included as part of an aggregate nullifier on the blockchain, the nullifier accumulator containing the nullifier enables the sender to satisfy the `payment_finalize` predicate to produce the updated `AcctState`, the receiver's `Coin`, and the corresponding proofs.

An exemplary PCD transcript of Shielded CSV, illustrating the vertex and edge labels, is shown in Fig. 3.

Publishing Publishers receive a `PaymentInitOutputFee` data structure from senders, which contains a Schnorr public key, the commitment to the transaction R, and the actual fee amount. The sender also provides a PCD proof of correctness for `PaymentInitOutputFee` and the s-value of a Schnorr signature. The publisher constructs a Schnorr signature as (R, s) and runs the $\mathsf{Sig}_{\mathsf{Verify}}$ algorithm with the public key and message "Shielded CSV: state update". After verifying the PCD proof using the $\mathsf{PCD}_{\mathsf{Verify}}$ algorithm, the publisher checks if the fee amount matches expectations.

The publisher continues to collect `PaymentInitOutputFee` and s-values, aggregating the resulting signatures using the $\mathsf{Sig}_{\mathsf{AggregateSig}}$ algorithm. Next, the publisher creates a fresh address for their account ID. Finally, the publisher generates a aggregate nullifier composed of the Schnorr public keys, the $\mathsf{NISSHAC}$ signature, and the fresh address. Once the aggregate nullifier is posted to the blockchain, the publisher can create a new `Coin` containing the fee amount for the fresh address. This `Coin` satisfies the

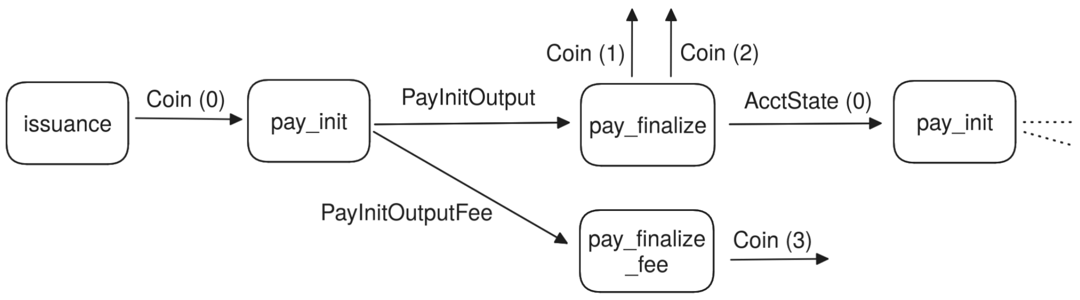

Figure 3: This figure shows an exemplary PCD transcript of Shielded CSV. Witnesses $\mathsf{w}_{\mathsf{loc}}$ are omitted for brevity. The vertex label shows the compliance predicate that must be satisfied for the given incoming edge types. The number in parenthesis is an exemplary account ID. In this PCD transcript, account 0 receives coins in an issuance, and creates a transaction that pays account 1 and account 2. The transaction fee can be claimed by account 3. Note that account 0 was freshly created in the transaction; if the account already existed, the account state would have to be input to `pay_init`.

`payment_finalize_fee` predicate, which allows the publisher to run the $\mathsf{PCD}_{\mathsf{Prove}}$ algorithm to generate the corresponding coin proof.

Account Creation To create a new account, the user first generates a new account ID by running $\mathsf{Sig}_{\mathsf{KeyGen}}$. Next, the user then creates an address, and receives one or more `Coin`s. At some point, the user can spend the `Coin`s and simultaneously create the first `AcctState`.

It's crucial that the user cannot choose the first nullifier public key. This key must be deterministically generated from the account ID. Otherwise, the user could successfully double-spend coins by creating multiple chains of `AcctState`s for a single account ID, with each chain having a different spent accumulator. In Shielded CSV, the first nullifier public key is simply the account ID.

5 Advanced Spending Policies

Thus far, our focus has been on basic transactions between a single sender and receiver. In this section, we explore additional spending policies offered by Shielded CSV. These policies can serve as the foundation for more complex protocols built atop Shielded CSV, such as atomic swaps (discussed in Appendix A.1.2).

5.1 Shared Accounts

Shielded CSV natively supports "t-of-n" shared ownership of accounts. This means that for a shared account with n owners, a transaction requires signatures from t owners to be valid. This is achieved using multi-signatures or threshold signatures compatible with Schnorr signatures, such as MuSig2 [18] or FROST [19]. Additionally, the signature scheme must support sign-to-contract to allow committing to the transaction within the signature. We believe this feature can be easily added to both MuSig2 and FROST.

To create a shared account, the prospective owners generate an account ID group element. This can be done using either the non-interactive MuSig2 key aggregation algorithm or FROST's interactive key generation protocol. The shared account is funded like any other account by generating an address from the account ID and transferring coins to that address.

When creating a transaction, the owners generate a new nullifier public key similar to the account ID. To sign the transaction, they use the MuSig2 or FROST signing protocols with sign-to-contract to create a Schnorr signature that commits to the transaction hash. After the final signature is produced, the nullifier can be published to the blockchain. At this point, any owner who has signed the transaction has sufficient information to generate the coin proof.

5.2 Depending on Unconfirmed Transactions

Certain protocols, such as atomic swaps (Appendix A.1.2), rely on signing transactions that depend on transactions that are unconfirmed. Unconfirmed transactions are those whose nullifiers have not been posted to the blockchain. A signature on a transaction depending on unconfirmed transactions gives the signature holder the ability to create the corresponding nullifier and post it to blockchain once the dependencies are confirmed.

While Shielded CSV supports signing transactions depending on unconfirmed transactions, it does so in a restrictive manner. The provided mechanism is secure for the atomic swap protocol explained in Appendix A.1.2, but it may lead to the loss of coins in other scenarios. This is because such Shielded CSV transactions behave differently from regular transactions during blockchain reorganizations. All transactions include the conditional nullifier accumulator value and, therefore, depend on the blockchain's state. For transactions with confirmed dependencies, this value is set to a nullifier accumulator containing all the dependencies. However, when dependencies are unconfirmed, the value is unknown.

Consequently, the following scenario can occur:

- Before signing the transaction, the signer sets the transaction's conditional nullifier accumulator value to the latest known value.
- Later, all nullifiers of the dependencies and the dependent transaction are posted to the blockchain.
- Subsequently, the blockchain reorganizes to a chain that does not contain the dependencies. This new blockchain may yield a nullifier accumulator that lacks different elements compared to the conditional nullifier accumulator.

Thus, it becomes impossible to prove that the dependent transaction is a no-op. Furthermore, since the account owner committed to a transaction whose dependencies no longer exist, they cannot create valid proofs for the account update, effectively burning the account balance.

Nevertheless, if the holder of the dependent transaction's nullifier can ensure that none of the dependencies are double-spent and can create new coin proofs for the dependencies, they can repost the dependencies to the new blockchain. In this case, the holder can create valid proofs for the dependent transaction as if the blockchain reorganization hadn't occurred.

6 Discussion

The following discussion highlights several aspects of Shielded CSV that have significant implications for its implementation and use. In particular, this section explores potential concrete instantiations of the Protocol's PCD primitive.

6.1 Communication Channels

A key component of Shielded CSV is the transfer of the coin and coin proof from the payment sender to the recipient. While Shielded CSV conceals the transaction history from blockchain observers, an entity able to intercept messages in the communication channels between senders and receivers could potentially reconstruct the transaction history. Therefore, it is crucial to use a one-way communication channel that encrypts messages to the intended recipient.

If the communication channel leaks metadata to an adversary, such as information about who is sending messages to whom, then the adversary can infer the flow of payments. Consequently, we require a communication channel for coin and coin proof transmission that does not leak additional metadata.

To illustrate potential scenarios, we present some common examples:

Sender and recipient are in close physical proximity: Communication can occur via QR codes or similar techniques.

Recipient is a merchant operating a web store: If the sender is purchasing an item from a web store, the same communication channel (provided it is authenticated and encrypted) can be used to submit the coin proof directly to the merchant's store. This does not leak significant metadata beyond what is already disclosed by visiting the store.

Sender and recipient regularly share messages via end-to-end encrypted instant messaging: The sender can transmit the coin and coin proof using the existing instant messaging channel to the recipient. This does not leak significant metadata beyond what has already been incurred through instant messaging.

Recipient publishes a static donation address: This case is more challenging than the others, as there is no existing metadata connecting the sender and recipient, and the recipient may not be online. The sender could upload the coin and coin proof to a server from which the recipient can retrieve it later. However, this should be implemented in a way that allows the recipient to retrieve the data privately and minimizes the metadata obtained by the server. This scenario requires further research and is considered future work.

6.2 Wallet State

Unlike traditional blockchains such as Bitcoin, where a wallet can be restored from a secret seed and public transaction data, Shielded CSV does not allow for such straightforward recovery. This is because the account state depends on data that is not available to blockchain observers, including coins spent and created by the account. Moreover, recreating a proof for the current account state requires the coin proofs of all spent coins, which generally cannot be created by the account owner. Consequently, if the wallet state is lost, the account balance becomes irretrievable.

Therefore, users must maintain the following:

- The latest account state, including the data necessary to produce non-membership and insertion proofs for the spent accumulator and the secret key corresponding to the account state's nullifier public key
- The associated account state proof
- Coin proofs of all unspent coins
- Randomness of the published addresses

As noted in Section 4, certain implementations of the spent accumulator do not require storing all elements ever inserted to allow proving the non-membership of newly received coins.

6.3 Coin Linkability

Coins generated within the same Shielded CSV transaction are linkable by their recipients. If recipients communicate with each other, they can determine that they were paid in the same transaction. This is due to the fact that coins in Shielded CSV contain both the transaction hash and the location of the nullifier that created them.

A potential privacy issue arising from this linkability can be illustrated by the following scenario: Suppose a sender creates a transaction that generates two coins: one for a recipient who knows the sender's identity (e.g., a shop requiring a shipping address), and another for a different recipient. If both receivers collude or if they are compromised, they could link the sender's identity to the second coin, thereby identifying the sender of that coin.

Given this potential for compromised privacy, Shielded CSV users are advised to create multiple coins in a single transaction only when coin linkability does not pose a concern.

6.4 Instantiation

As Shielded CSV is defined over an abstract PCD scheme, it can be instantiated using a number of different concrete constructions. Numerous approaches exist for instantiating a PCD scheme, including recursive SNARKs [20], and accumulation [16] and folding schemes [21]. Within these broad categories, further degrees of freedom are available. For instance, many different proof systems can be used to construct recursive SNARKs, and there are various folding schemes one could choose from. Each choice will yield distinct tradeoffs.

However, it is crucial to emphasize that both recursive SNARKs and folding schemes can be constructed without a trusted setup, while maintaining concrete efficiency. Although questions remain regarding the optimal implementation, there is no question that this is feasible today. Two mature techniques are particularly noteworthy: recursive STARKs [22] (or more broadly, code-based SNARKs) and folding schemes instantiated over an Inner Product Argument (IPA) such as Bulletproofs [23].

This section provides a high-level description of these two approaches and outlines their respective tradeoffs. Additionally, we introduce two novel approaches that aim to unify the best properties of recursive STARKs and folding schemes: LatticeFold [24] and Accumulation without Homomorphism [25]. While these schemes are not yet suitable for implementation, they show considerable promise.

6.4.1 Recursive STARKs

Of the two approaches, recursive STARKs represent the more mature technology, with libraries like Plonky2 [26] seeing widespread real-world use. STARKs [2] offer several desirable properties, including state-of-the-art prover performance, relatively simple and performant recursive constructions, [3], and plausible quantum resistance. The primary drawbacks of STARKs are their size, ranging from tens to hundreds of kilobytes, and their relatively complex security arguments. While optimizing for either proof size or provable security is feasible, simultaneously optimizing for both can make the security analysis of concrete systems challenging.

In many applications, for example Zcash, where users must generate proofs themselves over private data and proofs are put on a public blockchain, the large proof sizes of STARKs pose a serious problem. Replacing Halo2 with Plonky2, for example, would increase the amount of Zcash proof data by at least an order of magnitude. However, in the client-side validated setting, we are able to almost entirely sidestep this problem. The proofs associated with the PCD graph are only ever shared between parties to a payment, and in particular are never put on the blockchain. Thus, using a proof system with large proofs does not substantially affect the performance of the overall system.

Construction To instantiate PCD with recursive STARKs, we follow the basic recursive SNARK recipe originally outlined in [20]. In this construction, the proof of an outgoing edge of a vertex demonstrates knowledge of satisfying inputs to the predicate, while also recursively proving knowledge of valid proofs for each incoming edge of the vertex. Various practical considerations notwithstanding, this is basic recipe is how all recursive STARK constructions work today.

The soundness of this technique is somewhat subtle, since proof recursion in the random oracle model of superlogarithmic depth is not provably sound. This is because proof recursion necessitates instantiating the random oracle, and to prove soundness, the extractor must handle a superpolynomial number of transcripts, rendering it inefficient. There are a number of papers exploring this problem [29], as well as alternative approaches to proving soundness that avoid the random oracle model and its associated theoretical issues. However, current evidence does not suggest that these theoretical concerns pose significant practical risks.

6.4.2 Folding Schemes

Folding schemes are an alternative approach to proof recursion that evolved out of the initial work of Halo [30]. The authors of Halo sought to recursively compose Bulletproofs, addressing the fundamental challenge that the Bulletproof verifier has linear complexity in the statement to prove, rendering naïve proof recursion ineffective. Halo demonstrates that instead it is possible to exploit the structure of the inner product argument to combine proofs and defer the expensive part of the Bulletproof verifier. This approach fundamentally takes advantage of the structure of Bulletproofs as a type of sumcheck [31] and the expensive part of the verifier as a group multilinear extension evaluation.

Following Halo, which [16] classifies as an atomic accumulation scheme, subsequent works progressively reduced the computational burden. The inner product argument was entirely eliminated, leading to the "accumulation" of NARKs rather than SNARKs. Folding schemes, especially those based on Nova [21] like HyperNova [32] and those based on ProtoStar [33] including ProtoGalaxy [34], represent the state of the art in this field. Unlike SNARKs, which typically employ two algorithms, folding schemes utilize three: prove, folding verify, and decide. The folding verification algorithm simply checks that two "instances" were correctly folded, not that the instances themselves were valid. This folding step essentially entails taking a random linear combination of two instances. The decider algorithm then verifies the validity of the final accumulated instance. If valid, all folded instances must have been valid as well.

Construction To instantiate PCD with a folding scheme, we define several terms in the style of ProtoGalaxy. A folding scheme is defined with respect to a relation $\mathcal{R}$ which consists of pairs of an instance and witness (ϕ, ω). The instance ϕ commits to the witness ω.

A folding scheme then defines a relaxed relation for $\mathcal{R}$, denoted as $\mathcal{R}^*$, which also consists of instance-witness pairs. Instances for $\mathcal{R}^*$ are referred to as accumulators. For a more detailed explanation, we refer

[2]Technically, many proof systems commonly referred to as "STARKs", such as Plonky2, are not technically STARKs since they do not meet the poly$(\log n)$ preprocessing requirement. For clarity, we will use "STARKs" to refer to all code-based SNARKs, even if they do not technically meet the formal definition, and will not refer to recursive, uniform IPA based proofs as STARKs even though they technically meet the definition. This reflects common usage of the term.

[3]See the history of NOVA on a cycle of curves [27] and CycleFold [28].

the reader to [34]. The scheme allows us to "fold" any number of elements of $\mathcal{R}$ and $\mathcal{R}^*$ to produce a new element of $\mathcal{R}^*$. In a folding-based PCD construction, each proof associated with an edge in the PCD graph is an element of $\mathcal{R}^*$. This element is the folding of the accumulators for incoming edges and a NARK for the edge's predicate ϕ. By the properties of the folding scheme, if the NARK for an edge is in $\mathcal{R}^*$ then the graph must be ϕ-compliant.

Folding schemes can be instantiated using any linearly homomorphic commitment scheme. There are two natural choices for this: Pedersen commitments and KZG commitments [35]. Unlike KZG, Pedersen commitments do not require a trusted setup. While there are numerous folding schemes with various tradeoffs, a detailed comparison is beyond the scope of this paper.

6.4.3 New Directions

Recursive STARKs and folding schemes each possess distinct desirable properties, prompting the question of whether a single scheme could capture the advantages of both. Specifically, can a single scheme combine the post-quantum resistance and fast prover times of recursive STARKs with the low recursion overhead of folding schemes? At the time of writing, the answer remains unclear in practice, although two promising approaches have emerged.

The first approach is lattice-based folding schemes like LatticeFold [24], which exploit the partially linearly homomorphic properties of lattice-based cryptography to instantiate a post-quantum folding scheme. The second approach, Accumulation without Homomorphism (AwoH) [25], generalizes the linear combination check of folding schemes, allowing the combination of folding scheme techniques with non-linear Merkle commitments. While neither scheme has been implemented, and the latter faces significant practical challenges, both show promise for instantiating PCD for our purposes.

These techniques converge towards a general form similar to that of a folding scheme. Provers pass around NARKs, potentially with auxiliary information to aid in deciding, and accumulate NARKs by taking random linear combinations. However, this random linear combination step must be generalized, as neither lattices nor AwoH directly support taking unbounded linear combinations of commitments. In the case of lattices, the prover must renormalize the committed vectors to ensure their norm remains sufficiently small for the commitment scheme to be binding. For Merkle trees, the prover can take commitments to the inputs and the result of the linear combination and perform sufficient spot checks to be convinced the result is well formed. For technical reasons, the latter can only be recursively performed a logarithmic number of times, rendering it impractical at present.

References

[1] Maxim Orlovsky et al. *RGB: Turing-complete, Scalable & Confidential Smart Contract Layer for Bitcoin & LN.* https://blackpaper.rgb.tech/. 2023.

[2] Olaoluwa Osuntokun. *TAP: Taproot Assets Protocol.* https://github.com/Roasbeef/bips/blob/bip-tap/bip-tap.mediawiki. 2021.

[3] Erik Rybakken, Leona Hioki, and Mario Yaksetig. "Intmax2: A ZK-rollup with Minimal Onchain Data and Computation Costs Featuring Decentralized Aggregators". In: *Cryptology ePrint Archive* (2023). https://eprint.iacr.org/2023/1082.pdf.

[4] Daira Hopwood et al. *Zcash protocol specification.* https://zips.z.cash/protocol/protocol.pdf.

[5] Peter Todd. *Disentangling Crypto-Coin Mining: Timestamping, Proof-of-Publication, and Validation.* https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2013-November/003714.html. 2013.

[6] Robin Linus. *zkCoins: A payment system with strong privacy and scalability.* https://gist.github.com/RobinLinus/d036511015caea5a28514259a1bab119. 2023.

[7] Bitcoin Wiki. *Blockchain attacks on privacy.* https://en.bitcoin.it/wiki/Privacy#Blockchain_attacks_on_privacy.

[8] Robin Linus et al. "BitVM2: Bridging Bitcoin to Second Layers". In: (2024). https://bitvm.org/bitvm_bridge.pdf.

[9] Peter Todd. *Building Blocks of the State Machine Approach to Consensus.* https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2016-June/012773.html. 2016.

[10] Andrew Poelstra et al. "Confidential Assets". In: *FC 2018 Workshops.* Ed. by Aviv Zohar et al. Vol. 10958. LNCS. Springer, Berlin, Heidelberg, Mar. 2019, pp. 43–63. DOI: 10.1007/978-3-662-58820-8_4.

[11] Joseph Poon and Thaddeus Dryja. *The bitcoin lightning network: Scalable off-chain instant payments.* 2016.

[12] Peter Todd. *Why Scaling Bitcoin With Sharding Is Very Hard.* https://petertodd.org/2015/why-scaling-bitcoin-with-sharding-is-very-hard. 2015.

[13] Konstantinos Chalkias et al. "Non-interactive Half-Aggregation of EdDSA and Variants of Schnorr Signatures". In: *CT-RSA 2021.* Ed. by Kenneth G. Paterson. Vol. 12704. LNCS. Springer, Cham, May 2021, pp. 577–608. DOI: 10.1007/978-3-030-75539-3_24.

[14] Alessandro Chiesa and Eran Tromer. "Proof-Carrying Data and Hearsay Arguments from Signature Cards". In: *ICS 2010.* Ed. by Andrew Chi-Chih Yao. Tsinghua University Press, Jan. 2010, pp. 310–331.

[15] Foteini Baldimtsi, Ran Canetti, and Sophia Yakoubov. "Universally Composable Accumulators". In: *CT-RSA 2020.* Ed. by Stanislaw Jarecki. Vol. 12006. LNCS. Springer, Cham, Feb. 2020, pp. 638–666. DOI: 10.1007/978-3-030-40186-3_27.

[16] Benedikt Bünz et al. "Proof-Carrying Data Without Succinct Arguments". In: *CRYPTO 2021, Part I.* Ed. by Tal Malkin and Chris Peikert. Vol. 12825. LNCS. Virtual Event: Springer, Cham, Aug. 2021, pp. 681–710. DOI: 10.1007/978-3-030-84242-0_24.

[17] Ilja Gerhardt and Timo Hanke. "Homomorphic payment addresses and the pay-to-contract protocol". In: *arXiv preprint arXiv:1212.3257* (2012).

[18] Jonas Nick, Tim Ruffing, and Yannick Seurin. "MuSig2: Simple Two-Round Schnorr Multi-signatures". In: *CRYPTO 2021, Part I.* Ed. by Tal Malkin and Chris Peikert. Vol. 12825. LNCS. Virtual Event: Springer, Cham, Aug. 2021, pp. 189–221. DOI: 10.1007/978-3-030-84242-0_8.

[19] Chelsea Komlo and Ian Goldberg. "FROST: Flexible Round-Optimized Schnorr Threshold Signatures". In: *SAC 2020.* Ed. by Orr Dunkelman, Michael J. Jacobson Jr., and Colin O'Flynn. Vol. 12804. LNCS. Springer, Cham, Oct. 2020, pp. 34–65. DOI: 10.1007/978-3-030-81652-0_2.

[20] Eli Ben-Sasson et al. "Scalable Zero Knowledge via Cycles of Elliptic Curves". In: *CRYPTO 2014, Part II.* Ed. by Juan A. Garay and Rosario Gennaro. Vol. 8617. LNCS. Springer, Berlin, Heidelberg, Aug. 2014, pp. 276–294. DOI: 10.1007/978-3-662-44381-1_16.

[21] Abhiram Kothapalli, Srinath Setty, and Ioanna Tzialla. "Nova: Recursive Zero-Knowledge Arguments from Folding Schemes". In: *CRYPTO 2022, Part IV.* Ed. by Yevgeniy Dodis and Thomas Shrimpton. Vol. 13510. LNCS. Springer, Cham, Aug. 2022, pp. 359–388. DOI: 10.1007/978-3-031-15985-5_13.

[22] Eli Ben-Sasson et al. "Scalable Zero Knowledge with No Trusted Setup". In: *CRYPTO 2019, Part III.* Ed. by Alexandra Boldyreva and Daniele Micciancio. Vol. 11694. LNCS. Springer, Cham, Aug. 2019, pp. 701–732. DOI: 10.1007/978-3-030-26954-8_23.

[23] Benedikt Bünz et al. "Bulletproofs: Short Proofs for Confidential Transactions and More". In: *2018 IEEE Symposium on Security and Privacy.* IEEE Computer Society Press, May 2018, pp. 315–334. DOI: 10.1109/SP.2018.00020.

[24] Dan Boneh and Binyi Chen. *LatticeFold: A Lattice-based Folding Scheme and its Applications to Succinct Proof Systems.* Cryptology ePrint Archive, Report 2024/257. 2024. URL: https://eprint.iacr.org/2024/257.

[25] Benedikt Bünz et al. *Accumulation without Homomorphism.* Cryptology ePrint Archive, Report 2024/474. 2024. URL: https://eprint.iacr.org/2024/474.

[26] Polygon Zero Team. *Plonky2: Fast Recursive Arguments with PLONK and FRI.* https://github.com/0xPolygonZero/plonky2/blob/main/plonky2/plonky2.pdf. 2022.

[27] Wilson Nguyen, Dan Boneh, and Srinath Setty. *Revisiting the Nova Proof System on a Cycle of Curves.* Cryptology ePrint Archive, Report 2023/969. 2023. URL: https://eprint.iacr.org/2023/969.

[28] Abhiram Kothapalli and Srinath Setty. *CycleFold: Folding-scheme-based recursive arguments over a cycle of elliptic curves.* Cryptology ePrint Archive, Report 2023/1192. 2023. URL: https://eprint.iacr.org/2023/1192.

[29] Hyeonbum Lee and Jae Hong Seo. *On the Security of Nova Recursive Proof System.* Cryptology ePrint Archive, Report 2024/232. 2024. URL: https://eprint.iacr.org/2024/232.

[30] Sean Bowe, Jack Grigg, and Daira Hopwood. *Halo: Recursive Proof Composition without a Trusted Setup.* Cryptology ePrint Archive, Report 2019/1021. 2019. URL: https://eprint.iacr.org/2019/1021.

[31] Jonathan Bootle, Alessandro Chiesa, and Katerina Sotiraki. "Sumcheck Arguments and Their Applications". In: *CRYPTO 2021, Part I.* Ed. by Tal Malkin and Chris Peikert. Vol. 12825. LNCS. Virtual Event: Springer, Cham, Aug. 2021, pp. 742–773. DOI: 10.1007/978-3-030-84242-0_26.

[32] Abhiram Kothapalli and Srinath T. V. Setty. "HyperNova: Recursive Arguments for Customizable Constraint Systems". In: *CRYPTO 2024, Part X.* Ed. by Leonid Reyzin and Douglas Stebila. Vol. 14929. LNCS. Springer, Cham, Aug. 2024, pp. 345–379. DOI: 10.1007/978-3-031-68403-6_11.

[33] Benedikt Bünz and Binyi Chen. *ProtoStar: Generic Efficient Accumulation/Folding for Special Sound Protocols.* Cryptology ePrint Archive, Report 2023/620. 2023. URL: https://eprint.iacr.org/2023/620.

[34] Liam Eagen and Ariel Gabizon. *ProtoGalaxy: Efficient ProtoStar-style folding of multiple instances.* Cryptology ePrint Archive, Report 2023/1106. 2023. URL: https://eprint.iacr.org/2023/1106.

[35] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. "Constant-Size Commitments to Polynomials and Their Applications". In: *ASIACRYPT 2010.* Ed. by Masayuki Abe. Vol. 6477. LNCS. Springer, Berlin, Heidelberg, Dec. 2010, pp. 177–194. DOI: 10.1007/978-3-642-17373-8_11.

[36] Bitcoin Optech. *Hash Time Locked Contract (HTLC).* https://bitcoinops.org/en/topics/htlc/. 2024.

[37] Maurice Herlihy. "Atomic Cross-Chain Swaps". In: *37th ACM PODC.* Ed. by Calvin Newport and Idit Keidar. ACM, July 2018, pp. 245–254. DOI: 10.1145/3212734.3212736.

[38] Bitcoin Optech. *Point Time Locked Contracts (PTLCs).* https://bitcoinops.org/en/topics/ptlc/. 2024.

A Appendix

A.1 Extensions

In this section, we present extensions to Shielded CSV. First, we introduce an extension that supports time-locked transactions, followed by an example of an atomic swap protocol that leverages this feature. Subsequently, we outline an extension of Shielded CSV to accommodate multiple assets. Finally, we demonstrate how modifications to the compliance predicate can lead to reduced proving costs.

A.1.1 Time-locked Transactions

A time-locked transaction with *locktime* $h \in \mathbb{N}$ is only valid if the corresponding nullifier is included in a block at height h or above. While Shielded CSV as described in Section 4 does not support time-locks, several methods could be implemented to add this functionality. We outline the conceptually simplest method, which requires the following modifications to Shielded CSV:

- Add an optional locktime field to the nullifier.
- If locktime h is present in a nullifier, the signature covers the message "Shielded CSV: state update h".
- Exclude nullifiers with a locktime less than the current blockchain height from insertion into the nullifier accumulator.

As a result, it is impossible to create a coin proof for a time-locked transaction, unless its nullifier is included in a block where the locktime is expired. A drawback of this approach is that it increases the size of nullifiers for time-locked transactions and, therefore, more data needs to be written to the blockchain. Additionally, the public visibility of the locktime may reduce user privacy.

A blockchain reorganization may place a nullifier into a block where its locktime has not yet expired. Since this nullifier is ignored by Shielded CSV users, it can be resubmitted to the blockchain once the locktime expires in the new chain.

A.1.2 Atomic Swaps

A basic atomic swap [36, 37] is a protocol that enables two parties, each possessing a coin, to securely exchange their respective coins. The critical feature of an atomic swap is its all-or-nothing nature: either the swap occurs completely, or not at all, preventing any party from obtaining the other's coin without transferring their own. This section outlines an atomic swap between coins on Shielded CSV, extended with time-locked transactions (Appendix A.1.1), and another protocol, such as a blockchain or a "Layer 2" system.

For this atomic swap, we require that the other protocol supports *Point Time Locked Contracts* (PTLCs) [38]. A PTLC for parties A and B, group element P (typically an elliptic curve *point*), and locktime h locks a coin such that it can be spent in two ways: either by party A after providing an integer x to party B such that $xG = P$, where G is the group generator, or by party B after the locktime h has expired. We further require that the group element P must belong to the same group $\mathbb{G}$ used by Shielded CSV Schnorr signatures.

For simplicity, we describe an atomic swap with Bitcoin, which supports PTLCs. The swap involves two parties: Alice, who possesses coins on Shielded CSV, desires bitcoin, and has an ephemeral Schnorr key pair $(a, A) \in \mathbb{Z}_p \times \mathbb{G}$. Bob, who owns bitcoin, wants Shielded CSV coins, and has an ephemeral Schnorr key pair (b, B). The protocol proceeds as follows:

1. Alice creates a Shielded CSV transaction with the coins she wants to swap that funds a 2-of-2 account between Alice and Bob, which requires a signature for the aggregate public key of A and B to update. She neither publishes the corresponding nullifier nor signs the transaction. Alice also creates a time-locked recovery transaction to return the funds from the shared account to herself, with the locktime expiring at block height h_2. Alice then sends the recovery transaction to Bob.
2. Bob verifies the recovery transaction and creates a MuSig2 partial signature using his signing key b, which he sends to Alice.
3. Alice verifies Bob's MuSig2 partial signature and aggregates it with her own signature into a single Schnorr signature, authorizing her time-locked recovery from the shared account. She then signs the 2-of-2 account funding transaction, publishes the nullifier, and sends the coin proof to Bob.
4. Bob verifies the coin proof. He then deposits bitcoin into a Point Time Locked Contract (PTLC) that can be spent by Alice providing preimage a or by Bob after block h_1 (where $h_1 < h_2$).
5. Alice claims the on-chain bitcoin by creating a transaction that publishes a and spends the PTLC.
6. Bob learns key a, giving him full control over the shared Shielded CSV account and allowing him to spend the funds.

If Bob does not create the on-chain PTLC, Alice can recover her funds at block height h_2 using the recovery transaction. If Bob creates the PTLC but Alice does not claim it, Bob can recover the funds at block height h_1. Thus, the described swap is atomic.

As discussed in Section 5.2, we must analyze the impact of blockchain reorganizations on Alice's recovery transaction, as it depends on the unconfirmed funding transaction. Alice sets the conditional nullifier accumulator value of her recovery transaction to match that of her funding transaction. A blockchain reorganization may occur without affecting the conditional nullifier accumulator. If the new blockchain contains the funding nullifier, Alice posts the recovery nullifier and create new coin proofs if necessary. Otherwise, Alice can either repost the funding nullifier to the blockchain again or double-spend her original funding nullifier by sending the funds directly to herself. Thus, Alice's coins are not burned even in the event of a blockchain reorganization.

A.1.3 Supporting Multiple Assets

While Shielded CSV currently supports only a single asset, extending it to accommodate multiple assets is a straightforward process. This can be achieved by appending an integer asset type field to `CoinEssence` objects, and modifying the `AcctStateEssence` balance to be asset-type specific.

The revised compliance predicate would verify the balance equation for every asset type independently, ensuring that the sum of incoming amounts equals the sum of outgoing amounts for each asset type. It is crucial that the implementation of issuances ensures that new assets are assigned unique asset types that do not collide with any existing asset type.

Future work should address efficient atomic swaps between different asset types within Shielded CSV.

A.1.4 Reducing Proving Cost

In a PCD transcript, every outgoing edge of a vertex requires a separate invocation of the proving algorithm. As a consequence, for every payment in Shielded CSV, users must create two proofs for the `payment_init` vertex and at least two proofs for the `payment_finalize` vertex. By accepting a moderate increase in circuit complexity, we can generically transform any PCD vertex with multiple outgoing edges into a vertex with a single outgoing edge, significantly reducing the proving cost.

The high-level idea is as follows: given a vertex in a PCD transcript, we replace its outgoing edges labeled $z^{(o_1)}, \ldots, z^{(o_n)}$ with a single outgoing edge labeled with an accumulator value $z^{(o)\prime}$, which represents all labels $z^{(o_1)}, \ldots, z^{(o_n)}$. To access an output $z^{(o_i)}$ of the original transcript, we change the compliance predicate to require a membership proof that $z^{(o_i)}$ is in the set represented $z^{(o)\prime}$.

To make this idea more concrete, we define compliance predicate ϕ' that uses the original compliance predicate ϕ such that PCD transcripts only require a single outgoing edge for each vertex:

```
$\phi'(z^{(o)\prime}, \mathsf{w}_{\mathsf{loc}}^{(u)\prime}, z^{(e_1)\prime}, \ldots, z^{(e_m)\prime})$
 1 : $(\mathsf{w}_{\mathsf{loc}}^{(u)}, z^{(e_1)}, \ldots, z^{(e_m)}, \pi_1, \ldots, \pi_m, z^{(o_1)}, \ldots, z^{(o_n)}) \leftarrow \mathsf{w}_{\mathsf{loc}}^{(u)\prime}$
 2 : for $i = 1..m$
 3 :   if $\mathsf{AccV}_{\mathsf{VerifyMembership}}(z^{(e_i)\prime}, z^{(e_i)}, \pi_i) \neq 1$
 4 :     return 0
 5 : $\mathsf{state} \leftarrow \mathsf{AccM}_{\mathsf{New}}()$
 6 : for $i = 1..n$
 7 :   if $\phi(z^{(o_i)}, \mathsf{w}_{\mathsf{loc}}^{(u)}, z^{(e_1)}, \ldots, z^{(e_m)}) \neq 1$
 8 :     return 0
 9 :   $\mathsf{state} \leftarrow \mathsf{AccM}_{\mathsf{Insert}}(\mathsf{state}, z^{(o_i)})$
10 : $v \leftarrow \mathsf{AccM}_{\mathsf{Value}}(\mathsf{state})$
11 : if $v \neq z^{(o)\prime}$
12 :   return 0
13 : return 1
```

We can observe that assuming the accumulator is secure, then for any set $Z = \{z^{(o_1)}, \ldots, z^{(o_n)}\}$ it holds that $\phi'(z^{(o)\prime}, \mathsf{w}_{\mathsf{loc}}^{(u)\prime}, z^{(e_1)\prime}, \ldots, z^{(e_m)\prime}) = 1$ if and only if for all $i = 1..m$ π_i proves that $z^{(e_i)}$ is a member of the set represented by $z^{(e_i)\prime}$, $z^{(o)\prime}$ represents set Z, and $\phi(z^{(o_i)}, \mathsf{w}_{\mathsf{loc}}^{(u)}, z^{(e_1)}, \ldots, z^{(e_m)}) = 1$ for all $z^{(o_i)} \in Z$.

A.2 Instantiating a ToS-accumulator

A ToS-accumulator as defined in Section 3 represents a tuple of sets and is a core component of Shielded CSV. This section describes $\mathsf{ToSAccMT}$, a basic instantiation of a ToS-accumulator using Merkle trees, consisting of accumulator manager $\mathsf{ToSAccMTM}$ and accumulator verifier $\mathsf{ToSAccMTV}$.

We assume that the Merkle trees used by $\mathsf{ToSAccMT}$ provide the following standard management and verification algorithms.

- $\mathsf{MTreeM}_{\mathsf{New}}() \rightarrow \mathsf{state}$ outputs the initial state state of the Merkle tree.
- $\mathsf{MTreeM}_{\mathsf{Value}}(\mathsf{state}) \rightarrow v$ outputs the Merkle tree value ("root") v given a Merkle tree state state.
- $\mathsf{MTreeM}_{\mathsf{Insert}}(\mathsf{state}, x) \rightarrow \mathsf{state}'$ outputs a new state state' after inserting element x into the Merkle tree with state state.
- $\mathsf{MTreeM}_{\mathsf{Remove}}(\mathsf{state}) \rightarrow \mathsf{state}'$ outputs new state state' after removing the last element inserted into the Merkle tree with state state.
- $\mathsf{MTreeM}_{\mathsf{ProveMember}}(\mathsf{state}, x) \rightarrow \pi$ outputs proof π that x is in the Merkle tree with state state.
- $\mathsf{MTreeV}_{\mathsf{New}}() \rightarrow v$ outputs the value v of an empty Merkle tree.

- $\mathsf{MTreeV}_{\mathsf{VerifyMember}}(v, x, \pi) \to b$ outputs $b = 1$ if the proof π proves that x is contained in the Merkle tree with value v. Otherwise, it outputs $b = 0$.

For simplicity, we also assume the existence of certain convenience management algorithms that read the Merkle tree's state. If the Merkle tree does not provide these convenience algorithms, the $\mathsf{ToSAccMT}$ manager can implement them by keeping track of the elements inserted into the Merkle trees.

- $\mathsf{MTreeM}_{\mathsf{GetLeaf}}(\mathsf{state}, i) \to x$ outputs the i-th element x inserted into the Merkle tree with state state.
- $\mathsf{MTreeM}_{\mathsf{IsContained}}(\mathsf{state}, x) \to b$ outputs $b = 1$ if the element x has been inserted into the Merkle tree with state state. Otherwise, it outputs $b = 0$.

Furthermore, we define the following helper algorithm to initialize a Merkle tree with a set:

```
$\mathsf{MTreeM}_{\mathsf{InsertMulti}}(\mathcal{X})$
1 : $\mathsf{state} \leftarrow \mathsf{MTreeM}_{\mathsf{New}}()$
2 : for $x \in \mathcal{X}$
3 :   $\mathsf{state} \leftarrow \mathsf{MTreeM}_{\mathsf{Insert}}(\mathsf{state}, x)$
4 : return $\mathsf{state}$
```

The Merkle tree-based ToS-accumulator $\mathsf{ToSAccMT}$ uses an "outer" Merkle tree whose leaves contain the values of "inner" Merkle trees representing the appended sets. When a new set is appended, $\mathsf{ToSAccMTM}$ creates a new inner Merkle tree, inserts all elements of the set, and then inserts the inner Merkle tree's value into the outer Merkle tree. The state of $\mathsf{ToSAccMTM}$ is the tuple $(\mathsf{stateOut}, n, \mathsf{stateIn}_1, \ldots, \mathsf{stateIn}_n)$ where $\mathsf{stateOut}$ is the state of the outer Merkle tree, n is the number of appended sets, and $\mathsf{stateIn}_i$ is the state of the i-th inner Merkle tree.

The elements contained in the outer Merkle tree are tuples (i, v_0, v_1), where i is the number of elements inserted prior to this tuple incremented by one, v_0 is the value of the outer Merkle tree before the tuple's insertion the tuple, and v_1 is the value of the inner Merkle tree that represents the appended set. In particular, v_1 is used for proving membership in the union of sets, v_0 for proving $\mathsf{IsPrefix}$ and i for proving $\mathsf{DistinctElement}$.

See Fig. 4 for the definition of the $\mathsf{ToSAccMT}$ algorithms.

```
$\mathsf{ToSAccMTM}_{\mathsf{New}}()$
return $(\mathsf{MTreeM}_{\mathsf{New}}(), 0)$

$\mathsf{ToSAccMTM}_{\mathsf{Value}}(\mathsf{state})$
$(\mathsf{stateOut}, \ldots) \leftarrow \mathsf{state}$
return $\mathsf{MTreeM}_{\mathsf{Value}}(\mathsf{stateOut})$

$\mathsf{ToSAccMTM}_{\mathsf{AppendSet}}(\mathsf{state}, \mathcal{X})$
$(\mathsf{stateOut}, n, \mathsf{stateIn}_1, \ldots, \mathsf{stateIn}_n) \leftarrow \mathsf{state}$
$v_0 \leftarrow \mathsf{MTreeM}_{\mathsf{Value}}(\mathsf{stateOut})$
$\mathsf{stateIn}_{n+1} \leftarrow \mathsf{MTreeM}_{\mathsf{InsertMulti}}(\mathcal{X})$
$v_1 \leftarrow \mathsf{MTreeM}_{\mathsf{Value}}(\mathsf{stateIn}_{n+1})$
$\mathsf{stateOut} \leftarrow \mathsf{MTreeM}_{\mathsf{Insert}}(\mathsf{stateOut}, (n+1, v_0, v_1))$
return $(\mathsf{stateOut}, n+1,$
            $\mathsf{stateIn}_1, \ldots, \mathsf{stateIn}_n, \mathsf{stateIn}_{n+1})$

$\mathsf{ToSAccMTM}_{\mathsf{ProveUnionMembership}}(\mathsf{state}, x)$
$(\mathsf{stateOut}, n, \mathsf{stateIn}_1, \ldots, \mathsf{stateIn}_n) \leftarrow \mathsf{state}$
for $i = 1 \ldots n$
  if $\mathsf{MTreeM}_{\mathsf{IsContained}}(\mathsf{stateIn}_i, x) = 1$
    $\pi_1 \leftarrow\!\!\$\ \mathsf{MTreeM}_{\mathsf{ProveMember}}(\mathsf{stateIn}_i, x)$
    $v_1 \leftarrow \mathsf{MTreeM}_{\mathsf{Value}}(\mathsf{stateIn}_i)$
    $v_0 \leftarrow \mathsf{MTreeM}_{\mathsf{GetLeaf}}(\mathsf{stateOut}, i)_2$
    $\pi \leftarrow\!\!\$\ \mathsf{MTreeM}_{\mathsf{ProveMember}}(\mathsf{stateOut}, (i, v_0, v_1))$
    return $(i, v_0, \pi, v_1, \pi_1)$
return $\bot$

$\mathsf{ToSAccMTM}_{\mathsf{ProveIsPrefix}}(\mathsf{state}, \mathsf{state}')$
$(\_, n, \ldots) \leftarrow \mathsf{state}$
$(\mathsf{stateOut}', n', \ldots) \leftarrow \mathsf{state}'$
if $n = n'$
  return $()$
$(\_, v_0, v_1) \leftarrow \mathsf{MTreeM}_{\mathsf{GetLeaf}}(\mathsf{stateOut}', n+1)$
$\pi \leftarrow\!\!\$\ \mathsf{MTreeM}_{\mathsf{ProveMember}}(\mathsf{stateOut}', (n+1, v_0, v_1))$
return $(n+1, v_1, \pi)$

$\mathsf{ToSAccMTM}_{\mathsf{ProveDistinctElement}}(\mathsf{state}, \mathsf{state}')$
$(\mathsf{stateOut}, n, \ldots) \leftarrow \mathsf{state}$
$(\mathsf{stateOut}', n', \ldots) \leftarrow \mathsf{state}'$
for $i = 1 \ldots \min(n, n')$
  $(\_, v_0, v_1) \leftarrow \mathsf{MTreeM}_{\mathsf{GetLeaf}}(\mathsf{stateOut}, i)$
  $(\_, v_0', v_1') \leftarrow \mathsf{MTreeM}_{\mathsf{GetLeaf}}(\mathsf{stateOut}', i)$
  if $v_1 \neq v_1'$
    $\pi \leftarrow\!\!\$\ \mathsf{MTreeM}_{\mathsf{ProveMember}}(\mathsf{stateOut}, (i, v_0, v_1))$
    $\pi' \leftarrow\!\!\$\ \mathsf{MTreeM}_{\mathsf{ProveMember}}(\mathsf{stateOut}', (i, v_0', v_1'))$
    return $(i, v_0, v_1, \pi, v_1', \pi')$
return $\bot$
```

```
$\mathsf{ToSAccMTM}_{\mathsf{RemoveSet}}(\mathsf{state})$
$(\mathsf{stateOut}, n, \mathsf{stateIn}_1, \ldots, \mathsf{stateIn}_{n-1}, \_) \leftarrow \mathsf{state}$
$\mathsf{stateOut} \leftarrow \mathsf{MTreeM}_{\mathsf{Remove}}(\mathsf{stateOut})$
return $(\mathsf{stateOut}, n-1, \mathsf{stateIn}_1, \ldots, \mathsf{stateIn}_{n-1})$
```

```
$\mathsf{ToSAccMTV}_{\mathsf{New}}()$
return $(\mathsf{MTreeV}_{\mathsf{New}}())$

$\mathsf{ToSAccMTV}_{\mathsf{VerifyUnionMembership}}(v, x, \pi)$
$(i, v_0, \pi, v_1, \pi_1) \leftarrow \pi$
if $\mathsf{MTreeV}_{\mathsf{VerifyMember}}(v_1, x, \pi_1) \neq 1$
  return $0$
return $\mathsf{MTreeV}_{\mathsf{VerifyMember}}(v, (i, v_0, v_1), \pi)$

$\mathsf{ToSAccMTV}_{\mathsf{VerifyPrefix}}(v, v', \pi)$
if $v = v'$
  return $1$
$(i, v_1, \pi) \leftarrow \pi$
return $\mathsf{MTreeV}_{\mathsf{VerifyMember}}(v', (i, v, v_1), \pi)$

$\mathsf{ToSAccMTV}_{\mathsf{VerifyDistinctElement}}(v, v', \pi)$
$(i, v_0, v_1, \pi, v_1', \pi') \leftarrow \pi$
if $v_1 = v_1'$
  return $0$
if $\mathsf{MTreeV}_{\mathsf{VerifyMember}}(v, (i, v_0, v_1), \pi) \neq 1$
  return $0$
return $\mathsf{MTreeV}_{\mathsf{VerifyMember}}(v', (i, v_0, v_1'), \pi')$
```

Figure 4: An instantiation of a ToS-accumulator manager ToSAccMTM and verifier ToSAccMTV that use an "outer" Merkle tree and "inner" Merkle trees.