Cavefish: Communication-Optimal Light Client Protocol for UTxO Ledgers

Anonymous author
Anonymous affiliation

Abstract

Blockchain light clients (LCs) are agents with limited computational or storage resources who cannot maintain a fully validated local copy of the ledger state. Instead, they rely on service providers (SPs), typically full nodes, to access data required for tasks such as constructing transactions or interacting with off-chain applications.

In this work, we introduce Cavefish, a novel protocol for UTxO-based platforms that enables LCs to interact with the ledger and submit transactions with minimal trust, storage, and computation. Cavefish defines a two-party computation protocol between an LC and an SP, in which the LC specifies a transaction and the SP constructs it. Consequently, the LC only receives a blinded version of the transaction, preventing it from modifying or reusing the transaction while still being able to verify that the transaction matches the original intent of the LC. The SP is compensated inside the constructed transaction, eliminating the need for another protocol or exchange.

To support this, we propose a variant of the predicate blind signature (PBS) scheme of Fuchsbauer and Wolf (Eurocrypt 2024), allowing the SP to obtain a valid signature on the unblinded transaction, which it can then broadcast on the network and post on chain. Moreover, the resulting signatures verify as standard Schnorr signatures. Our construction achieves a trustless interaction in which the LC achieves their transaction goal, and the SP receives fair compensation for their effort. When Cavefish is combined with hierarchical deterministic (HD) wallets, the LC can provide a single public key and chain code to the SP, reducing communication footprint to a minimum.

To further optimize communication and computational overhead, our PBS variant relaxes the unlinkability guarantees of traditional blind signatures in favor of efficiency. We argue that this relaxation is adequate, since transactions only need to be kept private until posted on a public ledger. We implement and benchmark the Non-interactive Argument of Knowledge (NArg) component of our protocol on two major UTxO-based blockchains. Despite being the most computationally demanding part, our results show that proving and verification times, as well as circuit sizes, are practical for real-world deployment.

2012 ACM Subject Classification Security and privacy → Security protocols

Keywords and phrases light clients, protocols, wallets, blockchain

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

Blockchain technologies have emerged as a foundational component of decentralized systems, offering strong guarantees of data integrity, censorship resistance, and fault tolerance through cryptographic protocols and distributed consensus. Within this domain, the Unspent Transaction Output (UTxO) model represents a distinctive paradigm for managing asset ownership and validating transactions. The UTxO model was initially introduced by Bitcoin and subsequently adopted by other platforms such as Cardano. In contrast to account-based models, UTxO-based blockchains accommodate parallelism and concurrent processing more effectively but also introduce challenges in terms of complexity and client verification.

Full nodes in a UTxO-based blockchain are required to download and validate the entire chain history to ensure correctness and security. This requirement presents a significant barrier to participation for resource-constrained devices, such as smartphones and embedded systems. Light client (LC) protocols aim to mitigate this issue by enabling nodes to interact with the blockchain in a secure and efficient manner without maintaining full historical data. These

42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:30
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

protocols must strike a careful balance between minimizing resource consumption and preserving critical security properties, such as transaction inclusion, double-spending resistance, and above all, chain validity.

Blockchains are append-only data structures that grow continuously over time. As the chain length increases, it becomes prohibitively expensive for a light client (LC) to scan the entire history to verify past transactions or to locate a specific UTxO.

The question answered by this paper is "how can a user of a light client engage with a full node acting as a service provider to request and approve transactions (e.g. from their wallet) in a secure way without knowing anything about the current chain and ledger state and minimal communication effort?" In this paper, we present a solution to this question in the form of *Cavefish*, a novel intent-based light client protocol designed for UTxO-based blockchains. Cavefish enables LCs to avoid querying the current ledger state and submit transactions requiring only minimal local storage and computation. In order to submit a transaction on chain, the LC engages with a service provider (SP) in a two-party computation protocol that yields a signed transaction indistinguishable from one created by a full node. In addition to the low storage and computational requirements, our LC protocol is communication-optimal. After the LC has instructed the SP about the type of transaction it wishes to create, the protocol can be completed in as few as two rounds. The LC does not need to download let alone parse the blockchain. The only information the SP must obtain from the LC are the addresses where the funds are located. Cavefish is compatible with hierarchical (HD) wallets [49] allowing straightforward address discovery over a range of child addresses with the LC sending only a single public key together with its chain code. The only complexity is having the LC sign the requested signature after the SP constructs it. Sending the signature in the open would allow the LC to modify the transaction, potentially removing SPs fee. Instead, the SP transmits the result in a redacted form, which we call an *abstract transaction*.

The LC and SP then complete a blind signature protocol where the LC verifies if the transaction satisfies its defined specifications and only then creates valid signature(s) which are sent back to the SP. The abstract transaction is used to speed up this check. As a last step, the SP attaches the signatures to the transaction and posts them on the blockchain on behalf of the LC.

To make our scheme viable in the real world, we adapt the blind predicate Schnorr signature scheme from [24]. More precisely, we do not require the unlinkability requirement once the transaction has been published on the blockchain. Accounting for timing, values and payees the potential anonymity set for a blinded transaction is trivially small, and in any case the transaction signed by the LC only needs to stay *private until posted*. This insight allows us to introduce the notion of a *weakly* blind predicate signature scheme, a simplification that reduces the space and time complexity of the zero-knowledge component which asserts that the abstract transaction meets the LC's specifications.

In addition to the low communication overhead, our light client protocol gives the SP the ability to be reimbursed for its computation time required to construct the transaction without the need of an additional protocol or exchange. The requested transaction includes the SP's fee, i.e., an additional UTxO output, which makes up for the SP's costs and small reward. Additionally, if the client wishes to engage with the SP over a period of time, our mechanism can be used to initialize a payment channel that can be used subsequently for fast payments to the SP without impacting the size of transactions beyond the first.

To summarize, our contributions are:

- We introduce the notion of intents, describing the desired end result of the light client's actions, as opposed to the method by which they are accomplished. We believe this to be an important abstraction as it assists in the conception, development and study of specialized efficient protocols as opposed to aiming for parity with full clients. To this end, we describe a domain-specific language (DSL) allowing the concise construction of intents for UTxO-based ledgers.

- We propose Cavefish, a light client protocol that avoids any enrollment or synchronizing with chain whilst maintaining safety, providing compensation to the SP, imposing minimal communication overhead and being compatible with any UTXO blockchains using Schnorr signatures.
- We introduce the notion of weakly blind predicate signatures, motivated by our "private until posted" goal. This brings together the notions of blind predicate signatures of [24] with the notion of signatures on committed messages (SBCM) from [7].
- We implement and benchmark the zero-knowledge proof component of Cavefish for two major blockchain platforms, Bitcoin and Cardano, and show that an "unoptimized" implementation achieves proving and verification times that are viable in a real-world deployment.

2 Background and Related Work

2.1 Light clients, off-chain payments, and chain explorer services

Light clients. Our approach facilitates transaction submission for a light client that is not aware of the history of a blockchain. Instead, it operates more like a single-chain intent-resolution protocol. Therefore, we compare our work with existing solutions for light clients as well as mechanisms that allow the interaction with a blockchain when in resource-constrained operation. Our approach is different from a more "traditional" (i.e., what commonly is referred to) light client that has the primary goal of syncing to the blockchain in order to acquire the information necessary to interact with a smart contract or to submit a transaction. We first describe the common idea of a light client and then outline concepts related to our work that complement light clients.

There is existing work analyzing light client functionality [2] [15]. The majority of light client designs include the following main functionalities it is expected to perform: (1) issue queries, such as for the balance of an account, or the state of a transaction, and (2) safeguard secret information and submit transactions to the blockchain. In order to implement these functionalities, light clients use several generic techniques, most notably: header verification and consensus evolution verification. Unlike a *full node*, a header verification light client *only verifies the headers of blocks*, and skips the verification of transactions and account balances [2]. The light client is able to trust the resulting chain state if the data is signed by a sufficiently large set of full nodes (a multi-prover approach). The technique was made popular with SPV in Bitcoin [37] and nearly universally adopted by most practical approaches for light clients, e.g., [42], along with its variations [41]. Because the validator set can change, consensus evolution verification is needed for blockchains running on proof-of-stake. Another common technique is to compress the blockchain and/or ledger state in order to reduce the information a light client need for functioning reliably (see, e.g., [9]).

Some light clients use game-theoretic approaches to eliminate the need for a trusted committee (a single-prover model), e.g. by implementing the slashing of previously deposited collateral in case of misbehavior [34]. Our model is both single-prover and does not require a deposit to be made, as neither the service provider nor the light client risk their assets during the protocol execution. Moreover, our design is independent of the underlying consensus, relying, instead, on the ledger model.

The main cryptographic building blocks that are used to realize those techniques are succinct representation and proofs, such as data accumulators (often Merkle trees) and commitments and SNARKs (e.g. used to aggregate multi-signatures [46]). Suitable signatures and hash functions are needed for certain light client designs. Examples of these include aggregate signatures and threshold signatures [11, 2].

Intents. We also consider *intents* and *solver networks* to be related work. These designs attempt to establish a relationship between solvers and users via their (light) clients [20]. A light client issues an intent via as an abstracted transaction object. Then, solvers process the intent, incentivized by transaction fee or intent execution reward. The users are then free to accept or reject a solver's proposal. In case the solvers are required to provide a deposit, slashing incentivizes honest behavior of a rational actor. Since concepts around solver networks are relatively new, there is currently no universal standard governing the specification for intents and abstract transaction objects. To the best of our knowledge, this is the first work describing an intent-based light client protocol taking advantage of specific features of the UTxO ledger model that both minimizes communication and offers built-in incentive structure.

Cross-chain intents. A lot of work on intents applies is focused on building cross-chain intent resolution, which requires more sophisticated protocols that are able to communicate with multiple chains at once [50] . Some protocol designs choose to resolve intents on-chain directly [43], while other offer both layer-1 and layer-2 options [17]. Yet another designs is a consensus protocol relying on on minimal trust assumptions while conforming to the interledger standard [18]. Focusing on a single chain allows us to construct a protocol which makes use of the unique features of the underlying blockchain, and requires reduced communication (only between two parties). In its simplicity and limited communication, our protocol also resembles payment channels.

Payment channels. The concept of *payment channels*, and the related notion of *state channels*, also bear similarities to our approach. Payment channels are a type of off-chain mechanism for blockchains. A payment channel can be used to conduct a series of transactions without interacting with the main blockchain, e.g. [32] [44] [42]. The creation of a payment channel between parties requires locking funds in an on-chain transaction [35]. Some channels, such as Hydra [14], not only allow simple payments between users, but can simulate the majority of the on-chain transaction processing mechanism internally. This type of channel is usually called a *state channel*.

Cavefish interacts well with payment channels: on the one hand cavefish can be used to setup a payment channel from a light client, and on the other setting up a payment channel allows the SP to be compensated offchain eliminating the on-chain overhead of Cavefish.

API and Explorer Services. Many realistic UTxO ledger implementations (e.g. Ergo [1], Cardano [2], and BitCoin [3]) are set up in a way that an invalid transaction will, in most cases, not result in any update to the ledger state, but get rejected instead. This makes services such as blockchain explorers [4] or Blockfrost (API as a service for accessing the Cardano blockchain) [5] some of the strongest competitors with our proposal, as they provide the data needed for the LC to construct their transaction, often with a relatively high degree of reliability. Unlike our design, these services do not currently appear to support transaction construction for light clients. Revenue from explorer services is either ad-based, or they may charge users in fiat currency, or the service is free in order to promote use of the specific service/blockchain. We, on the other hand, propose a protocol in which service providers are compensated in assets on the same blockchain as the one to which their transaction gets applied, and the payment structure consists of a one-time, no-setup atomic exchange of assets for services.

[1] `https://ergoplatform.org/`
[2] `https://cardano.org/`
[3] `https://bitcoin.org/`
[4] `https://beta.explorer.cardano.org/`
[5] `https://blockfrost.dev/`

2.2 Hierarchical deterministic wallets and address discovery

UTXO based blockchains, including Bitcoin and Cardano use the concept of Hierarchical Deterministic (HD) wallets [49] where child addresses can be created in a deterministic way using a public key of a keypair sk,pk, and some short auxiliary data (the chain code). Given a public key pk, chain code c and requested index i for a child key, a hash function is used to produce a scalar s_i enabling the derivation of the child key as $sk_i = sk + s_i$. It is also possible to use s_i without knowing sk to directly derive the corresponding public key $pk_i = pk \cdot g^{s_i}$. The same method is used to produce a child chain code c_i, so that multiple generations of children are possible.

In our application, we can assume that a wallet can give the service provider (SP) a single public key and chain code, letting the SP do address derivation and lookup on their side using some agreed-upon bounds or heuristics on the index depth. This keeps the communication cost between the light client and the SP constant.

2.3 Schnorr blind signatures

As soon as their patent expired in 2008, Schnorr signatures [39] have been gaining significant adoption, replacing RSA in many scenarios due to their smaller size and faster verification. Compared to (EC)DSA, Schnorr offers similar efficiency and guarantees, but (EC)DSA security proofs are most likely not possible without strong idealization [29]. As a consequence, EdDSA [6], a popular variant of the Schnorr scheme, is currently under consideration by NIST for standardization. The security of Schnorr signaatures is based on the discrete logarithm assumption [38], with proofs in the random oracle model (ROM) as well as in the algebraic group model (AGM) and the ROM [23].

Standard Schnorr signatures are now widely used in blockchains such as Bitcoin, Bitcoin Cash, Litecoin, and Polkadot. Monero, Zcash, and Cardano use the EdDSA variant. The adption of Schnorr and EdDSA signatures is driven by privacy [8] and scalability gains [36] but also the straightforward extension to *blind Schnorr signatures* [16]. Most Schnorr-based blind signature schemes require multiple rounds [16], which can make the protocol suceptible to denial-of-service attacks. Research has thus been focusing on making blind signatures *concurrently* secure where more than one session can be intertwined, such as [4, 12, 45, 28].

Our construction hinges on the the concurrently secure blind and partially blind signing protocol for standard Schnorr signatures found in [24] by Fuchsbauer, et al.—the first work stating rigorous security guarantees for a practical blind signature scheme based on Schnorr signatures. Unlike schemes that had been presented before, [24] is not vulnerable to the attacks described in [47, 5] which showed that the hardness-assumption[6] existing schemes relied on for concurrent security can be solved in polynomial-time. As our light client protocol is based on [24], it also provides partial and predicate blindness, two properties that allow us to describe a transaction in an abstract way featuring “redacted” parts. We briefly outline the construction of Fuchsbauer et al. [24] in the following. The scheme is equivalent to the blind signature scheme [16] by Chaum and Pedersen but, in order to make the protocol concurrently secure, adds a commitment phase at the beginning where the user sends an encrypted version of the message m and blinding values (α,β) to the signer. In addition to that, the second message by the user includes a zero-knowledge proof alongside the Schnorr challenge c. The zero-knowledge proof asserts to the signer that the initial (encrypted) commitment and c have been derived from the same m,α,β.

[6] The ROS (Random inhomogeneities in a Overdetermined Solvable system of linear equations) problem was initially studied by Schnorr in [40]

In addition to obtaining concurrent security, one can leverage the zero-knowledge proof to assert additional facts, most notably, the user can prove to the signer that certain predicate(s) over message m holds. This effectively turns the fully blind scheme into a predicate blind scheme. Furthermore, support for predicate blindness implies partial blindness [24] since constructing a predicate that checks equality for parts of m can be used to assert to the user that parts of the message correspond to the expected value(s).

3 Technical Background

Notation. We use $y \leftarrow x$ to denote that variable y is assigned the (possibly randomized) evaluation of x. When x is a set, we denote uniformly random sampling from the set. We use $a = b$ to denote boolean comparison between a and b and $z := w$ to denote equality by definition. We use $||$ to denote concatenation of bitstrings. When algorithms are randomized, we denote them as $A(x;r)$, where x is the input and r is the randomness, belonging to a randomness space $\mathcal{R}$. When we write $y \leftarrow A(x)$ we imply $r \leftarrow \mathcal{R}; y \leftarrow A(x;r)$. In algorithm descriptions we write $x,y,z \subseteq a$ to imply parsing a to obtain x,y,z.

Our protocols operate in the discrete log setting, where we assume the discrete logarithm problem is hard. We also follow standard conventions with regards to public key encryption using ($\mathsf{PKE.KeyGen}, \mathsf{PKE.Enc}, \mathsf{PKE.Dec}$) to describe schemes and require IND-CPA security. For signatures, we use Schnorr signatures ($\mathsf{SDS.Setup}, \mathsf{SDS.KeyGen}, \mathsf{SDS.Sign}, \mathsf{SDS.Ver}$). Finally, we use parametrized non interactive Arguments ($\mathsf{NArg.Rel}, \mathsf{NArg.Setup}, \mathsf{NArg.Prove}, \mathsf{NArg.Ver}$, $\mathsf{NArg.SimProve}$) to allow the relation being proven to depend on the group setting. For reasons of space we present the full description of these primitives in Appendix A

The security of Schnorr signatures, has been well studied in the random oracle model (ROM), where a hash replaced with an ideal random function the adversary can only call as an oracle. However in this work the need arises to instantiate the underlying hash function H so that we may reason about it within a proof system [3] e.g. prove that a known y is $y = \mathsf{H}(x)$ for some secret x. This renders ROM-based proofs inapplicable necessitating an assumption on the security of the scheme. This does not differ significantly from typical implementation practices (where the hash function is infact drawn from a small pool for standardized options), and also from the treatment of this issue in the literature [24].

▶ **Definition 1.** *A hash function generator* $\mathsf{HGen}(n) \to \mathsf{H}$*, on input* $n \in \mathbb{N}$ *generates a hash function* $\mathsf{H}: \{0,1\}^* \to \mathbb{Z}_n$.

▶ **Assumption 1.** [24] There exists a group generator GGen and hash function generator HGen s.t. the Schnorr signature scheme SDS is strongly unforgeable under definition 16.

Similarly, in the ROM we can trivially guarantee H is pseudorandom, whereas for our setting we require an additional assumption:

▶ **Assumption 2.** The hash function generator HGen is such that for all n

$$\left|\Pr[\mathsf{H} \leftarrow \mathsf{HGen}(n); (m_0,m_1) \leftarrow \mathcal{A}(\mathsf{H}); b \leftarrow \{0,1\}; r \leftarrow \{0,1\}^\lambda : b = \mathcal{A}(\mathsf{H}(m_b||r))] - 1/2\right|$$

is negligible in λ.

4 Ledger Model

The ledger model to which we tailor our light client design is a UTxO ledger with multi-asset support (UTxO_{ma}), first introduced in [13]. The UTxO ledger model, such as the one used by BitCoin [37], Ergo [22], and Cardano [10], maintains a record, called the *UTxO set*, of transaction outputs added by transactions that have been applied throughout its history, but not yet spent by subsequent transactions. We chose the UTxO_{ma} ledger because it allows us to demonstrate relevant usecases of our light client design (which would also work for a single-asset UTxO ledger), without introducing unnecessarily complexity of the Extended UTxO ledger. For completeness, and in order to establish notation, we include an overview of the UTxO_{ma} model. For additional notation explanation, see Figure 9.

Blocks and Ledger States. A block $\mathsf{Block} = \mathsf{Header} \times [\mathsf{Tx}]$ is a data structure used to update the state of the ledger by applying a list of transactions $lstx \in [\mathsf{Tx}]$ contained in the block, as well as doing some other checks and updates we do not model here. Among other data, the block header contains a slot number field $\mathsf{slot} : \mathsf{Header} \to \mathsf{Slot}$, which represents the blockchain time at which the block is produced.

The ledger state is a data structure which is updated by applying blocks incoming on the network. The ledger state LState contains (among other data we do not model here) the UTxO set field, $\mathsf{utxo} : \mathsf{LState} \to \mathsf{UTxO}$. It also contains the parameter $\mathsf{minfee} : \mathsf{LState} \in \mathbb{N}$, which is the minimum fee a transaction must pay. A block b can extend the blockchain (i.e. update the current state $s \in \mathsf{LState}$) whenever the function

$$\mathsf{checkBlock} : \mathsf{LState} \times \mathsf{Block} \to \{0,1\}$$

applied as $\mathsf{checkBlock}\ (s, b)$ returns 1. Then, the updated block state is computed by $\mathsf{updateState}\ (s,b)$. We do not give full specifications of $\mathsf{checkBlock}$ or $\mathsf{updateState}$, as they are not required to model our light client approach.

Full Nodes. Let s_0 be some verified state, e.g. a genesis state, or a verified checkpoint state, and suppose $[b_0,...,b_k]$ is a list of blocks that have been disseminated across the network since the time slot of s_0. We assume that a *full node* is one that is to able to (1) compute the current state s_k by applying them in sequence, i.e. computing $s_{i+1} = \mathsf{updateState}\ (s_i,b_i)$ for $0 \le i < k$, (2) is able to determine if all of the blocks in the list are valid (i.e. $\mathsf{checkBlock}\ (s_i,b_i) = 1$), and (3) can produce, send, and receive blocks on the network. If all the blocks are valid, we say that $[b_1,...,b_k]$ forms a valid blockchain.

Ledger State and the UTxO Set. The state of a UTxO-based ledger necessarily contains a UTxO set. While realistic ledgers often contain additional information in their state, in our model, the ledger state is just the UTxO set. The UTxO set is a finite map, $\mathsf{UTxO} := \mathsf{TxIn} \mapsto \mathsf{TxOut}$. A transaction updates the UTxO set by either adding and removing entries.

Transactions. A transaction is the following data structure :

$$\begin{aligned} \mathsf{Tx} \quad = \quad & (\mathsf{inputs} : \mathsf{Set}\ \mathsf{TxIn},\ \mathsf{outputs} : [\mathsf{TxOut}],\ \mathsf{validityInterval} : \mathsf{Interval}[\mathsf{Slot}], \\ & \ \mathsf{mint} : \mathsf{Value},\ \mathsf{fee} : \mathbb{N},\ \mathsf{aux} : \mathbb{H},\ \mathsf{sigs} : \mathsf{Signature}) \end{aligned}$$

An input $(txid, ix) \in \mathsf{TxIn} := \mathsf{TxId} \times \mathbb{N}$ is a pair of a transaction ID and a natural number. When a transaction is applied to a UTxO set, its set of inputs is used to identify the entries which the transaction is removing from the set. In each input, $txid \in \mathsf{TxId} := \mathbb{H}$ is the hash of a (previous) transaction that added that entry to the UTxO, and ix is the index of that output in the list of outputs of that transaction.

An output $(s, v) \in \mathsf{TxOut} := \mathsf{Script} \times \mathsf{Value}$ is a pair of a script s which specifies some constraints that are checked when the output is spent, and the assets v contained in the output. Note here that our ledger model has native *multi-asset support*, following the scheme outlined in prior work [13]. That is, the $v \in \mathsf{Value}$ in the output contains not only a quantity of a single currency (like BitCoin), but an arbitrary finite number of different types of assets with unique identifiers, alongside their quantities.

The list outputs of outputs of a transaction tx is used to construct a set of UTxO entries that will be added to the UTxO set, such that the unique identifier $txin$ of each output $o \in \mathsf{outputs}\ tx$ consists of the transaction hash txid tx, and the index of o in the list outputs tx. The entires added to the UTxO set by tx are computed in this way by mkOuts, see Figure 10.

The interval validityInterval specifies the range of slot numbers for which a transaction can be valid, or nothing whenever there is no limitation. The field $\mathsf{sigs} : \mathsf{Signature} := \mathsf{PubKey} \mapsto \mathbb{H}$ is a set of public keys, associated with their signatures on the the transaction (excluding sigs itself). The fee is the amount of primary currency a transaction pays as a system fee, which is checked to be at least the required fee minfee.

The mint field represents the assets being minted or burned by the transaction. Assets with positive quantities are said to be minted, while those with negative quantities are burned. When a transaction is applied, the constraints specified by every $p \in \mathsf{Policy} := \mathsf{Script}$ of each type of asset specified in this field are checked to make sure minting/burning of this type and quantity of asset is allowed. The aux is a field for arbitrary extra data encoded as a bytestring, e.g. a "note".

Ledger State Update. Given a UTxO set $utxo$ and a transaction tx, the function $\mathsf{updateUTxO} : \mathsf{UTxO} \times \mathsf{Tx} \rightarrow \mathsf{UTxO}$ computes the updated UTxO set by adding and removing the appropriate entries :

$$\mathsf{updateUTxO}\ (utxo,\ tx)\ =\ \{\ i \mapsto o \in utxo\ \mid\ i \notin \mathsf{inputs}\ (tx)\ \} \cup \mathsf{mkOuts}(tx)$$

While an update to the UTxO set can be computed for any transaction, only transactions that are *valid* for a given set are allowed to perform an update to the ledger state. For a given $utxo$ set, a transaction tx is valid whenever the function $\mathsf{checkTx} : (\mathsf{Slot} \times \mathbb{N}) \times \mathsf{UTxO} \times \mathsf{Tx} \rightarrow \{0,1\}$, applied as $\mathsf{checkTx}\ ((slot,fee), utxo,\ tx)$, returns 1. The checkTx function is the conjunction of the constraints specified in Section B.1. For a given block b and state s, the function $\mathsf{updateState}\ (s,b)$ updates s with the list of transactions $\pi_2\ b = [tx_1;...;tx_k]$ in such a way that the update to the UTxO set contained in s is computed by applying the transactions in sequence, i.e.

$$\mathsf{updateUTxO}\ (\mathsf{updateUTxO}\ ((...\mathsf{utxo}\ s...),\ tx_{k-1}),\ tx_k) = \mathsf{utxo}\ (\mathsf{updateState}\ (s,b))$$

For each transaction in the list, $\mathsf{checkTx}\ ((\mathsf{slot}\ b, \mathsf{fee}\ s), utxo_i,\ tx_i) = 1$ is first checked, and the entire block is considered invalid if this check fails. As part of checking transaction validity, constraints of every Script run by the transaction are checked. The constructors and evaluation of Script is given in Figure 11, and MOf is given in 10. A script, for a given set of signer keys khs and slot numbers $s1$,$s2$, can be defined to check that (some specific) m of them have signed the transaction (are included in the domain of Signature), and/or that the validity interval of the transaction starts after $s1$ and/or ends before $s2$.

5 Light Client Specification

We model light clients in terms of their functionality, constraints, and protocols for communication with full nodes and potential service providers. By functionality, we refer to the interaction between a light client and its user(s). We describe the functionality of a light client in terms of constructing (posting) and verifying resolved *intents*. A light client specifies the intent it wants resolved using a special domain-specific language (DSL). Then, the client sends the intent to a service provider (SP), who is incentivized to respond to this intent with an *abstract transaction*. An abstract transaction is a transaction that has been modified to partially conceal data in a way that (i) allows the light client to verify that their intent has been resolved in the original transaction and (ii) does not allow the client to use the abstract transaction to form a valid transaction that avoids paying the service provider fee.

$$
\begin{array}{rcl}
\mathsf{TxAbs} & = & (\mathsf{outputs}:[\mathsf{TxOut}],\ \ \mathsf{validityInterval}:\mathsf{Interval}[\mathsf{Slot}],\\
& & \ \mathsf{mint}:\mathsf{Value},\ \ \mathsf{fee}:\mathbb{N},\ \ \mathsf{sigKeys}:\mathsf{Set\ PubKey})\\
\\
\mathsf{mkAbs} & : & \mathsf{Tx} \rightarrow \mathsf{TxAbs}\\
\mathsf{mkAbs}\ tx & = & tx\ \{\ \mathsf{outputs} = tx.\mathsf{outputs},\\
& & \ \mathsf{outputs}[0].\mathsf{Value} = -1,\ \ \texttt{//Leftover balance to change address}\\
& & \ \mathsf{validityInterval} = tx.\mathsf{validityInterval},\\
& & \ \mathsf{mint} = tx.\mathsf{mint},\ \ \mathsf{fee} = tx.\mathsf{fee}\\
& & \ \mathsf{sigKeys} = \mathsf{dom}\ (tx.\mathsf{sigs})\ \}
\end{array}
$$

Figure 1 Abstract transaction type TxAbs and constructor mkAbs

A light client's constraints may be on the communication cost to answer a series of queries, available state/data storage, and assumptions about connectivity. As a consequence, a particular light client design may be limited in the intents it can construct even if a broader class of intents is supported by available software.

5.1 Intent specification

An *abstract transaction*, TxAbs, is the data structure that a light client receives from the service provider instead of the full plaintext transaction (see Fig. 1). In TxAbs, the aux and the inputs fields are removed, and instead of the Signature field, it contains only the signing keys sigKeys and not the corresponding signatures. The value of the first output is zeroed out to allow for any leftover input funds to be returned to the LC's change address. The intent a light client constructs is any expression in the DSL $\mathcal{I}_{\mathsf{post}}$, see Figure 2.

For the implementation of the example usecases of our design, we require the definition of the following functions:

(i) The function that constructs a transaction based on the specification (this function may fail, outputting nothing instead of a transaction, if the intent cannot be resolved), see Fig. 12.

$$\mathsf{mkToSpec} : \mathsf{LState} \times \mathcal{I}_{\mathsf{post}} \rightarrow \mathsf{Tx}^{?}$$

(ii) The function that checks that a given abstract transaction matches the intent specification, see Figure 2

$$\mathsf{chkSpec}\ (\mathcal{I}_{\mathsf{post}} \times \mathsf{TxAbs}) \rightarrow \{0{,}1\}$$

Note that the function mkToSpec constructs a transaction by editing an initial transaction $\mathsf{initTx}_{a,mf}$ (see Figure 12). This transaction includes a minimum fee mf as well as a tip tip to the service provider, output to its address a. These functions are defined such that for any l, i, where $\mathsf{mkToSpec}\ (l,\ i) \neq \mathsf{nothing}$, necessarily $[\![i]\!]_{\mathsf{DSL}}\ (\mathsf{mkAbs}\ (\mathsf{mkToSpec}\ (l,\ i))) = 1$. The design of the DSL is minimal, and is meant only to showcase the operation of the light client protocol. To expand the functionality of the light client, more constraints need to be included in the DSL, e.g., a way to limit transaction size, or support for specifying desired token exchanges. We leave this for future work.

We showcase two examples in the following, for a given ledger state l:

(txi_1) Intent to mint some token t within an interval of length j and maximum fee $f \leq \mathsf{minfee}\ l$ (note that this will not be a valid transaction since it will have no inputs):

$$i_1 = \mathsf{AndExps}\ [\mathsf{MustMint}\ t; \mathsf{MaxInterval}\ j; \mathsf{MaxFee}\ f; \mathsf{ChangeTo}\ s]$$

$\mathcal{I}_{\text{post}}$ CONSTRUCTORS

$$
\begin{aligned}
\mathsf{MustMint} &: \mathsf{Value} \to \mathcal{I}_{\text{post}}\\
\mathsf{SpendFrom} &: \mathsf{Script} \to \mathcal{I}_{\text{post}}\\
\mathsf{MaxInterval} &: \mathsf{Slot} \to \mathcal{I}_{\text{post}}\\
\mathsf{PayTo} &: (\mathsf{Value} \times \mathsf{Script}) \to \mathcal{I}_{\text{post}}\\
\mathsf{ChangeTo} &: \mathsf{Script} \to \mathcal{I}_{\text{post}}\\
\mathsf{MaxFee} &: \mathbb{N} \to \mathcal{I}_{\text{post}}\\
\mathsf{AndExps} &: [\mathcal{I}_{\text{post}}] \to \mathcal{I}_{\text{post}}
\end{aligned}
$$

EVALUATION OF $\mathcal{I}_{\text{post}}$

$$
\begin{aligned}
[\![\_]\!]_{\mathsf{DSL}} &: \mathcal{I}_{\text{post}} \to \mathsf{TxAbs} \to \{0,1\}\\
[\![\mathsf{MustMint}\ v]\!]_{\mathsf{DSL}}(tx) &= v \leq tx.\mathsf{mint}\\
[\![\mathsf{SpendFrom}\ s]\!]_{\mathsf{DSL}}(tx) &= [\![s]\!](\mathsf{dom}\ (tx.\mathsf{sigs}), tx.\mathsf{validityInterval})\\
[\![\mathsf{MaxInterval}\ i]\!]_{\mathsf{DSL}}(tx) &= (tx.\mathsf{validityInterval})_2 - (tx.\mathsf{validityInterval})_1 \leq i\\
[\![\mathsf{PayTo}\ (s,v)]\!]_{\mathsf{DSL}}(tx) &= (s,v) \in tx.\mathsf{outputs}\\
[\![\mathsf{ChangeTo}\ s]\!]_{\mathsf{DSL}}(tx) &= (s, \mathsf{consumed}\ -\ \mathsf{produced}) \in tx.\mathsf{outputs}\\
[\![\mathsf{MaxFee}\ f]\!]_{\mathsf{DSL}}(tx) &= tx.\mathsf{fee} \leq f\\
[\![\mathsf{AndExps}\ [a1;a2;\ldots;ak]]\!]_{\mathsf{DSL}}(tx) &= ([\![a1]\!]_{\mathsf{DSL}}\ tx) \wedge ([\![a2]\!]_{\mathsf{DSL}}\ tx) \wedge \ldots \wedge ([\![ak]\!]_{\mathsf{DSL}}\ tx)
\end{aligned}
$$

Figure 2 $\mathcal{I}_{\text{post}}$ constructors and evaluation

(txi_2) Intent to pay x from outputs locked by $\mathsf{RequireSig}\ k1$ to $\mathsf{RequireSig}\ k2$ (note that this intent may not be possible to resolve in the key $k1$ does not have sufficient funds on the ledger l)

$$i_1 = \mathsf{AndExps}\ [\mathsf{SpendFrom}\ (\mathsf{RequireSig}\ k1); \mathsf{PayTo}\ (\mathsf{RequireSig}\ k2, x); \mathsf{ChangeTo}\ (\mathsf{RequireSig}\ k1)]$$

The resulting transactions are given in Figure 13, assuming the intents can be resolved. The minimum required capacity of our light client is to perform all the actions that are specified in the protocol we describe in the next section.

6 Light Client Protocols

We give an overview on our light client protocol and its components. Our protocol maximizes space and communication efficiency by essentially pushing all storage and lookup tasks to the SP, whilst guaranteeing the resulting transaction is safe for the light client to sign, and compensating their SP for their effort. This is non-trivial as the transaction must be signed sight-unseen (blinded) otherwise, the light client could simply remove the payment to the SP and submit the modified transaction.

To satisfy both parties, we rely on cryptographic tools as well as some established features in implemented blockchains such as transaction validity periods and handling of invalid transactions. In the rest of this section, we make the following assumption: all communication between a light client and a service provider happens over a secure channel (e.g. a TLS connection).

6.1 Definitions and requirements

▶ **Definition 2** (Correctness). *A transaction building protocol is **correct** if for an honest light client, and service provider and any state* $\mathsf{LState}_i \in \mathsf{LState}$ *and intent* $int_{post} \in \mathcal{I}_{\text{post}}$ *such that:*

if $\mathsf{mkToSpec}(\mathsf{LState}_i, int_{post}) \neq \bot$*, then the protocol completes and the transaction* Tx' *produced by the SP is such that* $\mathsf{checkTxL}\ (\mathsf{LState}_i, \mathsf{Tx}') = 1$.

We use $\mathsf{checkTxL}\ (l, \mathsf{tx})$ as shorthand for $\mathsf{checkTx}\ ((slot, fee), utxo,\ \mathsf{tx})$, with ledger state l containing UTxO state utxo and current slot number slot and minimum fee fee.

▶ **Definition 3** (Safety). *A transaction building protocol is* ***safe*** *if for an honest light client and any service provider SP we have that: if transaction* Tx' *is produced by the SP is such that* $\mathsf{checkTxL}\ (LState, \mathsf{Tx}') = 1$ *, then it must be that* $\mathsf{chkSpec}(int_{post}, \mathsf{Tx}') = 1$.

The safety property is derived from two factors: first, the unforgeability property of $\mathsf{WBPS}[P]$ will ensure that only transactions that satisfy the given predicate will be signed. Second, our assumptions on ledger rules ensure that signed transactions involving spent UTXOs will have no effect (as opposed to something undesirable to the user).

▶ **Definition 4** (Private until posted). *A transaction building protocol has the* ***private until posted*** *(PuP) property if a PPT adversarial client* $C_{\mathcal{A}}$ *cannot win the following experiment with probability significantly higher than* $\frac{1}{2}$.

For an honest service provider SP and any client $C_{\mathcal{A}}$ *we have that: client provides 2 states* $\mathsf{LState}_0, \mathsf{LState}_1$ *and one intent* $int_{post} \in \mathcal{I}_{\mathsf{post}}$ *such that for* $i = 0,1$ *it is* $\mathsf{mkToSpec}(\mathsf{LState}_i, int_{post}) \neq \bot$*. The experiment flips a coin* d *and runs the protocol between SP and* C^* *using state* LState_d. $C_{\mathcal{A}}$ *outputs* d^**. The adversarial client wins iff* $d = d^*$.

The Private until posted property of our protocol is derived from Theorem 9 and the protocol structure. Given an adversary against $\mathcal{A}$ PuP, we can build an adversary against the weak blindness of the $\mathsf{WBPS}[P]$ scheme by running the SP once on LState_0 and once on LState_1 to produce two challenge messages for the blindness security game and simulate the role of the challenge towards $\mathcal{A}$ by passing on the rest of the protocol messages (the experiment enforces random padding internally). If A can distinguish which ledger was used in producing the messages we can use the same index as our guess in the PuP game.

6.2 Cavefish: a communication-optimal transaction building protocol

We given an overview on our light client protocol and its components. As the protocol is uniquely suited for UTxO-based blockchains, we develop our approach to support the general ledger model explained in Section 4 and the intents set forth in Section 5.1.

6.2.1 Security model and requirements

Rational Actors. For liveness, we assume that protocol participants are rational in the sense that they will opt to complete tasks that benefit them directly (i.e in the form of rewards, or in the form of a desired transaction being posted). For safety we assume parties can be fully malicious.
Transaction Expiry. For safety, we require that transactions can be set to be valid only within a certain time window. This is to prevent trivial attacks where a malicious SP delays posting a transaction indefinitely. The user could contact a different SP to create a second transaction with similar parameters (e.g. by paying the same recipient with different coins) with the risk that the first SP would also post their transaction, making the user pay twice.
Idempotent Doublespends. For safety we also require that attempted double spends are idempotent, i.e. they are simply not valid transactions and do not penalize (slash) the user for signing them.
Trusted Setup. We specify a proof system using a trapdoor, which is a common characteristic of efficient options such as Groth16 [27] or Plonk [25]. It is possible to use alternatives featuring

transparent setup with some impact to communication size. In the case of a transparent setup for NArg, we still need to generate keys for PKE. In the case of Elgamal in the random oracle model, this can be accomplished by hashing an unpredictable string into a group element.

6.3 Protocol description

The protocol operates as follows: the light client connects to the SP and sends a posting intent int_{post}, as well as a public key that has received payments (and, optionally a BIP-032 chaincode so that the SP can also derive child keys from the initial one –we explain the details in Section 8). The SP will then search for UTXOs belonging to the client and form a transaction that satisfies the intent int_{post}, including in the transaction a tip for the SP's own work. Lastly, the SP adds a random note $\mathsf{aux}_{\mathsf{nt}}$ to the note field of the transaction. Then, the client and SP engage in a weakly blind predicate signature protocol so as to efficiently have the client sign the transaction whilst ensuring that (1) the client needs to sign before learning the full transaction (typically the used UTXOs are hidden), and (2) that the signed transaction satisfies int_{post} by means of satisfying the corresponding predicate.

We can instantiate our protocol for two kinds of intents: "light" and "extended". In the light version, we support intents where the transaction abstract does not give the light client any information the SP does not wish to disclose—by design, the abstract hides the input transactions, but it could reveal, e.g. some state details of a smart contract.

A practical example of such intents is $int_{\text{light}}(\mathsf{tx}) := \{\textbf{return } (\mathsf{makeAbs}(\mathsf{tx}) = \mathsf{makeAbs}(tx_0))\}$ i.e. the light client fully determines the requested transaction (including SP fee) to be tx_0, apart from the input UTXOs (which it does not know) and the note field aux which is free to the SP to set. This implies that TxAbs provides no new information to the light client. This reduces the computational effort by SP invested in running the NArg prover as chkSpec consists only of substring equality checks. We note here that no range check is necessary for values: makeAbs (Section 5.1) retains the prescribed value assigned to all outputs except the first (which is used as a "change" address), i.e., the only option for the SP is to assign all excess value to the change address.

Alternatively, in the extended variant, we set $\mathsf{makeAbs}(x) := \{\textbf{return } 1\}$, and change the predicate check from chkSpec $(\mathcal{I}_{\mathsf{post}} \times \mathsf{TxAbs}) \to \{0,1\}$ to chkSpec $(\mathcal{I}_{\mathsf{post}} \times \mathsf{Tx}) \to \{0,1\}$. This too ensures that no information is leaked by TxAbs. Our results hold for both variants.

We show our complete light client protocol in Figure 3, leveraging the weakly blind signatures predicate of section 7.2. For simplicity, the figure describes the process for a single UTXO signature, if more are needed, multiple WBPSsessions can be run sharing the same com_{tx}.

6.3.1 Security

▶ **Theorem 5.** *If the ledger rules include the* Idempotent Doublespends *property and* $\mathsf{WBPS}[P]$ *is unforgeable, the protocol of Section 6.3 is safe.*

Proof. The safety property is derived from two factors: first, the unforgeability property of $\mathsf{WBPS}[P]$ (Theorem 10) will ensure that only transactions that satisfy the given predicate will be signed. Second, our assumptions on ledger rules ensure that signed transactions involving spent UTXOs will have no effect (as opposed to something undesirable to the user). ◀

▶ **Theorem 6.** *If* $\mathsf{WBPS}[P]$ *scheme is weakly blind, the protocol of Section 6.3 is private until posted.*

Proof. The Private until posted property of our protocol is derived from Theorem 9 and the protocol structure. Given an adversary against $\mathcal{A}$ PuP, we can build an adversary against

Signer (LC)		Service Provider (SP)
		Obtain newest LState_t ls
Generate intent $int_{post} \in \mathcal{I}_{\mathsf{post}}$		
	$\xrightarrow{int_{post}, vk_c}$	
		Determine client UTXOs from (int_{post}, vk_c)
		Generate $\mathsf{tx} = \mathsf{mkToSpec}(ls, int_{post})$,
		$\mathsf{TxAbs} = \mathsf{mkAbs}(\mathsf{tx})$, $\mathsf{aux}_{\mathsf{nt}} \xleftarrow{\$} \{0,1\}^{\kappa}$
WBPS Execution for $m := \mathsf{tx}\|\|\mathsf{aux}_{\mathsf{nt}}$		Produce commitment com_{tx}
	$\xleftarrow{com_{\mathsf{tx}}, \mathsf{TxAbs}}$	
Produce blind sig. com. $R = g^r$	$\xrightarrow{R}$	
	$\xleftarrow{c, \pi}$	Produce challenge c and proof π
Check proof π		
Produce $s = r + cx$	$\xrightarrow{s}$	Produce signature $\sigma = (R, s)$
		If valid, post $(\mathsf{tx}\|\|aux, \sigma)$ on chain.

Figure 3 Cavefish protocol featuring WBPS scheme. A detailed descripton of the WBPS scheme is in Fig. 6.

the weak blindness of the $\mathsf{WBPS}[P]$ scheme by running the SP once on LState_0 and once on LState_1 to produce two challenge messages for the blindness security game and simulate the role of the challenge towards $\mathcal{A}$ by passing on the rest of the protocol messages (the experiment enforces random padding internally). If A can distinguish which ledger was used in producing the messages we can use the same index as our guess in the PuP game. ◀

6.4 Discussion

Suboptimal Transactions. In response to an LC query, it is possible for an SP to construct a *sub-optimal* transaction (in terms of cost) which nevertheless matches the LC's specification. For example, the LC may include a larger-than-necessary system fee in the transaction, or not provide the LC with the best available exchange price for a specific token. We consider cost optimizations an orthogonal issue that can be addressed by e.g. a market system.

Liveness against Malicious SPs. Malicious SPs are able to force delays for the user by completing the signing and never posting the transaction. This forces the user to wait until the signed transaction is no longer valid before retrying. Otherwise, as intents are not necessarily uniquely satisfied , it is possible that the first (malicious) node may be able to post the first transaction (which was purposefully delayed) after the posting of the second one. This can cause the user to e.g. double pay for a service as well as the full node fee. Even so, given enough retries the user will reach an honest SP and the transaction will be posted.

6.4.1 Multi-SP protocols and optimality

However, we can also formulate a version of this protocol where the LC instead sends the specification to multiple SPs, selecting the best response, and engaging in the rest of the protocol only with that SP.

To do this, let $\mathsf{opt}: \mathsf{TxAbs} \to \mathbb{Z}$ be a function that rates abstract transactions to express LC's preferences. For example, the total amount of primary tokens spent by the transaction can be such a function : $\mathsf{opt}\ \mathsf{tx} = \mathsf{coinValue}\ (\sum_{o \in\ txd.\mathsf{spentOuts}} o.\mathsf{value})$. A transaction tx is *optimal* in a set $S \in \mathsf{Set\ TxAbs}$ when $\mathsf{opt}\ \mathsf{tx} = \mathsf{min}\ \{\ \mathsf{opt}\ tx' \mid tx' \in S\ \}$.

To get a multi-SP protocol, the first step of the single-SP protocol in Figure 3 must be augmented, so that the pair (opt,int_{post} is sent to each SP instead of just sending int_{post}. Upon receiving transactions $\mathsf{tx}_{A,i}$, with $0 \leq i < k$, from each of the k SPs responding to LC's query, LC will engage in the rest of the protocol only with the sender of $\mathsf{tx}_{A,i}$.

Note that the opt function and the specification serve different purposes in the protocol. The specification is checked, and any response transaction that does not satisfy it is discarded. On the other hand, it is not required that a transaction be optimal across all possible specification-satisfying transactions. The kind of sophisticated optimization (such as what is required, e.g., for optimized order-matching) would require an entirely distinct set of tools for demonstrating the optimality result, such as ZK proofs about the full blockchain state (rather than just the associated transaction), together with evidence that the proof is about state that is *sufficiently current* (see Section 6.4). For example, a proof that that there were no better offers for a specific token available on the ledger at the time the SP produced a response to the LC. We do not assume that either the SP or the LC are necessarily capable of performing or verifying (resp.) optimality according to the function LC requested be optimized, but an LC is capable to comparing transactions using opt.

7 Blind Signatures for Abstract Transactions

In order to allow the light to sign an abstract transaction we implement blind signatures on (partially) blinded transaction objects. At the minimum the SP hides the inputs to the transaction, i.e., the references to UTxO objects which are present in the ledger and required to cover the transaction. This ensures that the light client cannot simply obtain the transaction from the SP, then edit it to remove the SPs compensation.

Our construction is inspired by the predicate blind signature mechanism in [24]. It realizes a concurrently secure blind and partially blind signing protocol resulting in standard Schnorr signatures. This is a core property as it allows our protocol to be compatible with popular blockchains such as Bitcoin and Cardano without any modifications.

Given our application, the unlinkability property of blind signatures is not of much benefit: signed transactions are somewhat infrequent, often have different payees or payment sums (i.e different predicates) and re indirectly timestamped due to their time of posting on the blockchain. Further, there seems to be little benefit in preventing the light client from knowing which protocol section produced which payment. For this reason, we do away with the requirement of the signer being unable to distinguish between messages signed on different sessions. We only require that the singer is unable to distinguish the message *before* the signature-message pair is posted.

This weakening of the blindness property brings about the benefit that we can remove the blinding operations from the protocol (for a small efficiency increase) and also from the proof of correctness of the challenge generation (where the gain is more significant as it removes group exponentiations using foreign field arithmetic). The resulting scheme is similar in operation to the Signatures over Blocks of Committed Messages construction of Bobolz et. al. [7] with the addition of predicate checking.

7.1 Weakly blind predicate signatures

We adapt the definitions of [24] to account for the weak variant of blindness.

A WBPS scheme is parameterized by a family of polynomial-time-computable predicates, which are implemented by a p.t. algorithm P, the predicate compiler: on input a predicate description $prd \in \{0,1\}^*$ and a message $m \in \{0,1\}^*$, P returns 1 or 0 indicating whether m satisfies prd.

A WBPS scheme $\mathsf{WBPS}[P]$ for predicate P is defined by the following algorithms. We focus on schemes with 2-round (i.e., 4-message) signing protocols for concreteness.

- $Setup(1^\lambda) \to par$: the setup algorithm, on input the security parameter, outputs public parameters par, which define a message space $\mathcal{M}_{par}$.
- $KeyGen(par) \to (sk,vk)$: the key generation algorithm, on input the parameters par, outputs a signing/verification key pair (sk,vk), which implicitly contain par, i.e., $vk=(par,vk')$.
–
- $\langle Sign(sk,prd),User(vk,prd,m)\rangle \to (b,\sigma)$: an interactive protocol with shared input par (implicit in sk and vk) and a predicate prd is run between the signer and user. The signer takes a secret key sk as private input, the user's private input is a verification key vk and a message m. The signer outputs $d=1$ if the interaction succeeds and $d=0$ otherwise, while the user outputs a signature σ if it succeeds, and $\perp$ otherwise.
- $Ver(vk,m,\sigma) \to 0/1$: the (deterministic) verification algorithm, on input a verification key vk, a message m and a signature σ, outputs 1 if σ is valid on m under vk and 0 otherwise.

For a 2-round protocol the interaction $\langle Sign(sk,prd),User(vk,prd,m)\rangle \to (d,\sigma)$ can be realized by the following algorithms:

$$
\begin{array}{ll}
(txt_{U,0},st_{U,0}) \leftarrow User_0(vk,prd,m) & (txt_{S,1},st_S) \leftarrow Sign_1(sk,prd,txt_{U,0}) \\
(txt_{U,1},st_{U,1}) \leftarrow User_1(st_{U,0},txt_{S,1}) & (txt_{S,2},d) \leftarrow Sign_2(st_S,txt_{U,1}) \\
\sigma \leftarrow User_2(st_{U,1},txt_{S,2}) &
\end{array}
$$

We write $(d,\sigma) \leftarrow \langle Sign(sk,prd),User(vk,prd,m)\rangle$ as shorthand for the above sequence.

▶ **Definition 7.** *A WBPS scheme* $\mathsf{WBPS}[P]$ *satisfies weak blindness if for all p.p.t. adversaries* $\mathcal{A}$*:*

$$Adv^{\mathsf{BLD}}_{\mathsf{WBPS}[P],\mathcal{A}}(\lambda) := \Pr[\mathsf{BLD}^{\mathcal{A},1}_{\mathsf{WBPS}[P]}(\lambda) - \mathsf{BLD}^{\mathcal{A},0}_{\mathsf{WBPS}[P]}(\lambda)] \text{ is negligible in } \lambda.$$

We note that the BLD experiment mauls the message by appending a random bitstring r of length λ. This is appropriate to our setting, where the SP is explicitly allowed to use the "notes" field of a transaction to add a randomizer. Without this mauling, any indistinguishability-based definition would fail.

▶ **Definition 8.** . *A WBPS scheme* $\mathsf{WBPS}[P]$ *satisfies unforgeability if for all p.p.t. adversaries* $\mathcal{A}$

$$Adv^{\mathsf{EUF-CMA}}_{\mathsf{WBPS}[P],\mathcal{A}}(1^\lambda) := \Pr[\mathsf{CMA}^{\mathcal{A}}_{\mathsf{WBPS}[P]}(1^\lambda) = 1] \quad \text{is negligible in } \lambda.$$

7.2 Cavefish scheme

Our Scheme operates as follows: given an intent (i.e. predicate predicate description) $prd := int_{post}$ and predicate compiler P s.t. $(P(prd))(\mathsf{tx}) := \mathsf{chkSpec}(int_{post},\mathsf{mkAbs}(\mathsf{tx}))$ the requestor (i.e. the SP) encrypts the message, in our case the full transaction $m := \mathsf{tx}$ and sends the commitment C to the signer, who returns a random group element R used in the final Schnorr signature. The requestor replies with the Schnorr challenge c as well as a proof π of its correct construction, and of the fact that tx satisfies int_{post}.

We instantiate the scheme with a parametrisable NArg for the following relation,

$$\mathrm{R}_{\text{Cavefish}}\big(\underbrace{(q,\mathbb{G},G,\mathsf{H})}_{\text{parameters } par}, \overbrace{(X,R,com_{\text{tx}},TxAbs,c,int_{post})}^{\text{known statement } \theta}, \underbrace{(\mathsf{tx}||\mathsf{aux}_{\mathsf{nt}},\rho)}_{\text{witness } \omega}\big) \tag{1}$$

$\underline{\mathsf{BLD}^{\mathcal{A},b}_{\mathsf{WBPS}[P]}(\lambda)}$
$par \leftarrow Setup(1^\lambda)$
$(m_0, m_1, prd, vk', st_{\mathcal{A}}) \leftarrow \mathcal{A}_1(par)$
if $P(prd, m_0) = 0$ **or** $P(prd, m_1) = 0$
then **return** 0
$\mathsf{aux_{nt}} \leftarrow \{0,1\}^\lambda$
$m \leftarrow m_b || \mathsf{aux_{nt}}$
$vk \leftarrow (par, vk')$ $sess \leftarrow \mathsf{init}$
$b^* \leftarrow \mathcal{A}_2^{\mathsf{ChalUser}}(st_{\mathcal{A}})$
return $b = b^*$

$\underline{\mathsf{ChalUser}(msg = \emptyset)}$
if $sess = \mathsf{await}$
  $sess \leftarrow \mathsf{closed}$
  $\sigma \leftarrow User_2(st_u, msg)$
if $sess = \mathsf{closed}$
  $msg' \leftarrow Ver(vk, m, \sigma)$
if $sess = \mathsf{open}$
  $sess \leftarrow \mathsf{await}$
  $(msg', st_u) \leftarrow User_1(st_u, msg)$
if $sess = \mathsf{init}$
  $sess \leftarrow \mathsf{open}$
  $(msg', st_u) \leftarrow User_0(vk, prd, m)$
return msg'

Figure 4 Weak Blindness experiment $\mathsf{BLD}^{\mathcal{A},b}_{\mathsf{WBPS}[P]}(\lambda)$ for a WBPS scheme with predicate compiler P, adversary $\mathcal{A}$ and parameter b.

$\underline{\mathsf{CMA}^{\mathcal{A}}_{\mathsf{WBPS}[P]}(\lambda)}$
$par \leftarrow Setup(1^\lambda)$
$(sk, vk) \leftarrow KeyGen(par)$
$Q \leftarrow 0$
$S \leftarrow 0$
$P \leftarrow 0$
$(\vec{m^*}, \vec{\sigma}^*, \vec{prd^*}) \leftarrow \mathcal{A}^{SigInit, SigComplete}(vk)$
$n \leftarrow |\vec{m^*}|$
if $\prod_i Ver(vk, m_i^*, \sigma_i^*) \neq 1$ **return 0**
if $\prod_i prd_i^*(m_i^*) \neq 1$ **return 0**
if $\exists i,j : (i \neq j) \wedge (m_i^*, \sigma_i^*) = (m_j^*, \sigma_j^*)$ **return 0**
if $n > Q$ **return 1**
if $\nexists \rho \in Perm(S)$:
  $\forall i \leq n : (prd_{\rho(i)}(m_i) = 1) \wedge (st_{\rho(i)} = \perp)$
then **return 1**
return 0

$\underline{\mathsf{SigInit}(prd, msg_{U,0})}$
$S \leftarrow S+1; P \leftarrow P+1$
$(msg, st_S) \leftarrow Sign_1(sk, prd, msg_{U,0})$
$prd_S \leftarrow prd$
return msg

$\underline{\mathsf{SigComplete}(s, msg_{U,1})}$
if $s > S$ **or** $st_s = \perp$
then **return** $\perp$
$(msg, d) \leftarrow Sign_2(st_s, msg_{U,1})$
if $d = 1$ **then** $Q \leftarrow Q+1$
$P \leftarrow P-1$
$st_S \leftarrow \perp$
return msg

Figure 5 Chosen Message Unforgeability Experiment $\mathsf{CMA}^{\mathcal{A}}_{\mathsf{WBPS}[P]}(\lambda)$ for a WBPS scheme with predicate compiler P, adversary $\mathcal{A}$

checking that $\mathsf{chkSpec}(int_{post}, \mathsf{TxAbs}) = 1 \wedge m = \mathsf{tx}||\mathsf{aux_{nt}} \wedge C = \mathsf{PKE.Enc}(m;\rho) \wedge c = \mathsf{H}(R, X, m) \wedge \mathsf{TxAbs} = \mathsf{mkAbs}(\mathsf{tx})$. For the extended version, we alter the predicate compiler to (P_{ext} such that $(P_{\text{ext}}(prd))(\mathsf{tx}) := \mathsf{chkSpec}(int_{post}, \mathsf{tx})$ (i.e. we check tx rather than $\mathsf{mkAbs}(\mathsf{tx})$.

The signer completes the signing only if the proof verifies correctly. We present the full protocol in Figure 6.

7.3 Security

▶ **Theorem 9.** *The* WBPS$[P]$ *scheme of Figure 6 achieves weak blindness when the encryption scheme* PKE *is IND-CPA secure,* NArg *has the zero knowledge property, assumption 2 holds w.r.t. the hash generator* HGen.

Proof. We structure our proof as a sequence of games so that (1) two consecutive games are computationally indistinguishable to the adversary and (2) in the final game, the view of the adversary is independent of the challenge b. We define the initial game G_0 to be

$\underline{\mathsf{WBPS.Setup}}$	$\underline{\mathsf{WBPS.KeyGen}(par)}$
$(\mathbb{G}, g, q) \leftarrow \mathsf{GGen}(1^\lambda)$	$(\mathbb{G},g,q) :\subseteq par$
$\mathsf{H} \leftarrow \mathsf{HGen}(q)$	$x \leftarrow \mathbb{Z}_q; X \leftarrow g^x$
$sp \leftarrow (\mathbb{G},g,q,\mathsf{H})$	$sk \leftarrow (par,x)$
$(\mathrm{crs}, \tau) \leftarrow \mathsf{NArg.Setup}(sp)$	$vk \leftarrow (par,X)$
$(ek, dk) \leftarrow \mathsf{PKE.KeyGen}(1^\lambda)$	**return** (sk,vk)
$par \leftarrow (\mathrm{crs},ek)$	
return par	$\underline{\mathsf{WBPS.Ver}(vk,m,\sigma)}$
	$(\mathbb{G},g,q,\mathsf{H},X) :\subseteq vk$
	$(R,s) \leftarrow \sigma$
	$c \leftarrow \mathsf{H}(R,X,m)$
	return $(g^s = R \cdot X^c)$

$\underline{\mathsf{WBPS.Sign}(sk,prd)}$		$\underline{\mathsf{WBPS.Request}(vk,prd,m)}$
$(\mathbb{G},g,q,\mathrm{crs},ek,x) :\subseteq sk$		$(\mathbb{G},g,q,\mathsf{H},\mathrm{crs},ek,X) :\subseteq vk$
	$\xleftarrow{C}$	$C \leftarrow \mathsf{PKE.Enc}(ek,m;\rho)$
$r \leftarrow \mathbb{Z}_q; R \leftarrow g^r$	$\xrightarrow{R}$	$c \leftarrow \mathsf{H}(R,X,m)$
		$\theta \leftarrow (X,R,c,C,prd,ek)$
		$w \leftarrow (m,\rho)$
	$\xleftarrow{c,\pi}$	$\pi \leftarrow \mathsf{NArg.Prove}(\mathrm{crs},\theta,w)$
if $\mathsf{NArg.Ver}(\mathrm{crs},\theta,\pi) = 0$:		
return 0		
$s \leftarrow (r + c \cdot x) \mod q$	$\xrightarrow{s}$	**if** $g^s \neq R \cdot X^c$: **return** $\perp$
return 1		**return** $\sigma \leftarrow (R,s)$

Figure 6 The weakly blind predicate Schnorr signature scheme $\mathsf{WBPS}[P]$

$G_0 := \mathsf{BLD}^{\mathcal{A},b}_{\mathsf{WBPS}[P]}(\lambda)$. For G_1 we replace proofs with simulated ones. This is not detectable by the adversary due to the zero knowledge property of NArg. Second, for game G_2 we replace C with an encryption of a fixed message $\bar{m}$ rather than m_b, this too is undetectable[7] due to the IND-CPA security of PKE. For game G_3 we replace the challenge c to also use $\bar{m}$ rather than m_b in the hash. This is computationally indistinguishable due to assumption 2. ◀

▶ **Theorem 10.** *The* $\mathsf{WBPS}[P]$ *scheme of Figure 6 achieves unforgeability when the encryption scheme* PKE *is IND-CPA secure,* NArg *is sound, and assumption 1 holds w.r.t. the hash generator* HGen.

Proof. We follow the proof of [24] with little changes. As before, we structure our proof as a sequence of games so that the probability of adversarial success changes negligibly from one game to the next. We define the initial game G_0 to be $G_0 := \mathsf{CMA}^{\mathcal{A}}_{\mathsf{WBPS}[P]}(\lambda)$.

For game G_1, the experiment holds the decryption key for PKE, so that we extract the message from C ahead of time, and thus predict the c value once the value of R has been determined. If the value sent by the adversary is different from the predicted one, and the accompanying proof π verifies we abort early, diverging from the original game. This only

[7] This will also change the input of the simulator, but this is not an obstacle: we simply consider the simulator to be part of the adversary we are constructing against the IND-CPA security of PKE.

happens with negligible probability though, due to the adaptive soundness of NArg (if π does not verify, G_0 would abort as well).

For game G_2, we check to see if any of the adversarial signatures uses a message that belongs to a session that was not closed. If so, we abort early, diverging from G_1. However, this implies that the second part of the signature s along with the secret key x can be used to calculate the discrete log of R. Thus, if the early abort occurs (i.e. adversary manages to win using unfinished sessions) with non negligible probability we can build a discrete log calculator by embedding a discrete log challenge in one of the R values. Otherwise, we can assume that almost all of the adversarial wins use at least one "new" message, and thus the success probability for G_1 and G_2 differ only negligibly .

For G_3 we build a reduction to the EUF-CMA security of Schnorr (Assumption 1). We remove key generation from the challenger, and instead obtain a vk from the Schnorr EUF-CMA security game. When we decrypt a message m, we query it on our signing oracle, obtaining R,s as the signature. The reduction forwards R to the adversary anticipating c,π. The checks introduced in G_1 ensure the c value matches the predicted one, and thus s correctly completes the signature. This ensures the simulation can complete the game. If the adversary wins the security game $\mathsf{WBPS}[P]$ security game, there must be a signature on a new message that the reduction uses to also win the Schnorr EUF-CMA game with the same probability. ◀

8 Analysis & Applications

Storage and Communications Requirements. We point out that our protocol is optimal in terms of light client storage and light client & SP communication. The light client (LC) only needs to store information about which wallet subkeys have been active. In terms of communication, our protocol transmits only a constant number of elements when using a succinct proof system. This compares favorably to solutions such as SPV which incur linear costs or even solutions based on succinct proof systems for the chain tip, but where the LC needs to verify transaction inclusion into blocks via Merkle proofs.

Transaction discovery via HD Wallets. Assuming that the client only uses a single address can be unrealistic, as is requiring the LC to send the SP a list of addresses presumed active. Our protocol is compatible with BIP-32 [49] hierarchical deterministic (HD) wallets. This enables child addresses to be deterministically created from a parent address by hashing the parent key X, a chaincode value cc and an address index i. Concretely, $o_i := \mathsf{H}(X,\mathrm{cc},i)$ is the private key offset of child key i, i.e. the i-th child keypair is $x_i = x + o_i$ and $X_i = X \cdot g^{o_i}$. Child chaincodes are determined in the same way.

Thus, an LC can simply transmit the chaincode corresponding to its public key X, and the SP will be able to derive a list of child addresses (bounding indexes can be done heuristically, or with hints from the client). The protocol then proceeds with minor changes:

1. The challenge c is now derived as $c = \mathsf{H}(R,X_c,m)$
2. The witness w now includes the index of the child key i (or vector of indices $\boldsymbol{i}$ and chaincodes **cc** if the child is further down the tree).
3. The relation now checks that the hashed value is of the form $X_c = X \cdot g^{o_c}$ where $o_c = \mathsf{HDDerive}(X,\boldsymbol{i},\mathbf{cc})$.
4. The SP modifies the component $s := xc + r$ to $s' = s + o_c \cdot c$, adjusting for the child offset.

Notably, the LC does not learn which child address was used, it can simply sign with regards to the sent public key X and the SP can maul the signature as needed.

Asset and Non-Asset Compensation. In our model, we have so far assumed that an SP performs its services in exchange for a fee. This fee may be either flat, or calculated on the basis of transaction size, or complexity of transaction specification, etc.—the details of this fee calculation are dependent on the specifics of the implementation of our protocol, the marketplace, and SPs preferences. The fee may also be specified in either an amount of primary asset tokens, or some other user-defined tokens. A fee requirement in user-defined tokens could be useful in the case of the SP service for LCs being associated with another type of service trading in such tokens, such as a videogame token marketplace. Yet another option for SPs is to request non-monetary compensation for their services, such as requiring engagement with a specific smart contract (i.e. SP does DApp fee sponsorship), voting for a specific update, delegating stake, etc. All these actions can be performed by the very transaction that SP constructed according to LC's specification.

9 Implementation

To assess the efficiency of our construction, we implement and benchmark the NArg component of the Cavefish protocol. Given today's implementations of zk-SNARKs, running the NArg is expected to be the most time and resource intensive part of our light client protocol.

We base our tests on the trusted-setup zk-SNARK system *Groth16* [27] implemented by *Iden3* [30]. The circuits we construct and benchmark are inspired by the implementation of [24] and written in the domain-specific language of Circom 2.1.

The NArgwe implement captures the relation $\mathrm{R}_{\mathrm{Cavefish}}$ in the protocol in Section 7.2. We implement the "light" version of Cavefish, i.e., the intent is given by $int_{\mathrm{light}}(\mathsf{tx}) := \{\mathbf{return}\ (\mathsf{mkAbs}(tx) = \mathsf{mkAbs}(tx_0))\}$. The light client effectively specifies the transaction, except for the input UTxOs and note field aux.

9.1 Implementation choices

To implement and measure the arithmetic complexity of the relation $\mathrm{R}_{\mathrm{Cavefish}}$, we use BN254 as the curve for *Groth16* to operate on, i.e., the curve group has 254 bits and the relation is instantiated over an arithmetic circuit with modulus of 254 bits. The BN254 curve is one of the standard choices in Circom and forces the inputs to the circuit to be elements of the field given by BN254. We capture transaction tx and abstract transaction tx_A as bit strings of some length n encoded as field elements, which allows us to instantiate a circuit that can handle transactions of length up to n.

In order to implement the encryption scheme PKE that is needed to encrypt tx as C, we use ElGamal [21] public key encryption over the Baby-JubJub curve [48]. The reason for choosing Baby-JubJub curve is that the field given by BN254 is the base field of the Baby-JubJub curve and thus, any element can be represented as two elements of the BN254 field. As a consequence, the group operation is efficiently arithmetizable in the circuit [24]. Furthermore, we encapsulate key and encryption of ElGamal using DHIES [1], i.e., the shared secret in ElGamal serves as a seed to the PRF generating random group elements for an additive one-time-pad that encrypts tx. The PRF is instantiated as a Poseidon hash [26] that is efficiently arithmetizable by design.

Unfortunately, most common cryptographic hash functions are not "circuit-friendly" [26] and thus it would be beneficial to optimize the hash function used to create the Schnorr challenge in the blind signature protocol. However, to be compatible with existing blockchain platforms and create standard Schnorr signatures, we have to adopt the exact hash function specified by the respective ledger. We use the library of hash functions in [33] which provides Circom implementations for many popular hash functions. The authors put effort into the optimization, but more efficient implementations might be possible. Despite potential further improvements, the complexity of our resulting circuit is mainly governed by the number of rounds the hash function has to execute when producing the Schnorr challenge.

9.2 Benchmarks

We test and benchmark partially blind signatures for both Bitcoin and Cardano. Partial blindness can be interpreted as a predicate itself (see Sections 6.3 and 7.2). The "light" version of Cavefish uses int_{light} which can be implemented with substring equality checks, provided the transaction to be signed can be unambiguously separated into blinded and nonblinded parts. Cardano transactions are CBOR-encoded and contain the transaction-id and index for every UTxO serving as input [31]. Similarly, Bitcoin transactions (after Taproot update [19]) feature an input count followed by a list of inputs in binary format, each consisting of a transaction hash and output index. We assume for our benchmarks that one UTxO serves as input to the transaction and needs to be blinded.

We measure the number of contraints, the proving key size, and the proof size that $\mathrm{R}_{\text{Cavefish}}$ requires in Circom. The number of constraints are obtained from the arithmetization when given as a R1CS relation. We also keep track of the time it takes to create the resulting circuit, the proving time and the proof verification time.

The results are summarized in Table 1 for Bitcoin and Cardano as the target platform. The experiments were executed on commodity hardware based on an Intel(R) Core(TM) i7-8750H CPU operating at 2.20 GHz with 12 cores and 16 GB of RAM.

Table 1 Benchmark of $\mathrm{R}_{\text{Cavefish}}$ for Bitcoin and Cardano implementing the "light" version of Cavefish.

	Bitcoin	**Cardano**
Signature scheme	Schnorr	EdDSA
Curve	Secp256k1	Ed25519
Hash	SHA-256	SHA-512
Transaction size	254 B 288 b blinded	285 B 333 b blinded
Proving key size	115 MB	116 MB
Proving key verification time	18.6 s	20.5 s
Verification key size	67 kB	93 kB
Proving time	5.1 s	5.8 s
Proof size	806 B	805 B
Proof verification time	0.57 s	0.59 s
Number of constraints	226509	245181

Bitcoin has one major conceptual difference from Cardano as the message being signed by the Schnorr signature scheme is the hash of the transaction $\mathsf{H}(tx)$ instead of the transaction itself. Therefore, $\mathsf{WBPS}[P]$ is executed with $m := \mathsf{H}(tx||\mathsf{aux}_{\mathsf{nt}})$. A straightforward adaption to $\mathsf{WBPS}[P]$ that supports hashed transactions is to allow the predicate P to accept an additional input which is a witness attesting to the signed message m satisfying P. This additional witness must be included in the witness ω for $\mathrm{R}_{\text{Cavefish}}$ [24]. Therefore, compared to Cardano, the circuit for Bitcoin has to perform an additional invocation of the hash function (SHA-256). On the other hand, the hash function used by Cardano (SHA-512) has a larger codomain and higher complexity, which requires more constraints.

We remark that the projects our implementation builds upon are in development, such as the hash function library, and lack certain optimizations. Nevertheless, the introduction of a weak variant of blindness in our light client protocol allowed us to obtain results within practical bounds and shows that the Cavefish protocol is deployable in the real world.

References

1. Michel Abdalla, Mihir Bellare, and Phillip Rogaway. Dhies: An encryption scheme based on the diffie-hellman problem. *IACR Cryptol. ePrint Arch.*, 1999:7, 1999.
2. Frederik Armknecht, Ghassan Karame, Malcom Mohamed, and Christiane Weis. Practical light clients for committee-based blockchains, 2024. URL: `https://arxiv.org/abs/2410.03347`, `arXiv:2410.03347`.
3. Foteini Baldimtsi, Varun Madathil, Alessandra Scafuro, and Linfeng Zhou. Anonymous Lottery In The Proof-of-Stake Setting . In *2020 IEEE 33rd Computer Security Foundations Symposium (CSF)*, pages 318–333, Los Alamitos, CA, USA, June 2020. IEEE Computer Society. URL: `https://doi.ieeecomputersociety.org/10.1109/CSF49147.2020.00030`, `doi:10.1109/CSF49147.2020.00030`.
4. Bellare, Namprempre, Pointcheval, and Semanko. The one-more-rsa-inversion problems and the security of chaum's blind signature scheme. *Journal of Cryptology*, 16:185–215, 2003.
5. Fabrice Benhamouda, Tancrède Lepoint, Julian Loss, Michele Orrù, and Mariana Raykova. On the (in)security of ROS. Cryptology ePrint Archive, Paper 2020/945, 2020. URL: `https://eprint.iacr.org/2020/945`, `doi:10.1007/s00145-022-09436-0`.
6. Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. Cryptology ePrint Archive, Paper 2011/368, 2011. URL: `https://eprint.iacr.org/2011/368`.
7. Jan Bobolz, Jesus Diaz, and Markulf Kohlweiss. Foundations of anonymous signatures: Formal definitions, simplified requirements, and a construction based on general assumptions. In Jeremy Clark and Elaine Shi, editors, *Financial Cryptography and Data Security*, pages 121–139, Cham, 2025. Springer Nature Switzerland.
8. Dan Boneh and Chelsea Komlo. Threshold signatures with private accountability. Cryptology ePrint Archive, Paper 2022/1636, 2022. URL: `https://eprint.iacr.org/2022/1636`, `doi:10.1007/978-3-031-15985-5_19`.
9. Benedikt Bünz, Lucianna Kiffer, Loi Luu, and Mahdi Zamani. Flyclient: Super-light clients for cryptocurrencies. Cryptology ePrint Archive, Paper 2019/226, 2019. URL: `https://eprint.iacr.org/2019/226`.
10. Cardano Team. Full Cardano Ledger. `https://intersectmbo.github.io/formal-ledger-specifications/pdfs/cardano-ledger.pdf`, 2024.
11. Pyrros Chaidos and Aggelos Kiayias. Mithril: Stake-based threshold multisignatures. In Markulf Kohlweiss, Roberto Di Pietro, and Alastair Beresford, editors, *Cryptology and Network Security*, pages 239–263, Singapore, 2025. Springer Nature Singapore.
12. Rutchathon Chairattana-Apirom, Lucjan Hanzlik, Julian Loss, Anna Lysyanskaya, and Benedikt Wagner. Pi-cut-choo and friends: Compact blind signatures via parallel instance cut-and-choose and more. In *Annual International Cryptology Conference*, pages 3–31. Springer, 2022.
13. Manuel M. T. Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Jann Müller, Michael Peyton Jones, Polina Vinogradova, Philip Wadler, and Joachim Zahnentferner. $\text{UTXO}_{\mathsf{ma}}$: UTXO with multi-asset support. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Applications - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part III*, volume 12478 of *Lecture Notes in Computer Science*, pages 112–130. Springer, 2020. `doi:10.1007/978-3-030-61467-6_9`.
14. Manuel MT Chakravarty, Sandro Coretti, Matthias Fitzi, Peter Gaži, Philipp Kant, Aggelos Kiayias, and Alexander Russell. Fast isomorphic state channels. In *Financial Cryptography and Data Security: 25th International Conference, FC 2021, Virtual Event, March 1–5, 2021, Revised Selected Papers, Part II 25*, pages 339–358. Springer, 2021.
15. Panagiotis Chatzigiannis, Foteini Baldimtsi, and Konstantinos Chalkias. SoK: Blockchain light clients. Cryptology ePrint Archive, Paper 2021/1657, 2021. URL: `https://eprint.iacr.org/2021/1657`.

16 David Chaum and Torben Pryds Pedersen. Wallet databases with observers. In Ernest F. Brickell, editor, *Advances in Cryptology — CRYPTO' 92*, pages 89–105, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

17 Adrian Brink Christopher Goes, Awa Sun Yin. Anoma : a unified architecture for full-stack decentralized applications, 2022. URL: https://github.com/anoma/whitepaper/blob/main/whitepaper.pdf.

18 Arthur Britto David Schwartz, Noah Youngs. The ripple protocol consensus algorithm, 2018. URL: https://ripple.com/files/ripple_consensus_whitepaper.pdf.

19 Bitcoin Developers. Bip 340: Schnorr signatures for secp256k1. https://github.com/bitcoin/bips/blob/master/bip-0340.mediawiki, 2021. Accessed: 2025-05-28.

20 Ankur Dubey. A decentralised solver architecture for executing intents on evm blockchain. https://ethresear.ch/t/a-decentralised-solver-architecture-for-executing-intents-on-evm-blockchain/16608, 2023. Accessed: 2025-05-28.

21 Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE transactions on information theory*, 31(4):469–472, 1985.

22 Ergo Team. Ergo: A Resilient Platform For Contractual Money. https://whitepaper.io/document/753/ergo-1-whitepaper, 2019.

23 Georg Fuchsbauer, Antoine Plouviez, and Yannick Seurin. Blind schnorr signatures and signed elgamal encryption in the algebraic group model. In *Advances in Cryptology – EUROCRYPT 2020: 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10–14, 2020, Proceedings, Part II*, page 63–95, Berlin, Heidelberg, 2020. Springer-Verlag. doi:10.1007/978-3-030-45724-2_3.

24 Georg Fuchsbauer and Mathias Wolf. Concurrently secure blind schnorr signatures. In Marc Joye and Gregor Leander, editors, *Advances in Cryptology – EUROCRYPT 2024*, pages 124–160, Cham, 2024. Springer Nature Switzerland.

25 Ariel Gabizon, Zachary J Williamson, and Oana Ciobotaru. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. *Cryptology ePrint Archive*, 2019.

26 Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. Poseidon: A new hash function for Zero-Knowledge proof systems. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 519–535. USENIX Association, August 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/grassi.

27 Jens Groth. On the size of pairing-based non-interactive arguments. Cryptology ePrint Archive, Paper 2016/260, 2016. URL: https://eprint.iacr.org/2016/260.

28 Lucjan Hanzlik, Julian Loss, and Benedikt Wagner. Rai-choo! evolving blind signatures to the next level. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 753–783. Springer, 2023.

29 Dominik Hartmann and Eike Kiltz. Limits in the provable security of ECDSA signatures. Cryptology ePrint Archive, Paper 2023/914, 2023. URL: https://eprint.iacr.org/2023/914.

30 iden3. Circom: Circuit compiler for zero-knowledge proofs. https://github.com/iden3/circom, 2023. Accessed: 2025-05-28.

31 IntersectMBO. Cardano ledger specifications and implementations. https://github.com/IntersectMBO/cardano-ledger, 2025. Accessed: 2025-05-28.

32 Joseph Poon and Thaddeus Dryja. The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments. Technical report, University of Zurich, Department of Informatics, 01 2010.

33 Balazs Komuves. hash-circuits: Hashing circuits implemented in circom. https://github.com/bkomuves/hash-circuits, 2025. MIT License, maintained by Faulhorn Labs.

34 Yuan Lu, Qiang Tang, and Guiling Wang. Generic superlight client for permissionless blockchains. Cryptology ePrint Archive, Paper 2020/844, 2020. URL: https://eprint.iacr.org/2020/844.

35 Giulio Malavolta, Pedro Moreno-Sanchez, Aniket Kate, Matteo Maffei, and Srivatsan Ravi. Concurrency and privacy with payment-channel networks. In *Proceedings of the 2017 ACM*

SIGSAC Conference on Computer and Communications Security, CCS '17, pages 455–471, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3133956.3134096.

36 Gregory Maxwell, Andrew Poelstra, Yannick Seurin, and Pieter Wuille. Simple schnorr multi-signatures with applications to bitcoin. *Des. Codes Cryptography*, 87(9):2139–2164, September 2019. doi:10.1007/s10623-019-00608-x.

37 S. Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. https://bitcoin.org/en/bitcoin-paper, October 2008.

38 David Pointcheval and Jacques Stern. Security arguments for digital signatures and blind signatures. *J. Cryptology*, 13:361–396, 2000. doi:10.1007/s001450010003.

39 C. P. Schnorr. Efficient identification and signatures for smart cards. In Gilles Brassard, editor, *Advances in Cryptology — CRYPTO' 89 Proceedings*, pages 239–252, New York, NY, 1990. Springer New York.

40 Claus Peter Schnorr. Security of blind discrete log signatures against interactive attacks. In Sihan Qing, Tatsuaki Okamoto, and Jianying Zhou, editors, *Information and Communications Security*, pages 1–12, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

41 Ertem Nusret Tas, David Tse, Lei Yang, and Dionysis Zindros. Light clients for lazy blockchains, 2024. URL: https://arxiv.org/abs/2203.15968, arXiv:2203.15968.

42 Ethereum Team. Ethereum development documentation, 2023. URL: https://ethereum.org/en/developers/docs/.

43 Khalani Team. Khalani arcadia: An open platform for intent-driven agent collaboration, 2025. URL: https://khalani.network/aip_whitepaper.pdf.

44 Raiden Team. Raiden network 3.0.1 documentation, 2025. URL: https://raiden-network.readthedocs.io/.

45 Stefano Tessaro and Chenzhi Zhu. Short pairing-free blind signatures with exponential security. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 782–811. Springer, 2022.

46 Psi Vesely, Kobi Gurkan, Michael Straka, Ariel Gabizon, Philipp Jovanovic, Georgios Konstantopoulos, Asa Oines, Marek Olszewski, and Eran Tromer. Plumo: An ultralight blockchain client, 2025. White paper. URL: https://celo.org/papers/plumo.

47 David Wagner. A generalized birthday problem. In Moti Yung, editor, *Advances in Cryptology — CRYPTO 2002*, pages 288–304, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

48 Barry WhiteHat, Jordi Baylina, and Marta Bellés. Baby jubjub elliptic curve. *Ethereum Improvement Proposal, EIP-2494*, 29, 2020.

49 Pieter Wuille. BIP 0032: Hierarchical deterministic wallets. https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki#user-content-Full_wallet_sharing_m, 2012.

50 ScaleSphere Foundation Ltd. ("Foundation"). Celer network: Bring internet scale to every blockchain, 2018. URL: https://celer.network/doc/CelerNetwork-Whitepaper.pdf.

A Expanded Technical Backgroud

A.1 Discrete Log Groups

▶ **Definition 11.** *A group generator* GGen *is a probabilistic polynomial time (p.p.t.) algorithm with input a security parameter λ and outputs a group description $\mathbb{G},g,q$ such that $\mathbb{G}$ is a group of prime order $q \approx 2^\lambda$ with generator g. We say that the discrete logarithm problem is hard w.r.t.* GGen *if for all p.p.t $\mathcal{A}$ we have that*

$$\Pr[(\mathbb{G},g,q) \leftarrow \mathsf{GGen}(1^\lambda); t \leftarrow \mathbb{Z}_q; h \leftarrow g^t : t = A(\mathbb{G},g,q,h)] \text{ is negligible in } \lambda.$$

A.2 Public Key encryption and Schnorr Signatures

A public key encryption scheme PKE comprises a set of polynomial time algorithms KeyGen, Enc, Dec with the following syntax:

- $\mathsf{KeyGen}(1^\lambda) \to (ek, dk)$. Creates a public/private keypair ek, dk. The encryption key ek also defines the message space $\mathcal{M}$
- $\mathsf{Enc}(ek, m; \rho) \to C$. Encrypts a message $m \in \mathcal{M}$ under the public encryption key ek.
- $\mathsf{Dec}(dk, C) \to m$. Decrypts a ciphertext C using the private decryption key dk.

▶ **Definition 12.** *A public key encryption scheme* PKE *is* correct *if*

$$\Pr[(ek,dk) \leftarrow \mathsf{PKE.KeyGen}(1^\lambda); m \leftarrow \mathcal{M} : \mathsf{PKE.Dec}(sk, \mathsf{PKE.Enc}(pk,m)) = m] = 1.$$

▶ **Definition 13.** *A public key encryption scheme* PKE *is* IND-CPA secure *if for all stateful p.p.t. adversaries* $\mathcal{A}$*, the difference*

$$\left| \Pr[(ek,dk) \leftarrow \mathsf{PKE.KeyGen}(1^\lambda); (m_0,m_1) \leftarrow A(ek); d \leftarrow \{0,1\}; c^* \leftarrow \mathsf{PKE.Enc}(ek, m_d) : A(c^*) = d] - \frac{1}{2} \right|$$

is negligible in λ.

▶ **Definition 14.** *The Schnorr digital signature scheme* SDS *is defined for a group generator* GGen *and a hash function generator* HGen *and operates as shown on Figure 7.*

$\underline{\mathsf{SDS.Setup}(1^\lambda)}$
$(\mathbb{G}, g, q) \leftarrow \mathsf{GGen}(1^\lambda)$
$\mathsf{H} \leftarrow \mathsf{HGen}(q)$
$sp \leftarrow (\mathbb{G}, g, q, \mathsf{H})$
return sp

$\underline{\mathsf{SDS.KeyGen}(sp)}$
$(\mathbb{G}, g, q) :\subseteq sp$
$x \leftarrow \mathbb{Z}_q;\ X \leftarrow g^x$
$sk, vk \leftarrow (par, x), (par, X)$
return (sk, vk)

$\underline{\mathsf{SDS.Sign}(sk, m)}$
$(\mathbb{G}, g, q, \mathsf{H}, x) :\subseteq sk$
$r \leftarrow \mathbb{Z}_q;\ R \leftarrow g^x;\ c \leftarrow \mathsf{H}(R, X, m)$
$s \leftarrow (r + c \cdot x) \mod q$
return $\sigma \leftarrow (R, s)$

$\underline{\mathsf{SDS.Ver}(vk, m, \sigma)}$
$(\mathbb{G}, g, q, \mathsf{H}, X) :\subseteq vk$
$(R, s) \leftarrow \sigma; c \leftarrow \mathsf{H}(R, X, m)$
return $(g^s = R \cdot X^c)$

Figure 7 The Schnorr signature scheme SDS, with group and hash generators GGen, HGen.

▶ **Definition 15.** *A digital signature scheme* DS *is* correct *if*

$$\Pr[sp \leftarrow \mathsf{DS.Setup}(1^\lambda); (sk, vk) \leftarrow \mathsf{DS.KeyGen}(sp); m \leftarrow \mathcal{M}_S : \mathsf{DS.Ver}(vk, m, Sign(sk,m)) = 1] = 1.$$

▶ **Definition 16.** *A digital signature scheme* DS *is* strongly existentially unforgeable against chosen message attacks *(sEUF-CMA) if for all p.p.t. adversaries* $\mathcal{A}$*, we have*

$$\Pr[\mathsf{sEUF\text{-}CMA}^{\mathcal{A}}_{\mathsf{DS}}(1^\lambda)]$$

is negligible in λ *where the game* $\mathsf{GsEUF\text{-}CMA}^{\mathcal{A}}_{\mathsf{DS}}$ *is defined in Figure 8.*

$\underline{\mathsf{GsEUF\text{-}CMA}^{\mathcal{A}}_{\mathsf{DS}}(1^\lambda)}$	$\underline{\mathsf{OSig}(m)}$
$sp \leftarrow \mathsf{DS.Setup}(1^\lambda); Q \leftarrow \emptyset$	$\sigma \leftarrow \mathsf{DS.Sign}(skk,m)$
$(sk,vk) \leftarrow \mathsf{DS.KeyGen}(sp)$	$Q \leftarrow Q \cup \{(m,s)\}$
$(m^*,\sigma^*) \leftarrow \mathcal{A}^{\mathsf{OSig}}(vk)$	**return** σ
return $(m^*,\sigma^*) \neq Q \wedge \mathsf{DS.Ver}(vk,m^*,\sigma^*)$	

Figure 8 The security experiment $\mathsf{sEUF\text{-}CMA}^{\mathcal{A}}_{\mathsf{DS}}$ and supporting oracle OSig.

A.3 Parametrized Non-Interactive Zero Knowledge Arguments

Following [24], we define non-interactive zero knowledge arguments with regards to parametrized polynomial relations $\mathcal{P}:\{0,1\}^* \times \{0,1\}^* \times \{0,1\}^* \to \{0,1\}$, where the first argument represents a parameter set *par* (e.g. a group description). Given a value of *par*, we say that w is a witness for statement θ if $\mathcal{P}(par,\theta,w)=1$, i.e. $R = R_{par}(\theta,w) := \mathcal{P}(par,\theta,w)$ is an NP-relation, and $\mathcal{L} = \mathcal{L}_{par}$ is an NP-language. A NIZK for a relation $\mathcal{P}$ operates as follows:

- $\mathsf{Rel}(1^\lambda) \to par$. Generates a parameter set *par* that defines R and $\mathcal{L}$.
- $\mathsf{Setup}(par) \to (\mathrm{crs},\tau)$. Generates a common reference string (CRS) and trapdoor τ used by the simulator. We assume the CRS contains a description of R.
- $\mathsf{Prove}(\mathrm{crs},\theta,w) \to \pi$. Given a CRS crs, statement θ and witness w for θ, produces a proof π.
- $\mathsf{Ver}(\mathrm{crs},\theta,\pi) \to \{0,1\}$. Given a CRS crs, a statement θ and a proof π outputs 1 or 0, accepting or rejecting the proof.
- $\mathsf{SimProve}(\mathrm{crs},\theta,\tau)$. Given a CRS crs, statement θ and traproor τ for crs, produces a simulated proof π.

▶ **Definition 17.** *A system* $\mathsf{NArg}[R]$ *is perfectly correct if for all unbounded adversaries* $\mathcal{A}$

$$\Pr[par \leftarrow \mathsf{NArg.Rel}(1^\lambda); (crs,\tau) \leftarrow \mathsf{NArg.Setup}(par); (\theta,w) \leftarrow A(crs) : \\ \neg R(\theta,w) \vee \mathsf{NArg.Ver}(crs,\theta,\mathsf{NArg.Prove}(crs,\theta,w))] = 1$$

▶ **Definition 18.** *A system* $\mathsf{NArg}[R]$ *is adaptably computationally sound if for all p.p.t. adversaries* $\mathcal{A}$

$$\Pr[par \leftarrow \mathsf{NArg.Rel}(1^\lambda); (crs,\tau) \leftarrow \mathsf{NArg.Setup}(par); (\theta,\pi) \leftarrow A(crs) : \\ \neg L(\theta) \wedge \mathsf{NArg.Ver}(crs,\theta,\pi)] \text{ is negligible in } \lambda.$$

▶ **Definition 19.** *A system* $\mathsf{NArg}[R]$ *is computationally zero-knowledge if for all p.p.t. adversaries* $\mathcal{A}$

$$|\Pr[par \leftarrow \mathsf{NArg.Rel}(1^\lambda); (crs,\tau) \leftarrow \mathsf{NArg.Setup}(par); d \leftarrow \{0,1\} : \\ d = A^{\mathsf{OProve}_d}(crs)] - \frac{1}{2}| \text{ is negligible in } \lambda, \text{ where}$$

$$\mathsf{OProve}_0(\theta,w) := \textbf{\textit{if}}\ \neg R(\theta,w)\ \textbf{\textit{return}}\ \perp;\ \textbf{\textit{return}}\ \mathsf{NArg.Prove}(\theta,w), \textit{and}$$

$$\mathsf{OProve}_1(\theta,w) := \textbf{\textit{if}}\ \neg R(\theta,w)\ \textbf{\textit{return}}\ \perp;\ \textbf{\textit{return}}\ \mathsf{NArg.SimProve}(\theta,\tau).$$

B Additional Ledger Syntax

In Figure 9 we introduce the standard ledger syntax that we use throughout.

$\mathbb{H} = \bigcup_{n=0}^{\infty} \{0,1\}^{8n}$	the type of bytestrings
$(a,b) : \mathsf{Interval}[A]$	intervals over a totally-ordered set A
$\mathit{Key} \mapsto \mathit{Value} \subseteq \{\ k \mapsto v \mid k \in \mathit{Key},\ v \in \mathit{Value}\ \}$	finite map with unique keys
$[a1;\ldots;ak] : [C]$	finite list with terms of type C
$h :: t : [C]$	list with head h and tail t
$x \cup \mathsf{nothing} : A^{?}$	maybe type over A
$a\ \{\ \mathsf{field} = x\ \} : A$	record of type A with field changed to x

Figure 9 Notation

Figure 10 lists the primitives and derived types that comprise the foundations of the EUTxO model, along with some ancillary definitions. (Outputs normally refer to transaction IDs by hash, but we simplify here for clarity.)

LEDGER PRIMITIVES

$$\begin{array}{lll} \mathsf{checkSig} & : & \mathsf{Tx} \rightarrow \mathsf{PubKey} \rightarrow \mathbb{H} \rightarrow \{0,1\} \\ & & \textit{checks that a given key signed a transaction} \end{array}$$

HELPER FUNCTIONS

$$\mathsf{txid} : \mathsf{Tx} \rightarrow \mathsf{TxId}$$
$$\mathsf{txid}\ tx = \mathsf{hash}\ (tx\ \{\ \mathsf{sigKeys} = \ \mathsf{dom}\ (tx.\mathsf{sigs}),\ \mathsf{sigs} = \ \emptyset\ \})$$

$$\mathsf{toMap} : \mathbb{N} \rightarrow [\mathsf{TxOut}] \rightarrow (\mathbb{N} \mapsto \mathsf{TxOut})$$
$$\mathsf{toMap}(\_, [\,]) \qquad = [\,]$$
$$\mathsf{toMap}(ix,\ u\ ::\ outs) = \{\ ix \mapsto u\ \} \cup \mathsf{toMap}(ix+1,\ outs)$$
constructs a map from a list of outputs

$$\mathsf{mkOuts} : \mathsf{Tx} \rightarrow \mathsf{UTxO}$$
$$\mathsf{mkOuts}(tx) = \{\ (tx,\ ix) \mapsto o\ \mid\ (ix \mapsto o) \in\ \mathsf{toMap}(0,\ tx.\mathsf{outputs})\ \}$$
constructs a UTxO set from a list of outputs of a given transaction

$$\mathsf{MOf} : \mathbb{N} \rightarrow \mathbb{N} \rightarrow (A \rightarrow \{0,1\}) \rightarrow [A] \rightarrow \{0,1\}$$
$$\mathsf{MOf}\ k\ m\ f\ [\,] = m\ \leq\ k$$
$$\mathsf{MOf}\ k\ m\ f\ (h\ ::\ t) = \mathsf{if}\ (m\ \leq\ k)\ \mathsf{then}\ 1\ \mathsf{else}\ (\mathsf{MOf}\ (k\ +\ a)\ m\ f\ t)$$
$$\quad \textbf{where}\quad a\ =\ \mathsf{if}\ (f\ (h))\ \mathsf{then}\ 1\ \mathsf{else}\ 0$$
returns 1 *if enough elements of a list satisfy given function*

Figure 10 Primitives and basic types for the UTxO_{ma} model

B.1 Transaction Validation Rules

A transaction tx is *valid* if it follows the following rules.

(i) **The transaction has at least one input:**

$$tx.\mathsf{inputs}\ \neq\ \{\}$$

(ii) **The current slot is within transaction validity interval:**

$$slot \in tx.\mathsf{validityInterval}$$

(iii) **All outputs have positive values:**

$$\forall o \in tx.\mathsf{outputs},\ o.\mathsf{value} > \emptyset$$

(iv) **All output references of transaction inputs exist in the UTxO:**

$$tx.\mathsf{inputs}\ \subseteq\ \mathsf{dom}\ utxo$$

(v) **Value is preserved:**

$$tx.\mathsf{mint} + \sum_{i \in\ tx.\mathsf{inputs},\ (i\ \mapsto\ o) \in\ utxo} o.\mathsf{value} = \sum_{o \in\ tx.\mathsf{outputs}} o.\mathsf{value}\ +\ \mathsf{toValue}\ (tx.\mathsf{fee})$$

Constructors of Script

$$\begin{array}{rll}
\mathsf{RequireMOf} & : \mathbb{N} \to [\mathsf{Script}] & \to \mathsf{Script} \\
\mathsf{RequireSig} & : \mathsf{PubKey} & \to \mathsf{Script} \\
\mathsf{RequireTimeStart} & : \mathsf{Slot} & \to \mathsf{Script} \\
\mathsf{RequireTimeExpire} & : \mathsf{Slot} & \to \mathsf{Script}
\end{array}$$

Evaluation of Script

$$\begin{array}{rcl}
[\![_]\!] & : & \mathsf{Script} \to ((\mathsf{SetPubKey}) \times (Slot \times Slot)) \to \{0,1\} \\
[\![\mathsf{RequireMOf}\ n\ ls]\!](khs,(t1,t2)) & = & \mathsf{MOf}\ 0\ n\ ([\![_]\!]\ (khs,(t1,t2)))\ ls \\
[\![\mathsf{RequireSig}\ k]\!](khs,(t1,t2)) & = & k \in khs \\
[\![\mathsf{RequireTimeStart}\ t1']\!](khs,(t1,t2)) & = & t1' \le t1 \\
[\![\mathsf{RequireTimeExpire}\ t2']\!](khs,(t1,t2)) & = & t2 \le t2'
\end{array}$$

Figure 11 Script constructors and evaluation

(vii) **All inputs validate:**

$$\forall\ i \in tx.\mathsf{inputs},\ i \mapsto (s,v) \in utxo,\ [\![s]\!](\mathsf{dom}\ (tx.\mathsf{sigs}),\ tx.\mathsf{validityInterval}) = 1$$

(ix) **All minting scripts validate:**

$$\forall\ p \mapsto \_ \in tx.\mathsf{mint},\ [\![p]\!](\mathsf{dom}\ (tx.\mathsf{sigs}),\ tx.\mathsf{validityInterval}) = 1$$

(x) **All signatures are correct:**

$$\forall\ (pk \mapsto s) \in tx.\mathsf{sigs},\ \mathsf{checkSig}(tx,pk,s) = 1$$

(i) **The fee is sufficient:**

$$s.\mathsf{minfee} \le tx.\mathsf{fee}$$

B.2 Script construction and Evaluation

In Figure 11 we present the constructors and evaluation rules for scripts, and in Figure 12 we explain the transaction building function mkToSpec.

TRIVIAL TRANSACTION $\mathsf{initTx}_{a,mf}$

$$
\begin{aligned}
\mathsf{initTx}_{a,mf} \quad = \quad & \{\ \mathsf{inputs}=\emptyset, \\
& \mathsf{outputs}=[(a,\mathsf{tip})], \\
& \mathsf{validityInterval}=[\mathsf{nothing},\mathsf{nothing}], \\
& \mathsf{mint}=0, \\
& \mathsf{fee}=mf \\
& \mathsf{aux}=[] \\
& \mathsf{sigs}=\emptyset\ \}
\end{aligned}
$$

INPUT SELECTION FUNCTION

$$
\begin{aligned}
\mathsf{mkIns} \quad &: \quad (\mathsf{LState}\times(\mathsf{Set\,TxIn})\times\mathsf{Value}\times\mathsf{Script})\rightarrow(\mathsf{Set\,TxIn}) \\
\mathsf{mkIns}\ (l,\ i,\ v,\ s) \quad &= \quad \mathsf{if}\ \neg\ (v\leq 0)\ \mathsf{then} \\
&\qquad \mathsf{if}\ (v>0)\ \mathsf{then} \\
&\qquad\qquad \mathsf{mkIns}\ (l \setminus (\{j\mapsto\_\}),\ i\cup j,\ v\ -\ (u(j).\mathsf{value}),\ s) \\
&\qquad \mathsf{else}\ i \\
&\quad\ \mathsf{else} \\
&\qquad \mathsf{nothing} \\
&\quad\ \mathsf{where} \\
&\qquad j=\mathsf{pickInput}\ l\ s \\
&\qquad u=\mathsf{utxo}\ l
\end{aligned}
$$

AUXILIARY $\mathsf{mkToSpec}'$ DEFINITION

$$
\begin{aligned}
\mathsf{mkToSpec}' \quad &: \quad \mathsf{LState}\rightarrow\mathcal{I}_{\mathsf{post}}\rightarrow\mathsf{Tx}\rightarrow\mathsf{Tx}^{?} \\
\mathsf{mkToSpec}'\ l\ (\mathsf{MustMint}\ v)(tx) \quad &= \quad tx\ \{\mathsf{mint}\ =\ v\ +\ tx.\mathsf{mint},\mathsf{sigs}=tx.\mathsf{sigs}\cup\mathsf{getSigsVal}\ v, \\
&\qquad \mathsf{validityInterval}=\mathsf{restrictIntervalVal}\ tx.\mathsf{validityInterval}\ v\} \\
\mathsf{mkToSpec}'\ l\ (\mathsf{SpendFrom}\ s)(tx) \quad &= \quad tx\ \{\mathsf{outputs}\ =\ tx.\mathsf{inputs} \\
&\qquad \cup\ \mathsf{newIns},\mathsf{sigs}=tx.\mathsf{sigs}\cup\mathsf{getSigsUTxO}\ \mathsf{newIns}\ l, \\
&\qquad \mathsf{validityInterval}= \\
&\qquad\qquad \mathsf{restrictIntervalUTxO}\ tx.\mathsf{validityInterval}\ \mathsf{newIns}\ l\} \\
&\qquad\ \mathbf{where} \\
&\qquad \mathsf{newIns}=\mathsf{mkIns}\ l\ (tx.\mathsf{inputs})\ (\mathsf{produced}-\mathsf{consumed})\ s \\
\mathsf{mkToSpec}'\ l\ (\mathsf{MaxInterval}\ i)(tx) \quad &= \quad tx\ \{\mathsf{validityInterval}\ =\ (l.\mathsf{slot}, \\
&\qquad \mathsf{min}\ \{l.\mathsf{slot}\ +\ i,tx.\mathsf{validityInterval}_2)\}\} \\
\mathsf{mkToSpec}'\ l\ (\mathsf{PayTo}\ (s,v))(tx) \quad &= \quad tx\ \{\mathsf{outputs}\ =\ tx.\mathsf{outputs}\cup(s,v)\} \\
\mathsf{mkToSpec}'\ l\ (\mathsf{ChangeTo}\ s)(tx) \quad &= \quad \mathsf{if}\ \mathsf{consumed}-\mathsf{produced}\ >\ 0 \\
&\qquad \mathsf{then}\ tx\ \{\mathsf{outputs}\ =\ tx.\mathsf{outputs} \\
&\qquad \cup\ \{(s,\ \mathsf{consumed}-\mathsf{produced})\}\}\ \mathsf{else\ nothing} \\
\mathsf{mkToSpec}'\ l\ (\mathsf{MaxFee}\ f)\ (tx) \quad &= \quad \mathsf{if}\ tx.\mathsf{fee}\leq f \\
&\qquad \mathsf{then}\ tx\ \mathsf{else\ nothing} \\
\mathsf{mkToSpec}'\ l\ (\mathsf{AndExps}\ [a1;a2;...;ak])(tx) \quad &= \quad \mathsf{mkToSpec}'\ l\ ak\ (...\ (\mathsf{mkToSpec}'\ l\ a2\ (\mathsf{mkToSpec}'\ l\ a1\ tx)))
\end{aligned}
$$

$\mathsf{mkToSpec}$ DEFINITION

$$
\begin{aligned}
\mathsf{mkToSpec} \quad &: \quad (\mathsf{LState}\times\mathcal{I}_{\mathsf{post}})\rightarrow\mathsf{Tx}^{?} \\
\mathsf{mkToSpec}\ (l,i) \quad &= \quad \mathsf{mkToSpec}'\ l\ i\ \mathsf{initTx}_{a,mf}
\end{aligned}
$$

Figure 12 Building transactions according to the specific intent

$$
\begin{aligned}
\mathsf{txi}_1 \quad = \quad & (\mathsf{outputs} = \{\ (s,t)\ \}, \\
& \mathsf{validityInterval} = [\mathsf{slot}\ l, (\mathsf{slot}\ l) + j], \\
& \mathsf{mint} = t, \\
& \mathsf{fee} = \mathsf{minfee}\ l \\
& \mathsf{sigKeys} = \mathsf{getSignersVal}\ t) \\
\\
\mathsf{txi}_2 \quad = \quad & (\mathsf{outputs} = \{(\mathsf{RequireSig}\ k2, x)\ ,\ (\mathsf{RequireSig}\ k1\ , \\
& (\mathsf{balance}\ (\mathsf{mkIns}\ l\ \{\}\ x\ (\mathsf{RequireSig}\ k1))) - x)\}, \\
& \mathsf{validityInterval} = [\ \mathsf{nothing}\ , \mathsf{nothing}\ ], \\
& \mathsf{mint} = \{\ \}, \\
& \mathsf{fee} = \mathsf{minfee}\ l \\
& \mathsf{sigKeys} = \{\ k1\ \})
\end{aligned}
$$

Figure 13 Abstract transaction examples