Cavefish: Communication-Optimal Light Client Protocol for UTxO Ledgers

Aggelos Kiayias[1,2], Marc Roeschlin[2], Polina Vinogradova[2], Daniel Morales[4,2], and Pyrros Chaidos[3,2]

[1] University of Edinburgh, UK
[2] Input Output
[3] National & Kapodistrian University of Athens, Greece
[4] Decentralized Security

Abstract. Blockchain light clients (LCs) are agents with limited computational or storage resources that cannot maintain a fully validated copy of the ledger. They rely on service providers (SPs), typically full nodes, to access data required for tasks such as constructing a transaction (Tx) or interacting with off-chain applications.
We introduce Cavefish, a novel protocol for UTxO-based platforms that enables LCs to interact with the ledger and submit a transaction (Tx) with minimal trust, storage, and computation, without having to synchronize to the chain. The LC specifies a Tx (e.g., by specifying a source address rather than source UTxOs, a destination and value, and a change address), and the SP constructs it, including a payment to the SP as an additional output. The LC needs to verify Tx before signing it, but the SP cannot reveal it without fear of losing its compensation.
In order to resolve this two-sided trust problem, we propose a variant of the predicate blind signature (PBS) scheme of Fuchsbauer and Wolf (Eurocrypt 2024), which enables the SP to obtain valid Schnorr signatures on Tx, but only after proving to LC that Tx satisfies the specification.
Cavefish achieves a trustless interaction in which the LC fulfills their transaction goal, and the SP receives fair compensation for their effort. As transactions only need to stay private until posted, our PBS variant relaxes the unlinkability requirements of blind signatures. We implement and benchmark the Non-interactive Argument of Knowledge component of Cavefish on two major UTxO-based blockchains, using two different zero-knowledge proving systems.

Keywords: Blockchain · Light clients · UTxO · Blind signatures · Zero-knowledge proofs

1 Introduction

Blockchains are distributed append-only data structures that grow continuously over time. They offer strong guarantees of data integrity, censorship resistance, and fault tolerance, but only if full nodes download and validate the entire chain history to ensure correctness and security. This presents a barrier 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 without maintaining full historical data. These protocols must strike a balance between minimizing resource consumption and preserving critical security properties.

We present *Cavefish*, a novel intent-based LC protocol built on top of the *Unspent Transaction Output* (UTxO) [46,37] ledger model, providing robust formal security guarantees regarding transaction validation and parallelization. Cavefish performs a two-party computation that yields a signed transaction identical to one created by a full node. In addition to low storage and computational requirements, it is communication-optimal and can be completed in as few as two rounds (after initial intent submission) without having to synchronize with the chain. The communication costs of the protocol consist of (i) a transaction specification, (ii) a blinded version of the transaction, randomizer & a proof that it matches the specification and (iii) a signature sent to the SP. The transaction itself carries an overhead of one extra output, paying the SP. The only state the LC needs to maintain is the public key address(es) of its funds. Additionally, Cavefish is compatible with hierarchical (HD) wallets [62] that allow easy address discovery over a range of child addresses, with the LC announcing only a single public key together with its chain code. These are sufficient for a service provider (SP) to scan the UTxO ledger and construct a transaction according to the specifications requested by the LC.

Because Cavefish constitutes a transaction-building protocol for LCs, it must resolve a two-sided trust problem: the LC needs assurance that the SP will not construct a malicious or unintended transaction, while the SP needs assurance that the LC cannot obtain the complete transaction, strip out or evade the service payment, and submit it independently. This is precisely why Cavefish is designed as a two-party computation protocol built on blind-signing and predicate-proof mechanisms.

The protocol is initiated by the LC that sends an intent and one or more of their previously used addresses to the service provider (SP). An *intent* (Section 5) is a structured description of what a transaction should accomplish. The SP builds a transaction that satisfies the intent, with the input funds coming from the address(es) specified by the LC. If the SP were to share the constructed intent-fulfilling transaction in the clear, a freeloading LC could sign and submit the transaction to the chain on its own, evading the SP's reimbursement. Cavefish includes the reimbursement to the SP in the transaction itself; therefore, it does not require any additional setup. In fact, Cavefish can be used to initialize more complex protocols, such as a payment channel.

Enforcing compensation to the SP is achieved by hiding the full transaction and leads to the cryptographic complexity of Cavefish. The LC and SP complete a blind signature protocol where the LC verifies if the transaction satisfies its defined specifications, and only then creates valid signature(s) that are sent back to the SP. The SP attaches the signatures to the transaction and posts it on the blockchain on behalf of the LC. The SP may send a clear copy of the transaction

to the LC afterward to provide assurance that it has been posted and is, therefore, not possible to replay.

To make our scheme viable, we adapt the blind predicate Schnorr signature scheme from [31]. 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 anonymity set for a blinded transaction is trivially small, and thus the transaction signed by the LC only needs to stay *private until posted*. Due to this, we introduce the notion of a *weakly* blind predicate signature scheme in Section 6.1.

Cavefish allows any device with access to a user's HD key to post a transaction without the need to have ever accessed the user's UTxO set. This is useful for cold/non-custodial wallets that are not (continuously) synchronized with the chain, or if a user transacts with multiple devices (laptop, cellphone, etc.) that do not share wallet states.

To summarize our contributions:

- We propose Cavefish, a LC protocol with minimal set-up and chain state querying, that maintains safety, protects the SP from freeloading, and imposes minimal communication overhead. It is compatible with standard Schnorr signatures and variants such as EdDSA [9]. We also specify a multi-SP protocol version for increased liveness.
- We introduce weakly blind predicate signatures (motivated by our "private until posted" goal), which combines blind predicate signatures of [31] and signatures on committed messages (SBCM) [12].
- We benchmark the zero-knowledge part of Cavefish for Bitcoin and Cardano and show that even a prototype "unoptimized" implementation achieves proving and verification times that are viable in a real-world deployment.

2 Related Work

A "traditional" light client has the primary goal of obtaining some (cryptographically guaranteed or trusted) data about the current state of the blockchain [5,21]. To verify the correctness of the data a service provider (SP) sends to a LC, existing designs rely on a single-prover or a multi-prover model. Single-prover LCs perform cryptographic verification of the SP's response. The Simplified Payment Verification (SVP) approach [46] is an example of single-prover light client design for UTxO-based ledgers. It requires the LC to verify block headers of the longest chain, but not the block bodies. This attracts a linear communication cost by itself, which is compounded by the need to query bloom filters and download blocks. The newer Neutrino (BIP 157) solution adds privacy at a significant size overhead (approximately 1GB per year queried).

Other single-prover light clients use succinct representations and proofs, such as data accumulators (often Merkle trees), aggregate signatures and threshold signatures [18,5], and potentially SNARKs (e.g. to aggregate multi-signatures [57]). In such protocols the communication cost can range from linear (sparse certificate chain as in Plumo [32]), to logarithmic (e.g. NIPoPoWs [40]) to constant

using recursive SNARKs. In contrast to SPV or Neutrino, such protocols only allow the client to discover the current tip of the chain and enough metadata to evaluate past and future blocks. Recovering one's past transactions is beyond their scope, and attracts additional communication costs.

Multi-prover light clients, on the other hand, are able to trust the chain state supplied by a service provider if the data is confirmed by a sufficiently large set of full nodes [53,50]. Cavefish can follow a single-prover approach or be extended to a multi-SP variant as described in Sec. 8, both variants do not require chain synchronization. Furthermore, in traditional light client protocols, the client issues transaction queries after chain sync at an additional cost. In the case of Cavefish we avoid the need for a large reply by specifying a transaction intent as opposed to re-downloading all past transactions, often with significant overheads (i.e whole blocks or individual inclusion proofs).

Some light clients use game-theoretic approaches by implementing the slashing of deposited collateral in case of misbehavior [43]. Unlike in the design we propose, such a payment scheme is not atomic for each transaction. Another common technique is to compress the blockchain and/or ledger state to reduce the information that a light client requires to function reliably [18,13,32]. This, too, is different from our model, which does not require any chain data to be sent to the client.

Block explorers and APIs For UTxO ledgers (e.g. Ergo [52], Cardano [14], and Bitcoin [10]) transaction construction can safely rely on centralized services such as blockchain explorers [15] or APIs like Blockfrost [51]. Light wallets such as VESPR [58] operate on this principle. VESPR queries Blockfrost; and then constructs a transaction for the user which includes fees for using VESPR. A smart user can use a hardware wallet such as Keystone to inspect a VESPR transaction in full, and remove any fee payments before submitting it on chain. This vulnerability appears to not yet have been exploited for social rather than technical reasons. The multi-SP variant of Cavefish is decentralized, and service fees cannot be removed. Such a scheme could co-exist alongside the free Explorer or API service model and serve as a template for establishing universal monetization and decentralization of such services.

Intents Solver networks attempt to establish a relationship between solvers and users via their (light) clients [27]. A client issues an intent as an abstracted transaction object. Then, solvers process the intent, incentivized via some intent execution reward. Ours appears to be the first work to describe an intent-based light client protocol taking advantage of specific features of the UTxO ledger model that both minimizes communication and offers a built-in incentive structure. Some protocol designs choose to resolve intents on-chain directly [54], while others offer both layer-1 and layer-2 options [23]. Another design is a consensus protocol relying on minimal trust assumptions while conforming to the interledger standard [24].

Payment Channels Like Cavefish, payment channels are an off-chain protocol that can be used to conduct a series of transactions without interacting with the main ledger [38,55,53]. The creation of a payment channel requires the locking of

funds through an on-chain transaction [44]. *State channels* (such as [20]) not only allow simple payments between users but also internally simulate the on-chain transaction processing mechanism. Entering a state or payment channels require a user to be able to submit a set-up transaction. Once initialized and entered into by all required parties, they may be cheaper/faster than Cavefish, and similarly reliable. Our design interacts well with payment channels: In the single-SP variant, Cavefish is aware of the SP who posts the final transaction, and it can be used to set up a payment channel. The SP can be compensated through the channel and avoids additional on-chain overhead of subsequent uses of Cavefish.

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 [39]. 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 [39], and we use pseudorandom permutations [39]. For signatures, we use Schnorr signatures ($\mathsf{SDS.Setup}, \mathsf{SDS.KeyGen}$, $\mathsf{SDS.Sign}, \mathsf{SDS.Ver}$) [48,47], which we require to be strongly existentially unforgeable (sEUF-CMA) [39]. Finally, we parametrize the 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, following [31]. We use these primitives in their standard form and refer the reader to the cited works for their formal definitions; the two non-standard assumptions specific to our setting are stated bellow. We also provide an overview of the ZK primitives on which we build our protocol.

3.1 Cryptographic Assumptions

Our construction relies on a number of standard cryptographic primitives: discrete-logarithm-hard groups, pseudorandom permutations, IND-CPA-secure public-key encryption, strongly existentially unforgeable digital signatures, Schnorr signatures, and parametrized non-interactive zero-knowledge arguments. We use these in their standard form; pointers to references with their formal definitions are given in the Technical Background (Section 3). Below we state the two assumptions that are specific to our setting and not part of the standard treatment of these primitives.

The security of Schnorr signatures has been well studied in the random oracle model (ROM), where the hash is 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 [6], 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 in fact drawn from a small pool for standardized options), and also from the treatment of this issue in the literature [31].

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. [31] There exists a group generator GGen and hash function generator HGen such that the Schnorr signature scheme SDS is strongly existentially unforgeable under chosen-message attacks (sEUF-CMA) [39].

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

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

is negligible in λ.

3.2 Schnorr Blind Signatures

Blind Schnorr signatures originate with [22]. Most variants are interactive, and several concurrently-secure constructions were later broken by polynomial-time attacks on the ROS problem [60,8,49]. Cavefish builds on the concurrently-secure predicate blind Schnorr scheme of Fuchsbauer and Wolf [31], which is the first practical such scheme which is resistant to those attacks, as well as having other rigorous security guarantees stated in the paper. From it we inherit *partial* and *predicate* blindness, which let the SP present a transaction in abstract, partly "redacted" form while the LC still verifies it against its intent.

3.3 Parametrized Non-Interactive ZK Arguments.

Following [31], we define non-interactive zero knowledge (ZK) arguments with regard to *p*arametrized 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 $\mathcal{P}(\text{par}, \theta, w) = 1$, i.e. $R = R_{\text{par}}(\theta, w) := \mathcal{P}(\text{par}, \theta, w)$ is an NP-relation, and $\mathcal{L} = \mathcal{L}_{\text{par}}$ is an NP-language. A NIZK for a relation $\mathcal{P}$ operates as follows:

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

We use standard definitions for correctness, soundness, and zero knowledge; see [31].

4 Ledger Model

The UTxO ledger model, such as the one used by Bitcoin [46], Ergo [29], and Cardano [17], maintains a record, called the *UTxO set*, of transaction outputs added by transactions that have been applied throughout its history, but not yet spent. We tailor our design to (a pared down version of) the extended UTxO ledger model with support for user-defined scripts and transacting in multiple types of assets (EUTxO_{ma}), specified in previous work [19,59]. Here we discuss only the features our protocol relies on directly, point out any simplifications, and outline the relationship between transactions, blocks, and valid ledger states. For completeness, we give the details of the relevant primitives in Appendix A, and the formal specification of the ledger in Appendix B and C.

Blocks and Ledger States. A *ledger state* is a data structure which is updated by applying blocks ($\mathsf{Block} = \mathsf{Header} \times [\mathsf{Tx}]$) incoming on the network. The ledger state LState contains (among other data we do not model) the UTxO set field $\mathsf{utxo} : \mathsf{LState} \to \mathsf{UTxO}$, and the minimum transaction fee $\mathsf{minfee} : \mathsf{LState} \to \mathbb{N}$. A *full node* (1) has a verified checkpoint ledger state (e.g. the genesis state), (2) updates the ledger state by validating and applying all incoming blocks since that checkpoint, and (3) is able to produce, send, and receive blocks on the network. We assume the SP is such a node.

Transactions. A (simplified) EUTxO_{ma} transaction structure is as follows:

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

The UTxO set is a finite map from inputs $i = (\mathit{txid}, \mathit{ix}) \in \mathsf{TxIn} := \mathsf{TxId} \times \mathbb{N}$ to outputs $o \in \mathsf{TxOut}$. The transaction's ID $\mathit{txid} \in \mathsf{TxId}$ is its unique identifier, computed by hashing it (excluding its signatures). A transaction input $i \in \mathit{tx}.\mathsf{inputs}$ is a pointer to the UTxO map entry indexed by that input. The set of inputs of a transaction is used to specify which UTxO entries to remove when the transaction is processed. Also at that time, the transaction output $(s, v, d) \in \mathsf{TxOut} := \mathsf{Script} \times \mathsf{Value} \times \mathsf{Data}$ is added to the UTxO set with the

input identifier $i = (txid, ix)$, where $txid$ is the ID of the transaction adding the output, and ix is its position in the output list.

The components of (s, v, d) are as follows: (1) $s \in \mathsf{Script}$ represents both the address of the output, and the conditions a transaction must satisfy to spend the output (discussed below), (2) $v \in \mathsf{Value}$, which is a finite set of user-defined currencies and their corresponding quantities, and (3) a bytestring $d \in \mathsf{Data}$, specified by the user at the time of transaction construction, which can be used to represent contract state.

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 transaction (excluding sigs itself). The fee is the amount of primary currency a transaction pays as a system fee, which must be at least minfee. The Value in the mint field specifies the assets being minted (positive quantities) or burned (negative quantities) by the transaction. The aux is a field for arbitrary extra data encoded as a bytestring, e.g. a "note".

Script Construction and Evaluation. A script $s \in \mathsf{Script}$ specifies constraints on a transaction that must be satisfied in order to allow a specific action the transaction is performing, specifically, either spending a UTxO entry or minting currency. In an EUTxO model, script evaluation is not a state-updating computation, it merely outputs 0 (forcing transaction validation to fail) or 1 (allowing validation to continue). We use a small DSL (see Fig. 9 in Appendix C, or the DSL implementation in Cardano [37]) for building scripts – this is a simplification from the full EUTxO model, which uses a Turing-complete language. This DSL is distinct from the DSL used to specify light clients' intents in 5.

Ledger State Update. The function $\mathsf{updateUTxO} : \mathsf{Slot} \times \mathbb{N} \times \mathsf{UTxO} \times \mathsf{Tx} \to \mathsf{UTxO}$, for a given $tx \in \mathsf{Tx}$, checks its validity against the given slot, fee, and UTxO state, then, if it is valid, updates the UTxO by adding and removing entries. Validity of tx is checked by the function $\mathsf{checkTx}((slot, fee), utxo, tx) = 1$, defined formally in Appendix B. A block is applied by updating the ledger state with its transactions applied in sequence, rejecting the whole block if any transaction fails.

5 Intent specification

The primary goal of a Cavefish light client is to post a transaction on-chain that satisfies its intent. The intent a light client constructs is any expression in the DSL $\mathcal{I}_{\mathsf{post}}$, whose constructors are as follows:

MustMint	$: \mathsf{Value} \to \mathcal{I}_{\mathsf{post}}$	ChangeTo	$: [\mathsf{TxOut}] \to \mathsf{Script} \to \mathcal{I}_{\mathsf{post}}$
SpendFrom	$: \mathsf{Script} \to \mathcal{I}_{\mathsf{post}}$	MaxFee	$: \mathbb{N} \to \mathcal{I}_{\mathsf{post}}$
MaxInterval	$: \mathsf{Slot} \to \mathcal{I}_{\mathsf{post}}$	SignedBy	$: \mathsf{PubKey} \to \mathcal{I}_{\mathsf{post}}$
PayTo	$: [\mathsf{TxOut}] \to \mathcal{I}_{\mathsf{post}}$	AndExps	$: [\mathcal{I}_{\mathsf{post}}] \to \mathcal{I}_{\mathsf{post}}$

To fulfil the intent, the SP constructs (if possible) a $tx \in \mathsf{Tx}$, then sends to the LC an *abstract transaction*, $tx_A \in \mathsf{TxAbs}$, which is missing some of the data in

the original tx:

$$\begin{aligned}\mathsf{TxAbs} = (&\mathsf{outputs} : [\mathsf{TxOut}],\ \mathsf{validityInterval} : \mathsf{Interval}[\mathsf{Slot}],\\ &\mathsf{mint} : \mathsf{Value},\ \mathsf{fee} : \mathbb{N},\ \mathsf{sigKeys} : \mathsf{Set\ PubKey})\end{aligned}$$

Not divulging this data is required for the correct operation of Cavefish (see Section 6). TxAbs conceals some transaction data in a way that (i) allows the light client to verify that their intent has been resolved in the original transaction and (ii) is not sufficient to construct a valid transaction that avoids paying the service provider fee. Compared to Tx, the aux field 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, as given by the constructor:

$$\begin{aligned}\mathsf{mkAbs} :&\ \mathsf{Tx} \to \mathsf{TxAbs}\\ \mathsf{mkAbs}\ tx = &\ tx\ \{\ \mathsf{outputs} = tx.\mathsf{outputs},\ \mathsf{outputs}[0].\mathsf{Value} = -1,\\ &\ \mathsf{validityInterval} = tx.\mathsf{validityInterval},\ \mathsf{mint} = tx.\mathsf{mint},\\ &\ \mathsf{fee} = tx.\mathsf{fee},\ \mathsf{sigKeys} = \mathsf{dom}\ (tx.\mathsf{sigs})\ \}\end{aligned}$$

The intent mechanism for the Cavefish protocol consists of two functions: fulfilment of a client's specification (done by the SP), and verification that the intent was fulfilled correctly by the resulting tx_A (done by the LC).

Fulfilment. The function $\mathsf{mkToSpec} : \mathsf{LState} \times \mathcal{I}_{\mathsf{post}} \to \mathsf{Tx}^?$ constructs a transaction based on the specification (this function may fail, outputting nothing instead of a transaction, if the intent cannot be resolved), see Figs. 10 and 11. It works by editing an initial transaction $\mathsf{initTx}_{a,mf}$, which has no inputs and includes a minimum fee mf as well as a tip to the service provider, output to the SP's address a; all its remaining fields are empty.

The $[\![_]\!]_{\mathsf{DSL}}$ and mkToSpec functions are defined such that for any $l,\ i$, where $\mathsf{mkToSpec}\ (l,\ i) \neq \mathsf{nothing}$, necessarily $[\![i]\!]_{\mathsf{DSL}}\ (\mathsf{mkAbs}\ (\mathsf{mkToSpec}\ (l,\ i))) = 1$. When this holds, we say that i is satisfied by the abstract transaction $tx_A := \mathsf{mkAbs}\ (\mathsf{mkToSpec}\ (l,\ i))$. Note here that whether a valid tx_A satisfying i *exists* depends on the choice of l, but, given a tx_A, an LC can check whether or not it satisfies its i without knowing l (using chkSpec). The constructed tx_A may not be unique in its satisfaction of i, however, the mkToSpec function gives a deterministic way to construct tx_A. The multi-SP version of our protocol relies on most SPs choosing to use mkToSpec, see Section 8.

Verification. The function $[\![_]\!]_{\mathsf{DSL}} : \mathcal{I}_{\mathsf{post}} \to \mathsf{TxAbs} \to \{0, 1\}$ checks that a given abstract transaction matches the intent specification, defined per constructor as follows (writing $L_1 \ominus L_2$ for the multiset difference of lists L_1, L_2, i.e. removing

each element of L_2 from L_1 once):

$$
\begin{aligned}
&[\![\mathsf{MustMint}\ v]\!]_{\mathsf{DSL}} = v \leq tx.\mathsf{mint}\\
&[\![\mathsf{SpendFrom}\ s]\!]_{\mathsf{DSL}} = [\![s]\!]\ tx\\
&[\![\mathsf{MaxInterval}\ i]\!]_{\mathsf{DSL}} = (tx.\mathsf{validityInterval})_2 - (tx.\mathsf{validityInterval})_1 \leq i\\
&[\![\mathsf{PayTo}\ [(s,v,d)]]\!]_{\mathsf{DSL}} = (s,v,d) \in tx.\mathsf{outputs}\\
&[\![\mathsf{ChangeTo}\ p\ s]\!]_{\mathsf{DSL}} = (s,v,\text{“”}) \in tx.\mathsf{outputs},\\
&\quad \textbf{where}\ \ v = \mathsf{sumVals}\ (tx.\mathsf{outputs}) - \mathsf{sumVals}\ p\\
&[\![\mathsf{MaxFee}\ f]\!]_{\mathsf{DSL}} = tx.\mathsf{fee} \leq f\\
&[\![\mathsf{SignedBy}\ k]\!]_{\mathsf{DSL}} = k \in tx.\mathsf{sigKeys}\\
&[\![w]\!]_{\mathsf{DSL}} = \textstyle\bigwedge_{j=1}^{k} [\![a_j]\!]_{\mathsf{DSL}}\ \wedge\ \mathsf{numChangeTo}\ w \leq 1\\
&\quad \wedge\ tx.\mathsf{outputs} \ominus \mathsf{claimed}(w,tx) = [(a,\mathsf{tip},\text{“”})],\\
&\quad \textbf{where}\ \ w = \mathsf{AndExps}\ [a_1;\ldots;a_k]
\end{aligned}
$$

The DSL ensures that no unintended tokens are minted by a transaction that satisfies a given intent $w \in \mathcal{I}_{\mathsf{post}}$, since the mint field is checked to be equal to MustMint v in w. Similarly, the AndExps clause ensures that no extra, unauthorized outputs can be added to the transaction. That is, every output corresponds to either a PayTo or a ChangeTo constructor in w, or the SP's single fixed tip output. This is checked using the functions claimed and numChangeTo, given in Fig. 12. The same clause also ensures ChangeTo is used only once per specification.

We have developed a minimal DSL here for the purpose of showcasing the operation of the light client protocol. The DSL includes these particular verifications as a demonstration of how intent fulfilment in Cavefish can enforce security properties. To expand the functionality of the light client, the DSL can be swapped for a stronger transaction specification language.

We present two examples for LC intents. The resulting transactions are given in Fig. 1, assuming the intents can be resolved using the contents of the UTxO at the time of transaction construction. Given a ledger state l, we define:

(txi_1) Intent to mint some token t and store it at address a within an interval of length j and maximum fee $f \geq \mathsf{minfee}\ l$: $i_1 = \mathsf{AndExps}$ [MustMint t; MaxInterval j; MaxFee f; SpendFrom (RequireSig k); PayTo $[(a,\ t,\ \text{“”})]$]. This intent allows the SP to pick any outputs at script address RequireSig k to cover the tip.

(txi_2) Intent to interact with a specific smart contract (one with the unique token identifier t and script s), paying value p into it from script RequireSig k_1, and updating its state to d: $i_2 = \mathsf{AndExps}$ [SpendFrom (RequireSig k_1); PayTo $[(s,\ t+p,d)]$; ChangeTo $[(s,\ t+p,d),(a,\ \mathsf{tip},\ \text{“”})]$ (RequireSig k_1)]. Note that ChangeTo's first argument must list *every* other output of the transaction, including the SP's fixed tip, for its leftover-value computation to be accurate.

6 Light Client Protocols

We give an overview on our light client protocol and its components. Our protocol maximizes space and communication efficiency by pushing all storage and lookup

Transaction txi_1**:**

$\mathsf{txi}_1 = \{\ \mathsf{inputs} = \{inp_1,\ inp_2\},$
$\mathsf{outputs} = [(a, t, \text{“”}), (a, \mathsf{tip}, \text{“”})],$
$\mathsf{validityInterval} = [\mathsf{slot}\ l, (\mathsf{slot}\ l) + j],$
$\mathsf{mint} = t,\ \mathsf{fee} = \mathsf{minfee}\ l,$
$\mathsf{sigKeys} = \mathsf{dom}\ (\mathsf{getSigsVal}\ t)\ \}$

Transaction txi_2**:**

$\mathsf{txi}_2 = \{\ \mathsf{inputs} = \{inp_1,\ inp_2,\ inp_3\},$
$\mathsf{outputs} = [(\mathsf{RequireSig}\ k_1, c, \text{“”}),$
$(s, t + p, d), (a, \mathsf{tip}, \text{“”})],$
$\mathsf{validityInterval} = [\mathsf{nothing}, \mathsf{nothing}],$
$\mathsf{mint} = \{\},\ \mathsf{fee} = \mathsf{minfee}\ l, \mathsf{sigKeys} =$
$\{k_1\}\ \}$
where $c = out_1.\mathsf{value} + out_2.\mathsf{value} +$
$out_3.\mathsf{value} - t - p - \mathsf{minfee} - \mathsf{tip}$

Fig. 1. Abstract transaction examples

tasks to the SP whilst guaranteeing the resulting transaction is safe for the light client to sign, and compensating their SP for their effort. 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. A transaction building protocol $\mathsf{Proto}_{\mathsf{tx}}$ is run between two parties, the light client LC with input an intent int_{post} and a keypair PubKey, sk and a service provider SP with access to the ledger state LState_i. The output of the protocol is a transaction tx. We assume that 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* $int_{post} \in \mathcal{I}_{\mathsf{post}}$ *such that: if* $\mathsf{mkToSpec}(\mathsf{LState}_i, int_{post}) \neq \bot$*, then the protocol completes and the transaction* tx' *produced is s.t.* $\mathsf{checkTxL}(\mathsf{LState}_i, tx') = 1$.

We use $\mathsf{checkTxL}(l, tx)$ as shorthand for $\mathsf{checkTx}((slot, fee), utxo, tx)$, with ledger state l containing UTxO state utxo, 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: For a given client pubkey* PubKey *and intent* int_{post}*, if transaction* tx' *is produced by the SP, and* $\mathsf{checkTxL}\ (\mathsf{LState}, tx') = 1$*, then it must be that either* $\mathsf{chkSpec}(int_{post}, tx') = 1$ *or that* $(\mathsf{PubKey}) \notin (tx'.\mathsf{sigKeys})$.

We prove safety in Theorem 3: it follows from the unforgeability of $\mathsf{WBPS}[P]$. Our definition makes two allowances towards the SP.

First, the definition allows the SP to output transactions where the check $\mathsf{checkTxL}\ (\mathsf{LState}, tx') = 0$. Our assumptions on ledger rules ensure that signed transactions involving spent UTxOs will have no effect (as opposed to something undesirable for the user). Second, our definition allows the SP to output transactions that do not involve the user, e.g. by moving the SPs own funds. This is represented by the

Definition 4 (No Freeloading). *A transaction building protocol allows* ***no freeloading*** *if clients cannot gain enough information to build a transaction without completing the protocol with an SP.*

We model the no freeloading property in two ways. First, we define the (stronger) indistinguishable until posted notion, which prevents a client from distinguishing even a single bit of an unknown UTxO. In section 8.1, in order to efficiently support multiple SPOs, we weaken this definition to allow for partial leakage but not enough to allow a client to spend an UTxO.

Definition 5 (Indistinguishable until posted). *A transaction building protocol has the* ***indistinguishable until posted*** *(IuP) 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 $C_{\mathcal{A}}$ using state LState_d. $C_{\mathcal{A}}$ outputs d^*. The adversarial client wins iff $d = d^*$.

We prove this property in Theorem 4: it follows from the weak blindness of $\mathsf{WBPS}[P]$ and the protocol structure.

6.2 (Weakly) Blind Predicate Signatures for Abstract Tx's

To allow the light client to sign an abstract transaction, we use blind signatures on (partially) blinded transaction objects. At a minimum, the SP hides the transaction inputs, without knowing which the LC cannot submit any transaction. This ensures that the LC cannot simply obtain the transaction from the SP, then edit the transaction to remove the SPs compensation.

We formally introduce the notion of *weakly* blind predicate schemes (wPBS) which, similar to full PBS schemes, hide the signed message from the signer but may allow the signer to correlate signatures and signing sessions ex poste. We describe our weakly blind predicate scheme, a more efficient modification of the (full) PBS mechanism in [31].

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 are 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 no freeloading property is met, which translates to the signer being unable to distinguish the message *before* the signature-message pair is posted, or–in case of the multi service provider setting–leakage is limited (see Section 8).

Our construction is inspired by the predicate blind signature (PBS) mechanism in [31] which is compatible with standard Schnorr signatures (or even EdDSA signatures[5] when instantiated accordingly). This is a core property as it allows our protocol to be compatible with popular blockchains such as Bitcoin and Cardano without any modifications.

Our weakening of the PBS notion (WPBPS) brings about the benefit that we can remove certain 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. [12] with the addition of predicate checking. We adapt the definitions of [31] 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 polynomial-time P, the predicate compiler: on input a predicate description $\text{prd} \in \{0,1\}^*$ 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.

- $Setup(1^\lambda) \to \text{par}$: the setup algorithm, on input the security parameter, outputs public parameters par, which define a message space $\mathcal{M}_{\text{par}}$.
- $KeyGen(\text{par}) \to (sk, vk)$: the key generation algorithm, on input the parameters par, outputs a signing/verification key pair (sk, vk), which implicitly contain par, i.e., $vk = (\text{par}, vk')$.
- $\langle Sign(sk, \text{prd}), User(vk, \text{prd}, m)\rangle \to (b, \sigma)$: 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 $\langle Sign(sk, \text{prd}), User(vk, \text{prd}, m)\rangle \to (d, \sigma)$ interaction can be realized by the following algorithms:

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

We use $txt_{X,i}$ to represent the output of participant X in round i and $st_{X,i}$ to represent the internal state of participant X in round i, omitting i when it is

[5] More concretely, signature derivation is non-standard, but verification is unchanged.

not needed. As shorthand, we write $(d, \sigma) \leftarrow \langle Sign(sk, \text{prd}), User(vk, \text{prd}, m)\rangle$ for the above sequence.

Definition 6. *A WBPS scheme* $\mathsf{WBPS}[P]$ *satisfies weak blindness if for all p.p.t. adversaries* $\mathcal{A}$*:*

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

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.

$\underline{\mathsf{BLD}^{\mathcal{A},b}_{\mathsf{WBPS}[P]}(\lambda)}$
$\text{par} \leftarrow Setup(1^\lambda)$
$(m_0, m_1, \text{prd}, vk', st_{\mathcal{A}}) \leftarrow$
  $\mathcal{A}_1(\text{par})$
if $P(\text{prd}, m_0) = 0$ **or**
$P(\text{prd}, m_1) = 0$ **then**
return 0
$\mathsf{aux}_{\mathsf{nt}} \leftarrow \{0,1\}^\lambda$
$m \leftarrow m_b || \mathsf{aux}_{\mathsf{nt}}$
$vk \leftarrow (\text{par}, vk')$
$sess \leftarrow \mathsf{init}$
$b^* \leftarrow \mathcal{A}_2^{\mathsf{ChalUser}}(st_{\mathcal{A}})$
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 $sess = \mathsf{init}$
  $sess \leftarrow \mathsf{open}$
  $(msg', st_u) \leftarrow User_0(vk, \text{prd}, m)$
return msg'

Fig. 2. 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.

Definition 7. *A WPBS scheme* $\mathsf{WBPS}[P]$ *is unforgeable if for all p.p.t. adversaries* $\mathcal{A}$

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

6.3 WBPS Scheme

We present the full weakly blind predicate signature scheme $\mathsf{WBPS}[P]$ in Fig. 4.

6.4 Security

Theorem 1. *The* $\mathsf{WBPS}[P]$ *scheme of Fig. 4 achieves weak blindness when the encryption scheme* PKE *is IND-CPA secure,* NArg *has the zero knowledge property, and the hash generator* HGen *is pseudorandom (Assumption 2).*

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

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

Fig. 3. Chosen Message Unforgeability Experiment $\mathsf{CMA}^{\mathcal{A}}_{\mathsf{WBPS}[P]}(\lambda)$ for a WBPS scheme with predicate compiler P, adversary $\mathcal{A}$

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 $G_0 := \mathsf{BLD}^{\mathcal{A},b}_{\mathsf{WBPS}[P]}(\lambda)$. 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 due to the the IND-CPA security of PKE. This will also change the input of the simulator NArg, 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.

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 2. *The* WBPS$[P]$ *scheme of Fig. 4 achieves unforgeability when the encryption scheme* PKE *is IND-CPA secure,* NArg *is sound, and the Schnorr signature scheme* SDS *is unforgeable w.r.t. the hash generator* HGen *(Assumption 1).*

Proof. We follow the proof of [31] 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 $G_0 := \mathsf{CMA}^{\mathcal{A}}_{\mathsf{WBPS}[P]}(\lambda)$.

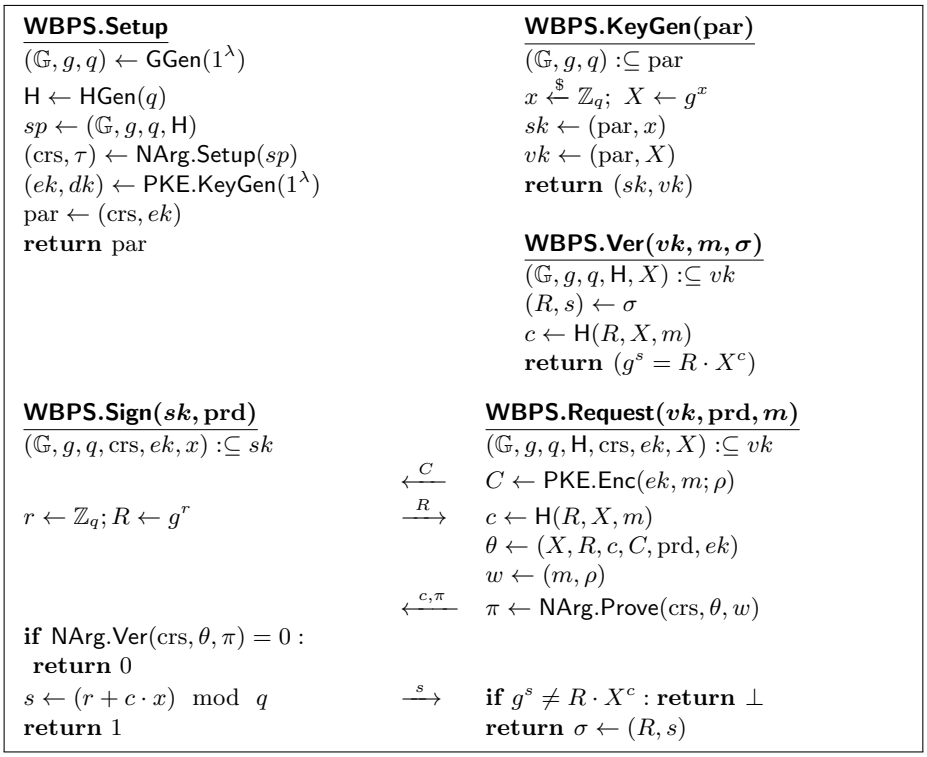


Fig. 4. The weakly blind predicate Schnorr signature scheme $\mathsf{WBPS}[P]$

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 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 $\mathsf{WBPS}[P]$ 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. □

7 Cavefish: Communication-optimal Transaction Building Protocol

We now present the concrete protocol, targeting the UTxO ledger model of Section 4 and the intents of Section 5.

7.1 Security Model and Requirements

Rational Actors. For liveness with a single service provider, 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. We will relax the rational requirement for liveness in Section 8 by introducing multiple SPs.

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.

Such functionality is present in Cardano via validity intervals [1]. In Bitcoin, there exist proposals [56,26] to enable transaction expiry, but it is not yet present. We can also remove this requirement by using multiple service providers, which we describe in Section 8. With multiple servers, the LC will be able to reconstruct a transaction equivalent to the delayed one *and* guarantee that the two transactions are mutually exclusive.

Idempotent Double-spends. For safety, we also require that attempted double-spends are idempotent, i.e., they are simply invalid and do not penalize (slash) the user for signing them.

Explicit Balancing. We require that the transaction specification allows explicit calculation of the value balance of inputs - outputs, commonly referred to as a "tip" to miners or validators. E.g., in Cardano, the tip value is part of the transaction data, and in Bitcoin (post SegWit) it can be calculated from the outputs and the signing data. In pre-SegWit Bitcoin, the calculation was implicit: transaction and signature data only included output values.

Trusted Setup. We specify a proof system using a trapdoor, which is a common characteristic of efficient options such as Groth16 [35] or Plonk [33]. 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 $\mathbb{G}$ element.

7.2 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 7.4). 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.

Our scheme operates as follows: given an intent (i.e. predicate description) $\mathrm{prd} := int_{post}$ and predicate compiler P such that $(P(\mathrm{prd}))(\mathsf{tx}) := \mathsf{chkSpec}(int_{post}, \mathsf{mkAbs}(\mathsf{tx}))$ the requestor (i.e. the SP) encrypts the message, in our case the full transaction $m := tx$, 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 relation

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

checking that $\mathsf{chkSpec}(int_{post}, tx_A) = 1 \wedge m = tx || \mathsf{aux}_{\mathsf{nt}} \wedge C = \mathsf{PKE.Enc}(m; \rho) \wedge c = \mathsf{H}(R, X, m) \wedge tx_A = \mathsf{mkAbs}(tx)$. The signer (LC) completes the signing only if the proof verifies correctly.

Light and Extended Versions. We showcase our protocol for two kinds of intents: "light" and "extended". In the light version, we only support intents where the transaction abstract does not give the light client any information the SP does not wish to disclose. In turn, this makes verification simpler. We can disclose the transaction abstract to the client in the open, and only need to check that the abstract was "filled" correctly in zero knowledge. 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 "light" intents is $int_{\mathrm{light}}(\mathsf{tx}) := \{\mathbf{return}\ (\mathsf{makeAbs}(\mathsf{tx}) = \mathsf{makeAbs}(tx_0))\}$, i.e. the light client fully

determines the requested transaction (including SP fee) to be tx_0, apart from the input UTxOs (which the LC does not know) and the note field aux which is free to the SP to set. This implies that abstract transaction $tx_A = \mathsf{mkAbs}(tx_0)$ provides no new information to the light client. This reduces the computational effort by SP invested in running the NArg prover as $\mathsf{chkSpec}$ consists only of substring equality checks. We note here that no range check is necessary for values: $\mathsf{makeAbs}$ (Section 5) 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.

For the extended version, we alter the predicate compiler to (P_{ext} such that $(P_{\text{ext}}(\text{prd}))(\mathsf{tx}) := \mathsf{chkSpec}(int_{post}, \mathsf{tx})$, i.e., we check tx instead of $\mathsf{mkAbs}(tx)$. Consequently, we set $\mathsf{mkAbs}(x) := \{\textbf{return } 1\}$, and change the predicate check from $\mathsf{chkSpec}$ $(\mathcal{I}_{\mathsf{post}} \times \mathsf{TxAbs}) \to \{0,1\}$ to $\mathsf{chkSpec}$ $(\mathcal{I}_{\mathsf{post}} \times \mathsf{Tx}) \to \{0,1\}$. This ensures that no information is leaked by the abstract transaction. Our results hold for both variants.

We show our complete light client protocol in Fig. 5, leveraging the weakly blind signatures predicate we described in Section 6.3. For simplicity, the figure describes the process for a single UTxO signature; if more are needed, multiple WBPS sessions can be run sharing the same com_{tx}.

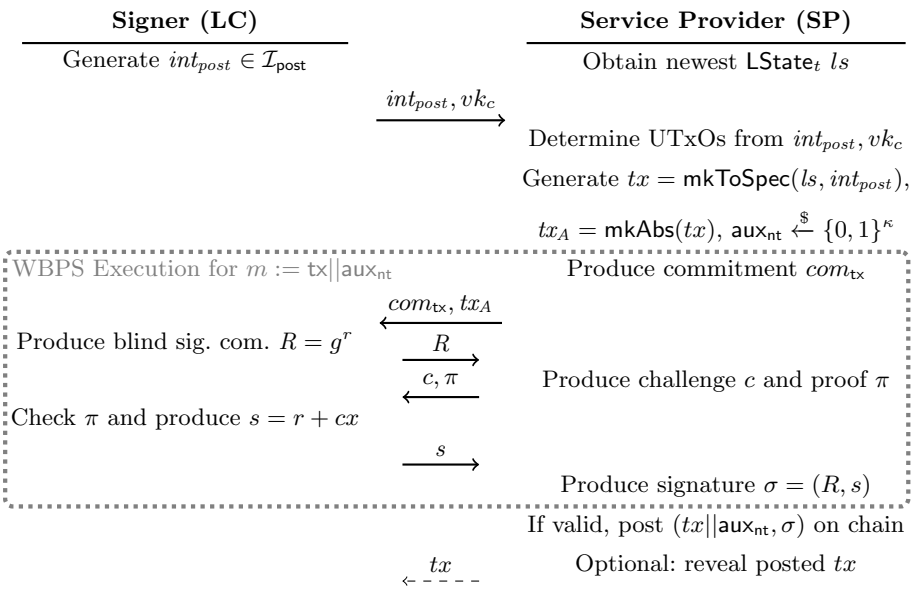


Fig. 5. Cavefish protocol featuring WBPS scheme. (WBPS is detailed in Fig. 4).

7.3 Security

Theorem 3. *If the ledger rules include the* idempotent doublespends *property and* $\mathsf{WBPS}[P]$ *is unforgeable, the protocol of Section 7.2 is safe.*

Proof. A safety violation is a transaction tx' to be produced by the SP, such that $\mathsf{checkTxL}\ (\mathsf{LState}, tx') = 1$, whilst also $\mathsf{chkSpec}(int_{post}, tx') = 0$.

The first clause implies one of three mutually exclussive events: tx' was signed by the LC during the protocol execution (event E1), tx' was not signed by the LC but includes inputs specified by int_{post} (event E2), tx' was not signed by the LC and does not include inputs specified by int_{post} (event E3).

We note that E1 can only hold with negligible probability when we have that $\mathsf{chkSpec}(int_{post}, tx') = 0$, due to the unforgeability property of $\mathsf{WBPS}[P]$ (Theorem 2, Section 6.4). Second, E2 can only hold with negligible probability due to the unforgeability of the Schnorr signature Scheme SDS and the unforgeability property of $\mathsf{WBPS}[P]$. Event E3 is explicitly allowed by the definition.

□

Theorem 4. *If* $\mathsf{WBPS}[P]$ *scheme is weakly blind, the protocol of Section 7.2 is indistinguishable until posted.*

Proof. The indistinguishable until posted property of our protocol is derived from Theorem 1 (Section 6.4) and the protocol structure. Given an adversary against $\mathcal{A}$ IuP, 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 for the messages we use the same index as our guess in the IuP game. □

7.4 Analysis & Extensions

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 [62] 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. We provide details in Appendix G.

Optional Final Protocol Step. The SP is incentivized to submit the LC's transaction to the network by the fee they earn. However, the submission may not be visible to the LC. We may add the one additional step to our protocol: Once the SP observes that the transaction has been included in the chain state (with some level of certainty), the SP sends the full transaction to the LC as evidence that the protocol has been completed.

8 Multi-SP Protocol Model

We present a Multi-SP variant of Cavefish that mitigates some of the limitations that arise when relying on a single SP, most notably, the resulting transactions might be inefficient in the economic sense (e.g. by overpaying for fees or using more inputs than needed), and that a single SP inevitably provides lower liveness guarantees. Revealing the posted transaction (as described above) helps, but only a Multi-SP approach can provide liveness when facing malicious SPs that purposely go silent during/after protocol execution.

Over-costed Transactions. In response to an LC query, it is possible for an SP to construct a *inefficient* transaction (in terms of cost) that 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. Another option is for the SP to cover the transaction fee for the LC, including it in their advertised service price.

Liveness against Unresponsive SPs. Even when an LC sends their intent to multiple SPs, exactly one transaction should be posted on-chain as a result. "Liveness" in this case is the ability to complete the protocol with (exactly) one of the participating SPs, the likelihood of which is increased with the number of honest SPs responding to the intent.

A compromised SP might (pretend to) fail late in the protocol after the LC has signed a transaction. At that point in time, if the LC switches to a different SP to successfully create a transaction, there is a risk of the first SP posting the initial transaction in addition to the one posted by the second SP. To prevent such double payments, we must ensure that transactions produced during a single protocol execution will be mutually exclusive.

As mentioned in Section 7.1, Bitcoin does not support transaction expiry at the time of writing. This shortcoming is remedied by the multi-SP variant of Cavefish, which allows an LC to create many equivalent but mutually exclusive Bitcoin transactions, ensuring that exactly one transaction is posted on chain even if a SP delays or halts the protocol.

Transaction Conflicts We say that two transactions are in *conflict* if they have overlapping sets of inputs. If we ensure that all the SPs' responses to a given LC's intent conflict with each other, whenever the LC completes the protocol with any number of these SPs, there is no chance that the intent may be fulfilled twice, as the blockchain will only allow one of these transactions to be posted. We make use of this in the following protocol variant to balance ensuring liveness with avoiding inefficient transactions and unintentional over-payments.

8.1 Multi-SP Protocol Variant

We can improve liveness by making a small adjustment that turns Cavefish into a multi-SP aware protocol. The core idea for this design relies on two guarantees of the protocol upheld by an honest majority of SPs. The protocol ensures that (honest) SPs will commit to conflicting candidate transactions. If an SP ignores an LC, that client will be able to resume with any SP without risking double payment. That is, an honest majority of SPs perform deterministic transaction construction as prescribed by our protocol, making use of only the settled part of the ledger state.

Definition 8 (Liveness under honest majority). *A transaction protocol is* ***live*** *if for an honest light client, a set of service providers* **SP** *such that the majority of SPs is honest, and any state* $\mathsf{LState}_i \in \mathsf{LState}$ *and intent* $int_{post} \in \mathcal{I}_{\mathsf{post}}$ *so that: if* $\mathsf{mkToSpec}(\mathsf{LState}_i, int_{post}) \neq \bot$*, then the protocol completes and transaction* tx' *produced by the SP is s.t.* $\mathsf{checkTxL}\ (\mathsf{LState}_i, tx') = 1$.

We also need to slightly weaken our indistinguishable until posted (IuP) property to private until posted (PuP). That is, a client will not be able to produce a full transaction without completing the protocol. In order to model the client's lack of knowledge, we have the experiment construct transactions which "pay" the adversary, and require the adversary to successfully spend a UTxO from one of these transactions before the protocol completes in order to win.

Definition 9 (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 non-negligible probability.*

For an honest service provider SP and any client $C_{\mathcal{A}}$ *we have that: The client provides an initial ledger state* LState*, n pairs of public keys* $\mathsf{PubKey}_0, \ldots, \mathsf{PubKey}_n$*, and values* $v_1, ..., v_n$ *as well as a source keypair* $(\mathsf{PubKey}_s, sk_c)$*. If the key pair* PubKey_s, sk_s *is invalid, or if* LState *does not have* UTxO*s addressed to* PubKey_s *of value at least* $\sum v_i$*, the experiment concludes and the adversary loses.*

The experiment then produces a follow-up ledger LState' *by adding transactions to* LState *accordingly so that new* UTxO*s are created, addressed to the specified public keys and with the corresponding values.*

$C_{\mathcal{A}}$ *posts an intent* $int_{post} \in \mathcal{I}_{\mathsf{post}}$*. If* $\mathsf{mkToSpec}(\mathsf{LState}', int_{post}) = \bot$*, the adversary loses and the experiment stops. The experiment runs the protocol between*

SP and $C_{\mathcal{A}}$ using state LState*. $C_{\mathcal{A}}$ outputs tx^*. If the protocol completes, the adversary loses and the experiment stops. At any point before that, the adversary may halt the protocol and instead output a transaction tx^*. The adversarial client wins iff tx^* is valid for* LState' *and spends at least one* UTxO *not in the initial* LState*.*

Ensuring only one signed transaction is posted. In the multi-SP variant of the protocol, the LC might end up signing multiple transactions satisfying its intent. Like the single SP protocol, the multi-SP protocol guarantees that only one signed transaction can eventually be posted on the ledger. Recall here that in UTxO-based ledgers, a transaction is not valid unless all its inputs are present in the ledger state to which it is applied, and an invalid transaction is never partially applied. Therefore, if one transaction from a set of valid transactions with intersecting inputs has been applied to a given state, no other transactions in the set will be able to make any state changes to the resulting updated state (this includes paying fees, etc.).

Since input selection is deterministic, we pick one input that must necessarily be selected by honest SPs for building the LCs transaction to be the *fingerprint* of the transaction. Using this fact, we formulate in brief a version of our protocol where the LC sends the specification to multiple SPs, selecting the response with the most popular fingerprint, and engaging in the rest of the protocol only with SPs that provided responses with that fingerprint. This ensures that the resulting set of candidate transactions has intersecting inputs.

8.2 Augmenting the NArg for Multi-SP

During setup, we select a family of pseudo random permutations F [39], and then, during the second step, after receiving the intent from the client and creating the transaction tx, each service provider will calculate a fingerprint $f \leftarrow F_k(tx)$ of the transaction, where the key k is the last input of tx, i.e., $k := tx.\mathsf{inputs}[-1]$. Note that tx does not yet include the $\mathsf{aux}_{\mathsf{nt}}$ field.

The client receives all the fingerprints and discards all except the most popular one. As we assume an honest majority and deterministic transaction construction, this fingerprint will be the one produced by all honest SPs. Then, it chooses at random one of the SPs that provided the most popular fingerprint value, and initiates the WBPS protocol with it.

The protocol enacts a small modification to the NArg relation (see Equation 1):

$$\mathrm{R}_{\mathrm{M}}(\mathrm{par}, (f, \theta), (tx, \mathsf{aux}_{\mathsf{nt}}, \rho)) := (F(tx) = f) \ \wedge \ \mathrm{R}_{\mathrm{Cavefish}}(\mathrm{par}, \theta, (tx || \mathsf{aux}_{\mathsf{nt}}, \rho))$$

Theorem 5. *If* $\mathsf{mkToSpec}$ *is deterministic and F is sampled from a family of secure pseudo-random functions, the protocol of Section 8.2, instantiated with the* $\mathsf{WBPS}[P]$ *scheme of Section 6.3 is private until posted.*

Proof. (sketch) We will prove this in a series of games. The experiment will first create a fresh keypair $(\mathsf{PubKey}_x, sk_x)$ and create transactions paying into

PubKey_x from the adversarial keypair $(\mathsf{PubKey}_s, sk_s)$ provided to the experiment. It then posts individual transactions from PubKey_x paying into the pubkeys requested by the adversary. The protocol then proceeds as normal. This defines the initial game, G_0. For the first hybrid, G_1, we invoke the zero-knowledge property of the proof system to swap out the proof given to the adversary with a simulated one. Then, for G_2, we invoke the hiding property of the commitment scheme to swap out com_{tx} for a commitment to 0. Finally, for the game G_3, we invoke the pseudorandomness of F to swap out the fingerprint with a random string. By construction, G_3 provides no information to the adversary about any of the new transactions, and the change in adversarial behaviour between G_3 and G_0 is negligible. □

Theorem 6. *If the ledger rules include the* Idempotent Doublespends *property and* $\mathsf{WBPS}[P]$ *is unforgeable, the protocol of Section 8.2 is safe. Additionally, if multiple valid transactions are produced by different SPs, they must be mutually exclusive. That is, if there are two output transactions* tx_0, tx_1 *where* $\mathsf{checkTxL}\ (LState, tx_i) = 1$ *and* $\mathsf{chkSpec}(int_{post}, tx_i) = 1$ *then it must be that* $\mathsf{checkTxL}\ (LState_i, tx_{1-i}) = 0$ *where* LState *is the initial ledger state, and* $\mathsf{LState_i}$ *is* LState *after applying* tx_i*.*

Proof. (sketch) The safety proof is identical to Theorem 3. For the additional property, if there exist two or more output transactions then they must all share the same fingerprint f for the LC to proceed.

The fingerprint f is perfectly binding since F_k is a permutation. Thus, either all transactions sharing f use the same UTxO set, or we derive a $\mathsf{WBPS}[P]$ forgery, which happens with negligible probability. If the transactions use the same input UTxO set, the semantics of updateUTxO (Section 4) imply that the first transaction to be applied to LState will consume its input set, invalidating all others. □

9 Implementation

To assess the efficiency of our construction, we implement and benchmark the NArg component of the protocol in Bitcoin and Cardano. Given today's implementations of zk-SNARKs, the NArg is expected to be the most time and resource intensive part of Cavefish. We base our tests on two zk-SNARK libraries with different proof-systems: *SnarkJS* [36] with *Groth16* [35] and *MidnightZK* [45] with *Plonk* [33]. The circuits we construct and benchmark are inspired by the implementation of [31] and written in the domain-specific language of Circom 2.1 in SnarkJS and using the standard library for circuits of MidnightZK (which uses a Halo2-like frontend). The NArg we implement captures the relation $\mathrm{R}_{\mathrm{Cavefish}}$ (see Sec. 7.2).

We test the "light" version of Cavefish, i.e., the intent is given by $int_{\mathrm{light}}(\mathsf{tx}) := \{\mathbf{return}\ (\mathsf{mkAbs}(\mathsf{tx}) = \mathsf{mkAbs}(tx_0))\}$, where $tx_A = \mathsf{mkAbs}(tx_0)$ is the abstract transaction that tx_0 effectively fully specifies, except for the input UTxOs and the note field $\mathsf{aux_{nt}}$ (which are left blank). We remark that the abstract transaction

$TxAbs$ is part of the known statement θ in $\mathrm{R}_{\mathrm{Cavefish}}$ as it constitutes an input to the predicate P. Therefore, the predicate compiler can be simply set to $(P(\mathrm{prd}))(\mathsf{tx}) := int_{\mathrm{light}}(\mathsf{tx})$ for the this "light" version of Cavefish.

An anonymized repository of our implementation is provided at `https://anonymous.4open.science/r/cavefish-artifact/`.

9.1 Implementation Choices

The "light" version of Cavefish avoids the costs introduced by a fully versatile "extended" version, which requires parsing arbitrary CBOR encoded transactions (in case of Cardano) before verifying `chkSpec`. Pre-specified fields of a CBOR encoding can be parsed efficiently inside a SNARK relation, as [30] showed. Parsing arbitrary structured data is an active line of research with promising results in grammars without context [4], but still unsolved for context sensitive ones. Instead of restricting the benchmarks to pre-determined types of transactions, we opt to showcase a general version supporting arbitrary blinded/redacted parts on the bit-level.

To measure the arithmetic complexity of the relation $\mathrm{R}_{\mathrm{Cavefish}}$, we select BN254 as the curve for Groth16 in SnarkJS and BLS12-381 as the curve for Plonk in MidnightZK. This forces the circuit inputs to be elements of the scalar field of each curve. In the case of transaction tx and abstract transaction tx_A, these are encoded as byte strings of some length n.

We rely on DHIES [2] key encapsulation applied to EC-ElGamal [28] for the encryption scheme PKE used to encrypt tx, i.e., the shared secret serves as a seed to a PRF the outputs of which are used as one-time-pads to encrypt the message. The required PRF can be instantiated with the SNARK-friendly Poseidon hash [34]. Moverover, for efficiently arithmetizable operations, EC-ElGamal is instantitated on Baby-JubJub [61] for BN254 and on JubJub [63] for BLS12-381. The reason is that the base fields of these curves are the scalar fields of BN254 and BL12-381, respectively.

Schnorr challenge. To be compatible with existing blockchain platforms and create standard Schnorr signatures, we have to adopt the hash functions (for signing) specified by the respective ledger, although they are not considered SNARK-friendly. Specifically, signatures in Cardano are produced on $txId$, which is the Blake2b-256 hash of the tx body. In Bitcoin-Taproot[6], what is signed is the SHA2-256 hash of a structured message built on some preimages and fields from the tx body (refer to Appendix F for more details). The final Schnorr challenges are computed using SHA2-512.

We use the library of hash functions in [42] for Circom and the hash gadgets implemented in MidnightZK, while noting that SHA2 and Blake2 implementations show various degrees of optimization, e.g., SHA2 in MidnightZK is highly optimized, whereas Blake2 still has potential for optimization.

[6] Taproot introduced Schnorr signatures in Bitcoin.

Table 1. Benchmark of $R_{Cavefish}$ implementing the "light" version of Cavefish with a 151 B transaction. Sizes in Bytes and multiples. Minimal total Communication includes the total LC↔SP communication cost for the minimal case of one source address, one destination address and value and one change address and a single SP.

Library (proof system)	**SnarkJS** (Groth16)		**MidnightZK** (Plonk)	
SNARK curve	BN256		BLS12-384	
	Bitcoin	**Cardano**	**Bitcoin**	**Cardano**
Signature scheme	Schnorr	EdDSA	Schnorr	EdDSA
Signature curve	Secp256k1	Ed25519	Secp256k1	Ed25519
Setup size	2^{19}	2^{18}	2^{15}	2^{17}
Proving key size	193 MB	116 MB	62 MB	244 MB
Verification key size	65 kB	65 kB	2.93 kB	2.88 kB
Proving time	7.274 s	5.198 s	4.782 s	29.228 s
Proof verif. time	655.47 ms	658.36 ms	11.425 ms	16.144 ms
Proof size	128 B	128 B	6336 B	10832 B
N. of constraints/rows	407518	245572	19949	9810
N. of lookup rows	n.a.	n.a.	15487	65536
Minimal total comms	666 B	712 B	10832 B	11432 B

9.2 Benchmarks

The intent int_{light} and the corresponding partially blind signatures can be implemented with substring equality checks, provided that the transaction can be split into blinded (UTxO inputs) and non-blinded parts. Cardano transactions are CBOR-encoded and contain the $txId$ and indices for every UTxO serving as input [37]. Bitcoin transactions (after Taproot [25]) feature an input count followed by a list of inputs in binary format, each consisting of transaction hash and output index. We assume for our benchmarks that one UTxO acts as input to the transaction and needs to be blinded.

We capture the number of constraints, the proving key size, and the proof size that $R_{Cavefish}$ requires. The number of constraints is obtained from the arithmetization when given as R1CS relation in SnarkJS and Plonkish in MidnightZK. It is worth mentioning that MidnightZK relies on lookup tables to compute the hash functions so additional rows must be included in the Plonkish table. We also measure the time it takes to create the resulting circuit, the proving time and the proof verification time. Our results are summarized in Table 1 for Bitcoin and Cardano as the target platform. The experiments were executed on an AWS EC2 instance of type c6a.12xlarge with 48 vCPUs and 96 GiB of memory.

Despite the non-optimized implementations of the hash functions, Table 1 shows that proving and verifying times are within practical bounds, allowing Cavefish to be deployable in the real world. The first thing to notice is how the SnarkJS circuit for Cardano outperforms the one for Bitcoin. This is due to the several SHA2 instances needed to compute the message to be signed according to Bitcoin Taproot specification. However, when using MidnightZK, the scenario is the opposite. MidnightZK's SHA2 gadget is highly optimized to minimize the number of lookups, while their Blake2b gadget is externally imported at the moment and lacks these optimizations. Another aspect to consider is the length

of the proof. SnarkJS proofs are constant size due to the circuit-specific setup of Groth16. On the other hand, MidnightZK allows for universal setups, but at the cost of variable length proofs. However, due to the use of KZG as PCS, this does not depend on the number of constraints (rows), but on the number of columns, lookups (19 in Cardano vs. 5 in Bitcoin), and permutation arguments. Optimizing Blake2b in MidnightZK can also lead to shorter proofs.

Transaction Size Analysis. For Cardano transactions, we evaluate how the proving times behave for larger circuits. In Figure 6, we show how proving times

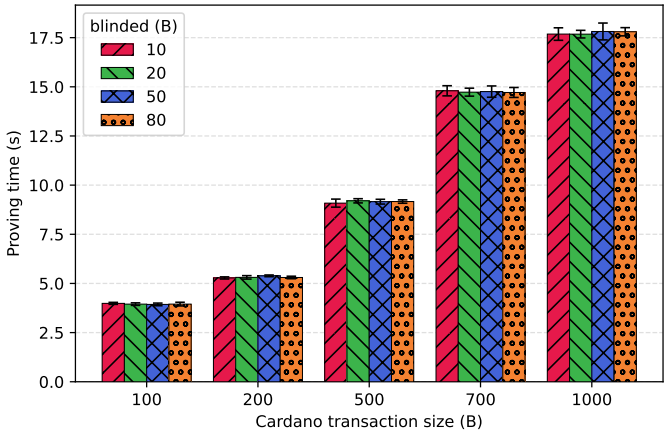


Fig. 6. Proving times for partial blindness given overall transaction size and size of blinded part. Error bars show 95% confidence intervals.

increase depending on transaction size and the size of the redacted/blinded part of the transaction. We use the same AWS EC2 instance as in Section 9.2, i.e., a virtual machine of type c6a.12xlarge with 48 vCPUs and 96 GiB of memory. We measure the proving times for the same NArg relation as in Table 1.

We note that when using SnarkJS (Groth16), the size of the blinded part does not significantly affect the proving times. They are largely governed by the overall tx size that defines the number of rounds in the hash function invocations because this accounts for most of the constraints and thus leads to a more costly and "slower" circuit. We compile the circuits in Circom with signal-to-signal and signal-to-constant optimization (--O1).

Estimated running costs. At the time of writing, the on-demand base pricing for running an Ubuntu Server on our AWS EC2 instance was ∼2USD per hour. With ideal utilization, this implies a cost of around 0.4 cents per proof for Groth16 for either chain. By comparison, the current fees for transactions in Bitcoin [11]

are around 6 cents at 0.5 satoshi/vB for a minimal transaction, or 12 cents at 1 satoshi/vB to ensure faster inclusion. In Cardano [16], the 17 ADA minimum fee is equivalent to ca. 4 cents. In both cases, transaction fees are higher than what a Groth16 proof costs by a factor of around 10. While this is far from a business analysis (as we ignore idle time, aborted sessions, and the cost of lookups and transaction building) our takeaway is that the cost of ZK proofs can be absorbed if the Cavefish fee structure operates in the same order of magnitude as chain fees.

10 Discussion and Real-World Deployment

Cavefish provides a scalable and monetizable framework for intent-based decentralized light clients. Even with an unoptimized prototype implementation, proving and verification times remain efficient enough for the resource-constrained environments of mobile and embedded systems.

A real-world deployment of cavefish would require a ceremony [41] to setup a structured reference string to support a proof system such as Groth16 or Plonk which were used in our prototype implementation. Alternatively, transparent schemes such as STARKS [7] offer competitive performance with proof size being their main weakness. However, as cavefish proofs are never posted on-chain, but only transmitted to the light client, this is a highly attractive use-case.

In a real world implementation, other matters such as dDoS resource exhaustion attacks could become a concern. While this is unexplored in our work, resource exhaustion can be handled on both ends by, e.g., imposing a rate limiting technique and/or a moderate computational cost on the client in the form of (cryptographic) puzzles [3], or by optimizing tradeoffs between proof size and prover effort in proof systems. Cavefish is particularly flexible in adopting any of those mitigations, as proofs are only exposed to the LC rather than on-chain. We leave such optimizations for future work.

We also point out that Cavefish can amplify the risk for use cases that are susceptible to frontrunning. Cavefish itself does not introduce frontrunning (or any derivative value extraction method) as a new attack vector, provided the service provider maintains adequate network connectivity. However, as with any off-chain processing that is handled non-locally, Cavefish extends the window available for potential value extraction, which worsens already existing frontrunning exposure.

Real-world implementations may also require further refinement of the intent-composition DSL, as its specification is inherently tied to the underlying system architecture. Although we have focused on light client interactions, the WPBS scheme and the transaction blinding mechanism could be extended to other blockchain platforms and solver networks. Exploring these additional use cases is a direction for future work.

Finally, it is worth considering whether other types of compensation could replace the service provider fee in our protocol. Instead of including the compensation directly in the transaction itself, the LC could be required to engage

with a specific DApp, cast an on-chain vote, delegate stake, or perform any other action that can be encoded in the transaction the SP sends to the LC for signing.

References

1. Time handling on Cardano — Cardano Docs — docs.cardano.org. https://docs.cardano.org/about-cardano/explore-more/time
2. Abdalla, M., Bellare, M., Rogaway, P.: Dhies: An encryption scheme based on the diffie-hellman problem. IACR Cryptol. ePrint Arch. **1999** (1999)
3. Abliz, M., Znati, T.: A guided tour puzzle for denial of service prevention. In: 2009 Annual Computer Security Applications Conference (ACSAC). pp. 279–288. IEEE Computer Society, Honolulu, HI, USA (2009). https://doi.org/10.1109/ACSAC.2009.33
4. Angel, S., Celi, S., Margolin, E., Mishra, P., Sander, M., Woods, J.: Coral: Fast succinct non-interactive zero-knowledge CFG proofs. Cryptology ePrint Archive, Paper 2025/1420 (2025), https://eprint.iacr.org/2025/1420
5. Armknecht, F., Karame, G., Mohamed, M., Weis, C.: Practical light clients for committee-based blockchains (2024), https://arxiv.org/abs/2410.03347
6. Baldimtsi, F., Madathil, V., Scafuro, A., Zhou, L.: Anonymous Lottery In The Proof-of-Stake Setting . In: Computer Security Foundations (2020)
7. Ben-Sasson, E., Bentov, I., Horesh, Y., Riabzev, M.: Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive, Paper 2018/046 (2018), https://eprint.iacr.org/2018/046
8. Benhamouda, F., Lepoint, T., Loss, J., Orrù, M., Raykova, M.: On the (in)security of ROS. Cryptology ePrint Archive, Paper 2020/945 (2020), https://eprint.iacr.org/2020/945
9. Bernstein, D.J., Duif, N., Lange, T., Schwabe, P., Yang, B.Y.: High-speed high-security signatures. Journal of Cryptographic Engineering **2**(2) (2012)
10. Bitcoin Project: Bitcoin — open source p2p money. https://bitcoin.org/ (2025), accessed: 2025-09-15
11. BitRef: Bitcoin fee calculator. https://bitref.com/fees/, bitRef Bitcoin Explorer. Accessed: 2026-04-30
12. Bobolz, J., Diaz, J., Kohlweiss, M.: Foundations of anonymous signatures: Formal definitions, simplified requirements, and a construction based on general assumptions. In: Financial Cryptography (2025)
13. Bünz, B., Kiffer, L., Luu, L., Zamani, M.: Flyclient: Super-light clients for cryptocurrencies. Cryptology ePrint Archive, Paper 2019/226 (2019), https://eprint.iacr.org/2019/226
14. Cardano Foundation: Cardano. https://cardano.org/ (2025), accessed: 2025-09-15
15. Cardano Foundation: Cardano blockchain explorer (beta). https://beta.explorer.cardano.org/ (2025), accessed: 2025-09-15
16. Cardano Foundation: Transaction fees. https://developers.cardano.org/docs/learn/core-concepts/fees/ (Nov 2025), cardano Developer Portal. Last updated November 14, 2025 by bora. Accessed: 2026-04-30
17. Cardano Team: Full Cardano Ledger. https://intersectmbo.github.io/formal-ledger-specifications/pdfs/cardano-ledger.pdf (2024)
18. Chaidos, P., Kiayias, A.: Mithril: Stake-based threshold multisignatures. In: Cryptology and Network Security (2025)
19. Chakravarty, M.M.T., Chapman, J., MacKenzie, K., Melkonian, O., Müller, J., Peyton Jones, M., Vinogradova, P., Wadler, P.: Native custom tokens in the extended utxo model. In: International Symposium on Leveraging Applications of Formal Methods (2020)

20. Chakravarty, M.M., Coretti, S., Fitzi, M., Gaži, P., Kant, P., Kiayias, A., Russell, A.: Fast isomorphic state channels. In: Financial Cryptography 2021 (2021)
21. Chatzigiannis, P., Baldimtsi, F., Chalkias, K.: SoK: Blockchain light clients. Cryptology ePrint Archive, Paper 2021/1657 (2021), https://eprint.iacr.org/2021/1657
22. Chaum, D., Pedersen, T.P.: Wallet databases with observers. In: CRYPTO' 92. Berlin, Heidelberg (1993)
23. Christopher Goes, Awa Sun Yin, A.B.: Anoma : a unified architecture for full-stack decentralized applications (2022), https://github.com/anoma/whitepaper/blob/main/whitepaper.pdf
24. David Schwartz, Noah Youngs, A.B.: The ripple protocol consensus algorithm (2018), https://ripple.com/files/ripple_consensus_whitepaper.pdf
25. Developers, B.: Bip 340: Schnorr signatures for secp256k1. https://github.com/bitcoin/bips/blob/master/bip-0340.mediawiki (2021), accessed: 2025-05-28
26. Doman, J.: Input-triggered transaction expiry — delvingbitcoin.org. https://delvingbitcoin.org/t/input-triggered-transaction-expiry/2667 (2026])
27. Dubey, A.: 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
28. ElGamal, T.: A public key cryptosystem and a signature scheme based on discrete logarithms. IEEE transactions on information theory **31**(4), 469–472 (1985)
29. Ergo Team: Ergo: A Resilient Platform For Contractual Money. https://whitepaper.io/document/753/ergo-1-whitepaper (2019)
30. Frigo, M., Shelat, A.: Anonymous credentials from ECDSA. Cryptology ePrint Archive, Paper 2024/2010 (2024), https://eprint.iacr.org/2024/2010
31. Fuchsbauer, G., Wolf, M.: Concurrently secure blind schnorr signatures. In: EUROCRYPT 2024. Cham (2024)
32. Gabizon, A., Karantias, K., Kiayias, A.: Plumo: Towards scalable interoperable blockchains using zk-snarks. Cryptology ePrint Archive, Report 2020/399 (2020), https://eprint.iacr.org/2020/399
33. Gabizon, A., Williamson, Z.J., Ciobotaru, O.: Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Report 2019/953 (2019), https://eprint.iacr.org/2019/953
34. Grassi, L., Khovratovich, D., Rechberger, C., Roy, A., Schofnegger, M.: Poseidon: A new hash function for Zero-Knowledge proof systems. In: USENIX Security 21 (2021)
35. Groth, J.: On the size of pairing-based non-interactive arguments. Cryptology ePrint Archive, Paper 2016/260 (2016), https://eprint.iacr.org/2016/260
36. iden3: Circom compiler and snarkjs. https://docs.iden3.io/circom-snarkjs/ (2023), accessed: 2026-04-20
37. IntersectMBO: Cardano ledger specifications and implementations. https://github.com/IntersectMBO/cardano-ledger (2025), accessed: 2025-05-28
38. Joseph Poon and Thaddeus Dryja: The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments. Tech. rep., University of Zurich, Department of Informatics (2010)
39. Katz, J., Lindell, Y.: Introduction to Modern Cryptography. CRC Press, 3rd edn. (2021)
40. Kiayias, A., Miller, A., Zindros, D.: Non-interactive proofs of proof-of-work. In: Financial Cryptography and Data Security: 24th International Conference, FC 2020 , Kota Kinabalu, Malaysia, February 10–14, 2020 Revised Selected Papers. p. 505–522. Springer-Verlag, Berlin, Heidelberg (2020). https://doi.org/10.1007/978-3-030-51280-4_27, https://doi.org/10.1007/978-3-030-51280-4_27

41. Kohlweiss, M., Maller, M., Siim, J., Volkhov, M.: Snarky ceremonies. In: Advances in Cryptology – ASIACRYPT 2021: 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6–10, 2021, Proceedings, Part III. p. 98–127. Springer-Verlag, Berlin, Heidelberg (2021). https://doi.org/10.1007/978-3-030-92078-4_4, https://doi.org/10.1007/978-3-030-92078-4_4
42. Komuves, B.: hash-circuits: Hashing circuits implemented in circom. https://github.com/bkomuves/hash-circuits (2025)
43. Lu, Y., Tang, Q., Wang, G.: Generic superlight client for permissionless blockchains. Cryptology ePrint Archive, Paper 2020/844 (2020), https://eprint.iacr.org/2020/844
44. Malavolta, G., Moreno-Sanchez, P., Kate, A., Maffei, M., Ravi, S.: Concurrency and privacy with payment-channel networks. In: ACM CCS '17 (2017)
45. Midnight Network: Midnight zk proof system. https://github.com/midnightntwrk/midnight-zk (2025), accessed: 2026-04-20
46. Nakamoto, S.: Bitcoin: A Peer-to-Peer Electronic Cash System. https://bitcoin.org/en/bitcoin-paper (October 2008)
47. Pointcheval, D., Stern, J.: Security arguments for digital signatures and blind signatures. J. Cryptology **13**, 361–396 (2000)
48. Schnorr, C.P.: Efficient identification and signatures for smart cards. In: CRYPTO' 89 (1990)
49. Schnorr, C.P.: Security of blind discrete log signatures against interactive attacks. In: Information and Communications Security (2001)
50. Tas, E.N., Tse, D., Yang, L., Zindros, D.: Light clients for lazy blockchains (2024), https://arxiv.org/abs/2203.15968
51. Team, B.: Blockfrost – cardano api platform. https://blockfrost.dev/ (2025), accessed: 2025-09-15
52. Team, E.: Ergo platform. https://ergoplatform.org/ (2025), accessed: 2025-09-15
53. Team, E.: Ethereum development documentation (2023), https://ethereum.org/en/developers/docs/
54. Team, K.: Khalani arcadia: An open platform for intent-driven agent collaboration (2025), https://khalani.network/aip_whitepaper.pdf
55. Team, R.: Raiden network 3.0.1 documentation (2025), https://raiden-network.readthedocs.io/
56. Todd, P.: [bitcoin-dev] OP_Expire and Coinbase-Like Behavior: Making HTLCs Safer by Letting Transactions Expire Safely - Peter Todd — petertodd.org. https://gnusha.org/pi/bitcoindev/{Z}{T}{M}{W}r{J}6{D}jxtsl{J}{B}n@petertodd.org/ (2023)
57. Vesely, P., Gurkan, K., Straka, M., Gabizon, A., Jovanovic, P., Konstantopoulos, G., Oines, A., Olszewski, M., Tromer, E.: Plumo: An ultralight blockchain client (2025), https://celo.org/papers/plumo, white paper
58. VESPR: Vespr wallet. https://vespr.xyz/ (2026), accessed: 2025-10-14; Non-custodial light wallet for the Cardano network
59. Vinogradova, P., Melkonian, O., Wadler, P., Chakravarty, M., Krijnen, J., Jones, M.P., Chapman, J., Ferariu, T.: Structured Contracts in the EUTxO Ledger Model. In: International Workshop on Formal Methods for Blockchains (2024)
60. Wagner, D.: A generalized birthday problem. In: CRYPTO 2002 (2002)
61. WhiteHat, B., Baylina, J., Bellés, M.: Baby jubjub elliptic curve. Ethereum Improvement Proposal EIP-2494 (2020), https://eips.ethereum.org/EIPS/eip-2494

62. Wuille, P.: BIP 0032: Hierarchical deterministic wallets. https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki#user-content-Full_wallet_sharing_m (2012)
63. zcash: Jubjub elliptic curve. https://bitzecbzc.github.io/technology/jubjub/index.html (2018), accessed: 2026-04-20

A Additional Ledger Syntax and Primitives

In Figures 7 and 8 we describe the basic types and primitives used in the UTxO_{ma} model (Fig. 8), as well as the syntax used throughout (Fig. 7), expanding on the description in Section 4.

$\mathbb{H} = \bigcup_{n=0}^{\infty} \{0,1\}^{8n}$	the type of bytestrings
$(a, b) : \mathsf{Interval}[A]$	intervals over a totally-ordered set A
$[a_1; \ldots; a_k] : [C]$	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 with field changed to x
$Key \mapsto Value\ \{\ k \mapsto v\ \mid\ k \in Key,\ v \in Value\ \}$	finite map with unique keys

Fig. 7. Notation

B Transaction Validation Rules

The function $\mathsf{checkTx}((slot, fee), utxo, tx)$ is the conjunction of the following constraints:

(i) **The transaction has at least one input:** $tx.\mathsf{inputs} \neq \{\}$
(ii) **The current slot is in the validity interval:** $slot \in tx.\mathsf{validityInterval}$
(iii) **All outputs have +ve values:** $\forall\, o \in tx.\mathsf{outputs},\ o.\mathsf{value} > \emptyset$
(iv) **All inputs are in the UTxO:** $tx.\mathsf{inputs} \subseteq \mathsf{dom}\ utxo$
(v) **Value is preserved:**

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

(vi) **All inputs validate:** $\forall\, i \in tx.\mathsf{inputs},\ i \mapsto (s, v, d) \in utxo,\ [\![s]\!](tx) = 1$
(vii) **All minting scripts validate:** $\forall\, p \mapsto _ \in tx.\mathsf{mint},\ [\![p]\!](tx) = 1$
(viii) **All sigs are correct:** $\forall\, (pk \mapsto s) \in tx.\mathsf{sigs},\ \mathsf{checkSig}(tx.\mathsf{txid}, pk, s) = 1$
(ix) **The fee is sufficient:** $fee \leq tx.\mathsf{fee}$

LEDGER PRIMITIVES

$\mathsf{checkSig} : \mathbb{H} \to \mathsf{PubKey} \to \mathbb{H} \to \{0,1\}$ *//a given key signed (the hash of) a tx*

HELPER FUNCTIONS

$\mathsf{txid} : \mathsf{Tx} \to \mathsf{TxId}$

$\mathsf{txid}\ tx = \mathsf{hash}\ (tx\ \{\ \mathsf{sigs} = \emptyset\ \})$

$\mathsf{toMap} : \mathbb{N} \to [\mathsf{TxOut}] \to (\mathbb{N} \mapsto \mathsf{TxOut})$ *//construct a map from a list of outputs*

$\mathsf{toMap}(_,\ [\,]) = [\,]$

$\mathsf{toMap}(ix,\ u ::\ outs) = \{ix \mapsto u\}\ \cup\ \mathsf{toMap}(ix+1, outs)$

$\mathsf{mkOuts} : \mathsf{Tx} \to \mathsf{UTxO}$ *//constructs a UTxO set from tx outputs*

$\mathsf{mkOuts}(tx) = \{(\mathsf{txid}\ tx, ix)\ \mapsto\ o \mid ix\ \mapsto\ o \in \mathsf{toMap}(0, tx.\mathsf{outputs})\}$

$\mathsf{MOf} : \mathbb{N} \to \mathbb{N} \to (A \to \{0,1\}) \to [A] \to \{0,1\}$

$\mathsf{MOf}\ k\ m\ f\ [\,] = k \geq m$

$$\mathsf{MOf}\ k\ m\ f\ (h :: t) = \begin{cases} 1 & \text{if } k \geq m \\ \mathsf{MOf}\ (k+a)\ m\ f\ t & \text{else} \end{cases}$$

$$\textbf{where}\ \ a = \begin{cases} 1 & \text{if } f(h)\ \textit{//enough list elements satisfy } f \\ 0 & \text{else} \end{cases}$$

Fig. 8. Primitives and basic types for the UTxO_{ma} model

C Script Evaluation

Figure 9 describes the semantics of Script construction and evaluation. A script $s \in \mathsf{Script}$ specifies constraints on a transaction that must be satisfied in order to allow a specific action the transaction is performing.

D Intent Checking

In Fig. 10, we give the function for constructing an abstract transaction from an intent specification and a given ledger state. We refer to this in Section 5 but relegate this figure to the appendix, because this is a highly simplified prototype used for demonstrating examples of the types of intents that can be processed. Implementing this would not be practical in a realistic system.

E Transactions

The Cavefish protocol requires editing a specific transaction template $\mathsf{initTx}_{a,mf}$, parametrized by the address a of the SP and the amount mf the SP charges for their services. The template pays a tip to the SP and is otherwise empty: $\mathsf{initTx}_{a,mf} = \{\ \mathsf{inputs} = \emptyset,\ \mathsf{outputs} = [(a, \mathsf{tip}, \text{“”})],\ \mathsf{validityInterval} = [\mathsf{nothing}, \mathsf{nothing}],\ \mathsf{mint} = 0,\ \mathsf{fee} = mf,\ \mathsf{aux} = [],\ \mathsf{sigs} = \emptyset\ \}$. We also provide the description of the input selection function mkIns in Fig. 11.

CONSTRUCTORS OF Script

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

EVALUATION OF Script

$$
\begin{array}{lcl}
[\![_]\!] : \mathsf{Script} \to \mathsf{Tx} \to \{0,1\} & & \\
[\![\mathsf{RequireMOf}\ n\ ls]\!](tx) & = & \mathsf{MOf}\ 0\ n\ ([\![_]\!](tx))\ ls \\
[\![\mathsf{RequireSig}\ k]\!](tx) & = & k \in \mathsf{dom}(tx.\mathsf{sigs}) \\
[\![\mathsf{RequireTimeStart}\ t']\!](tx) & = & t' \leq^{?} t_1 \\
[\![\mathsf{RequireTimeExpire}\ t']\!](tx) & = & t_2 \leq^{?} t' \\
[\![\mathsf{NextStep}\ (s,v,d)]\!](tx) & = & (s,v,d) \in \mathsf{outputs}
\end{array}
$$

where $t_1 := \mathsf{proj}_1(tx.\mathsf{validityInterval}), t_2 := \mathsf{proj}_2(tx.\mathsf{validityInterval})$

Fig. 9. Script constructors and evaluation

AUXILIARY mkToSpec′ DEFINITION

$$\mathsf{mkToSpec}' : \mathsf{LState} \to \mathcal{I}_{\mathsf{post}} \to \mathsf{Tx} \to \mathsf{Tx}^{?}$$

$$\mathsf{mkToSpec}'\ l\ (\mathsf{MustMint}\ v)\ (tx) = tx\ \{$$
$$\quad \mathsf{mint} = v + tx.\mathsf{mint},\ \mathsf{sigs} = tx.\mathsf{sigs} \cup \mathsf{getSigsVal}\ v,$$
$$\quad \mathsf{validityInterval} = \mathsf{restrictIntervalVal}\ tx.\mathsf{validityInterval}\ v\}$$

$$\mathsf{mkToSpec}'\ l\ (\mathsf{SpendFrom}\ s)\ (tx) = tx\ \{$$
$$\quad \mathsf{inputs} = tx.\mathsf{inputs} \cup \mathsf{newIns}, \quad \mathsf{sigs} = tx.\mathsf{sigs} \cup \mathsf{getSigsUTxO}\ \mathsf{newIns}\ l,$$
$$\quad \mathsf{validityInterval} = \mathsf{restrictIntervalUTxO}\ tx.\mathsf{validityInterval}\ \mathsf{newIns}\ l\}$$
$$\quad\quad \textbf{where}\ \ \mathsf{newIns} = \mathsf{mkIns}\ l\ (tx.\mathsf{inputs})\ (\mathsf{prod.} - \mathsf{cons.})\ s$$

$$\mathsf{mkToSpec}'\ l\ (\mathsf{MaxInterval}\ i)\ (tx) = tx\ \{$$
$$\quad \mathsf{validityInterval} = (l.\mathsf{slot}, \mathsf{min}\ (l.\mathsf{slot} + i, tx.\mathsf{validityInterval}_2))\}$$

$$\mathsf{mkToSpec}'\ l\ (\mathsf{PayTo}\ [(s,v,d)])\ (tx)$$
$$\quad = tx\ \{\mathsf{outputs} = (s,v,d) : tx.\mathsf{outputs}\}$$

$$\mathsf{mkToSpec}'\ l\ (\mathsf{ChangeTo}\ p\ s)\ (tx) =$$
$$\quad \mathsf{if}\ v > 0\ \mathsf{then}\ tx\ \{\mathsf{outputs} = (s,v,"") : tx.\mathsf{outputs}\}\ \mathsf{else\ nothing}$$
$$\quad\quad \textbf{where}\ \ v = \mathsf{sumVals}\ (tx.\mathsf{outputs}) - \mathsf{sumVals}\ p$$

$$\mathsf{mkToSpec}'\ l\ (\mathsf{MaxFee}\ f)\ (tx) =$$
$$\quad \mathsf{if}\ tx.\mathsf{fee} \leq f\ \mathsf{then}\ tx\ \mathsf{else\ nothing}$$

$$\mathsf{mkToSpec}'\ l\ (\mathsf{SignedBy}\ k)\ (tx) = tx\ \{\mathsf{sigs} = tx.\mathsf{sigs} \cup \mathsf{getSigsKey}\ k\}$$

$$\mathsf{mkToSpec}'\ l\ (\mathsf{AndExps}\ [a_1; \ldots; a_k])\ (tx) =$$
$$\quad \mathsf{mkToSpec}'\ l\ a_k\ (\ldots(\mathsf{mkToSpec}'\ l\ a_1\ tx)\ldots)$$

mkToSpec DEFINITION

$$\mathsf{mkToSpec} : (\mathsf{LState} \times \mathcal{I}_{\mathsf{post}}) \to \mathsf{Tx}^{?}$$
$$\mathsf{mkToSpec}\ (l,i) =$$
$$\quad \mathsf{if}\ \mathsf{numChangeTo}\ i \leq 1\ \mathsf{then}\ \mathsf{mkToSpec}'\ l\ i\ \mathsf{initTx}_{a,mf}\ \mathsf{else\ nothing}$$
$$\quad\quad \textbf{where}\ \ \mathsf{numChangeTo}\ \text{is as defined in Section 5}$$

Fig. 10. Building transactions according to the specific intent

INPUT SELECTION FUNCTION

$$\mathsf{mkIns} : (\mathsf{LState} \times (\mathsf{Set}(\mathsf{TxIn} \times \mathsf{Data})) \times \mathsf{Value} \times \mathsf{Script}) \to (\mathsf{Set}(\mathsf{TxIn} \times \mathsf{Data}))^?$$

$$\mathsf{mkIns}(l, i, v, s) = \begin{cases} i & \text{if } v \leq 0 \\ \mathsf{nothing} & \text{if } v > 0 \wedge \mathsf{pickInput}\ l\ s = \mathsf{nothing} \\ \mathsf{mkIns}(l',\ i \cup \{j\},\ v',\ s) & \text{otherwise} \end{cases}$$

$$\textbf{where}\ \ j = (\mathsf{pickInput}\ l\ s, \text{“”}),\ u = \mathsf{utxo}\ l,\ l' = l \setminus \{j \mapsto _\},\ v' = v - (u(j).\mathsf{value})$$

Fig. 11. Input selection function for transaction construction

AUXILIARY FUNCTIONS FOR $[\![_]\!]_{\mathsf{DSL}}$'S OUTPUT-COVERAGE CHECK

$$\mathsf{claimed} : \mathcal{I}_{\mathsf{post}} \times \mathsf{TxAbs} \to [\mathsf{TxOut}]$$
$$\mathsf{claimed}(\mathsf{PayTo}\ [(s, v, d)], tx) = [(s, v, d)]$$
$$\mathsf{claimed}(\mathsf{ChangeTo}\ p\ s, tx) = [(s, v, \text{“”})],$$
$$\textbf{where}\ \ v = \mathsf{sumVals}\ (tx.\mathsf{outputs}) - \mathsf{sumVals}\ p$$
$$\mathsf{claimed}(\mathsf{AndExps}\ [a_1; \ldots; a_k], tx) = \mathsf{claimed}(a_1, tx) +\!\!+ \ldots +\!\!+ \mathsf{claimed}(a_k, tx)$$
$$\mathsf{claimed}(_, tx) = [\,]$$
$$\mathsf{numChangeTo} : \mathcal{I}_{\mathsf{post}} \to \mathbb{N}$$
$$\mathsf{numChangeTo}(\mathsf{ChangeTo}\ p\ s) = 1$$
$$\mathsf{numChangeTo}(\mathsf{AndExps}\ [a_1; \ldots; a_k]) = \textstyle\sum_{j=1}^{k} \mathsf{numChangeTo}\ a_j$$
$$\mathsf{numChangeTo}(_) = 0$$

Fig. 12. Auxiliary functions used by $[\![_]\!]_{\mathsf{DSL}}$'s output-coverage check

F NArg relations

Below we provide some details which are specific to the NArg relation being instantiated, i.e., blockchain platform-wise. Despite the overall verifications are the same, small adjustments the interface of the NArg relation is the same in the light and extended strategies (as defined in 1), except that in the latter, the public input $TxAbs$ can be empty (nothing about tx is leaked to the LC).

Cardano. There are some implementation-related aspects that must be considered for a deployment of Cavefish in Cardano First, we specify the implementation details to consider when deploying the NArg relation in Cardano. The full set of verifications to be done are provided in the full version.

Bitcoin - P2TR with Key Path Spend. Bitcoin introduces an additional processing to be verified inside the NArg because the transaction signature is not just produced on the hash of the transaction body, but the hash of a specified structured message, the *common_signature_message*, which is built from some transaction fields. The parts of the transaction included in *common_signature_message* depend on the *hash_type*[7] used. For simplicity, we only specify the structure used with SIGHASH_ALL, where all inputs and outputs in the transaction are signed by the LC. See Fig. 13 for all the steps.

[7] `https://learnmeabitcoin.com/technical/upgrades/taproot/#hash-type`

The message to be signed with Schnorr is
$sig_msg = \mathsf{TaggedHash}(''TapSigHash'', common_sig_msg)$,
where $\mathsf{TaggedHash}(tag, msg) = \mathsf{SHA256}(\mathsf{SHA256}(tag)||\mathsf{SHA256}(tag)||msg)$.

Example with $hash_type = \mathsf{SIGHASH_ALL}$**:**

$common_sig_msg =$
$HashType||Version||Locktime||HashPrevouts||$
$HashAmounts||HashScriptPubKeys||HashSequences||$
$HashOutputs||SpendType||InputIndex$

$HashPrevouts = \mathsf{SHA256}(\mathsf{concat}(txid + vout)_{inputs})$
$HashAmounts = \mathsf{SHA256}(\mathsf{concat}(amount)_{inputs})$
$HashScriptPubKeys = \mathsf{SHA256}(\mathsf{concat}(scriptpubkeysize + scriptpubkey)_{inputs})$
$HashSequences = \mathsf{SHA256}(\mathsf{concat}(sequence)_{inputs})$
$HashOutputs = \mathsf{SHA256}(\mathsf{concat}(amount + scriptpubkeysize + scriptpubkey)_{outputs})$

If using light intent:
$TxAbs = \mathsf{mkAbs}(tx_body)$ *//Byte checking of public bytes (all except inputs)*
If using extended intent:
$\mathsf{chkSpec}(int_{post}, tx_body) = 1$
Common NArg checks:
$sig_msg = \mathsf{TaggedHash}(''TapSigHash'', common_sig_msg)$
$C = \mathsf{PKE.Enc}(sig_msg; rho)$
$c = \mathsf{SHA512}(R, X, sig_msg)$
If using HD Wallets:
$o_i = \mathsf{HDDerive}(X, cc, i)$ *//*HDDerive *is some hash function*
$X_i = X \cdot g^{o_i}$ *//Derive challenge c from* X_i *instead of* X

Fig. 13. Bitcoin - NArg verifications

G Details on HD Wallet support

Child addresses are 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). This allows to keep the communication cost between the LC and the SP constant. 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 $\mathbf{cc}$ 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 respect to the sent public key X and the SP can maul the signature as needed.